BACHELOROPPGAVE:

Inexpensive Data Hiding on USB Memory Sticks

FORFATTERE:

Knut Borg
Øystein Nilsen
Rune Søbye

DATO:

Vår 2011

Summary of Graduate Project

Title:	Inexpensive Data Hiding on USB Memory Sticks	Nr:
		Date: Vår 2011
Participants:	Knut Borg	
	Øystein Nilsen	
	Rune Søbye	
Supervisor:	André Årnes	
Employer:	eQ-3 Entwicklung	
Contact person:	Hanno Langweg, hanno.langweg@hig.no	
Keywords		
Pages: 103	Appendixes: 1	Availability: Open

Short description of the main project:

Memory sticks have increased their storage capacity while the price have decreased. Memory sticks are small, can carry lots of information and are easy to carry around. Unfortunately, it is also easy to lose a memory stick wherever the owner for instance is at a Internet cafe or at his/her workplace. Possible adversaries are thieves or nosy colleagues that want to copy read, edit or delete the information stored on the memory stick. One solution users can use in order to prevent people to inspect their memory stick is to use encryption, which is a a method many firms offers today. The downsides with using encryption for protection files is that it is time consuming and the cryptographic tools might be too complicated to use for regular PC users.

The German company eQ-3 Entwicklung GmbH has assigned us to develop a software, in order to find out whether it is feasible to hide files with using a more time efficient method to protect files than encryption. The software is not supposed to be able to hide files from people who can invest in time and forensic tools, but it should be able to prevent regular thieves and nosy colleagues from tempering with the information stored on a memory stick.

The group has analysed the speed of hiding and finding files compared to encryption, and also what can be found with forensic tools and other methods when files are hidden.

The main conclusion of the work done is that the software we have created shows that it is possible and feasible to hide files on a memory stick without the use of encryption.

Sammendrag av Bacheloroppgaven

Tittel:	Inexpensive Data Hiding on USB Memory Sticks	Nr:
		Dato: Vår 2011
Deltakere:	Knut Borg Øystein Nilsen Rune Søbye	
Veiledere:	André Årnes	
Oppdragsgiver:	eQ-3 Entwicklung	
Kontaktperson:	Hanno Langweg, hanno.langweg@hig.no	
Stikkord		
Antall sider: 103	Antall vedlegg: 1	Tilgjengelighet: Åpen

Kort beskrivelse av bacheloroppgaven:
Minnepenner har med tiden fått større lagringskapasiet og blitt billigere. Minnepenner er små og lette, kan lagre mye informasjon og er enkle å frakte med seg rundt for eieren. Dette gjør også at minnepenner fort kan mistes, for eksempel når eieren er på en internettkafé eller på jobb. Mulige motstandere er tyver eller nysgjerrige kollegaer, som ønsker å kopiere, slette, lese eller endre på informasjonen som er lagret på minnepennen. En løsning som kan brukes for å beskytte data lagret på en minnepenn er kryptering av data, som er en løsning mange firmaer leverer i dag. Hovedproblemene med kryptering er at det er tidkrevende og at verktøyene som brukes for å utføre krypteringen kan være vanskelige å forstå for vanlige PC-brukere.

Det tyske firmaet eQ-3 Entwicklung GmbH har gitt oss i oppgave å lage et program for å undersøke om det er hensiktsmessig å gjemme filer med bruk av mindre tidkrevende metoder enn kryptering. Det er ikke meningen at programmet skal kunne beskytte mot kunnskapsrike motstandere, men det er forventet at det skal gi tilstrekkelig beskyttelse mot vanlige tyver og nysgjerrige kollegaer.

Gruppa har analysert programmet med tanke på hvor raskt det er og hva som kan oppdages med forskjellige dataetterforskningsverktøy og andre metoder når filer er gjemt.

Hovedkonklusjonen på oppgaven er at programmet vi har laget viser at det er mulig og hensiktsmessig å gjemme filer på minnepenner uten bruk av kryptering.

Preface

We chose this project because we thought it sounded interesting to hide information on memory sticks without using encryption as well as gaining experience in creating a software from scratch. We also thought it would be interesting to learn how hardware is managed at a low level by Windows, as well as getting more into details about the FAT32 file system.

The bachelor thesis has been challenging, but instructive. We have learned a lot about the FAT32 file system and how to make a bigger software and GUI. There are several people we would like to thank:

- Hanno Langweg for constructive criticisms and supervising during the entire project period.
- Our supervisor André Årnes for giving us ideas on how we should approach different problems.
- Our employer (represented by Dirk Stüben) for answering question during the development period, for providing us with memory sticks free of charge and testing the final version of the software.
- Simon McCallum for advice about C# and DLLs.
- Cathrine Fjeldstad for proofreading the bachelor thesis report.

Contents

List of Figures

List of Tables

Abbreviations and definitions

Abbreviations

- AES - Advanced Encryption Standard
- API - Application Programming Interface
- BIOS - Basic Input/Output System
- BPB - BIOS Parameter Block
- CPU - Central Processing Unit
- DD - Data Description
- DLL - Dynamic-Link Library
- FAT - File Allocation Table
- GB - Gigabyte = 2^{30} bytes
- GUC - Gjøvik University College
- GUI - Graphical User Interface
- KB - Kilobyte = 2^{10} bytes
- MS - Microsoft
- MSVS - MS Visual Studio
- NTFS - New Technology File System
- OS - Operating System
- RAM - Random Access Memory
- TB - Terabyte = 2^{40} bytes
- TSK - The Sleuth Kit
- USB - Universal Serial Bus
- WE - Windows Explorer

Definitions

- Bad cluster - Cluster which can not be allocated.
- Binary data - Data which is represented by 0 or 1.
- Binary file - A file consisting of binary data.
- Cluster - A group of sectors.

- Cryptography - The knowledge of hiding information.
- dd - Unix program that copies data
- Decryption - Making encrypted information readable.
- Encryption - Protecting information by making it unreadable.
- File system - A system for storing information.
- File system forensic tools - Tools for analysing contents on storage devices.
- Hex editor - Program to view binary files.
- Image - A file containing the bytes of a hard drive.
- ls-command - Lists files in Unix file systems.
- Memory stick - A portable device for storing data.
- Open source - The source code is available for anyone.
- Root cluster - Defined as the cluster where file names are stored.
- Root cluster chain - A linked list of root clusters.
- Sector - A variable sized collection of bytes on a storage device.
- Steganography - Hiding data from plain sight by obscuring it.

1 Introduction

USB (Universal Serial Bus) memory sticks are common devices to store information on. They are small, lightweight and easy to carry around, which makes them susceptible to loss of data. We often hear stories about sensitive information found on lost memory sticks where no form of protection like encryption are implemented [1, 2]. Memory sticks are also easy to search through without the knowledge of the owner, for instance when the memory stick is lent out to a friend or a colleague.

There has always been a need to hide information from others, and there are several ways of doing it. There is steganography, where information is hidden, and cryptography, where the information is encrypted so it is unrecognizable and practically impossible to recover without a decryption algorithm and a decryption key. Our method is based on hiding information from plain sight, so it is similar to steganography. There is however a high possibility that the hidden files can be found if the attacker has forensic tools and experience.

There are methods to encrypt data and then hide it (the encryption program TrueCrypt [3] has a function that can encrypt files within other encrypted files, so that a user can claim plausible deniability [4]. If an encrypted message is found, it is still unreadable for anyone who does not know the key to decode it. One of the problems concerning encryption is the time consumed when large amounts of data are being encrypted. There is also the problem that it is practically impossible to recover the data if the user has forgotten the key needed to perform the decryption.

Further details about speed of encryption compared to our software can be found in chapter 6.2

1.1 About the layout and structure of the report

Since the assignment is more like a software development assignment in practice than a pure information security assignment would be, the report will have different sections describing the development process in addition to the analysis part. Source code samples will be marked and formatted with a different font and have syntax highlighting.

The report is divided into the following chapters:

- Chapter 1 describes the software, the users of the software and possible adversaries, as well as the goals of the project.
- Chapter 2 describes our background materials for the work done, in addition to software used. Finally, we list which means of quality control we used during the software development period.
- Chapter 3 describes the FAT32 file system and the logics behind it.

- Chapter 4 describes the development process.
- Chapter 5 gives development results, including a description on how we managed to manipulate the FAT32 in order to be able to hide and recover files.
- Chapter 6 gives an analysis of the software, where it is compared with cryptographic means of protecting files. Also, known program bugs are described in this chapter.
- Chapter 7 gives suggestions for future work.
- Chapter 8 contains discussions and evaluations, including criticism of the bachelor thesis and evaluation of the work done by the group.
- Chapter 9 contains the conclusions of the work done.

1.2 Target audience

The software is developed for a German company named eQ-3 Entwicklung GmbH and is intended to be easy to use for people with average computer skills. This report is about the theory behind the software and how we worked to solve the problems we encountered during the development process.

1.3 Objectives

This bachelor thesis is about hiding files on USB memory sticks with the FAT32 file system without the use of encryption. The reason for avoiding encryption is because it is a resource heavy operation, and for instance encrypting an entire 32 GB memory stick might take a while depending on how powerful the computer is, as seen in chapter 6.2. It may also not be trivial to use encryption for people with lacking experience regarding cryptographic tools. Therefore, it could be beneficial to users if they had access to software which hides files without using complicated cryptographic tools. The use of this software should be less time consuming and more user friendly than encryption. However, without the use of encryption, the software will not prevent attackers with file forensic tools and knowledge at their disposal to find the hidden files on the memory stick. The software is only supposed to hide files from people who might do a quick inspection of the memory stick, like using Windows Explorer (WE) to view files.

1.4 Our knowledge background

The group members are studying for a Bachelor's degree in Information Security at Gjøvik University College (GUC). This project is however mainly a software development project, and none of the group members have had any experience with writing bigger software other than a course in software engineering. To be able to create a program that hides and recovers files, be user friendly and have a graphical user interface (GUI), we had to learn a lot of new things that had not been covered by our curriculum. The FAT file system had to be learned in detail. Although all of us attended an operating system course, it did not go as much into detail about file systems as we needed for this bachelor thesis. Therefore, we had to study the FAT32 specification [5] thoroughly before we were able to manipulate the file system. We also needed to familiarize with the file forensic tool kit The Sleuth Kit (TSK) [6] for analysis purposes. In addition to that, we had to learn the Windows API (Application Programming Interface), which were needed for controlling the writing to, and reading from, the memory sticks. We had to figure out which functions we needed. Furthermore, we had to understand what the functions and the parameters to those functions did and how to use them. We read a lot of documentation on MS websites [7] to get an understanding of the standard system functions.

1.5 Choice of programming language

We had a choice of writing the software in either C, C++, Java or C# (all these languages could use Visual Studio 2008, which was a requirement). Our decision was to program in C, since it is low-level compared to other programming languages and is the most common language for controlling hardware, which was an important part of the coding in this project. By low-level, we mean access to functions that control the hardware, like device drivers.

None of us were familiar enough with C# for it to be a viable option for low-level programming. C# was however used to create the GUI, since it was the simplest option. More details about the choice of C# can be read in chapter 5.6.

1.6 Framework

Memory sticks and forensic tools were needed for testing and analysis. The report is written in LaTeX and the source code in Microsoft (MS) Visual Studio 2008. We used our own computers for development and computers at GUC for testing of the software. The costs were minimal, all we needed was five memory sticks for testing and development purposes.

1.7 Effect goals

Effect goals are the goals we believe might be the result of the finished product:

- A well made product will lead to good publicity for the employer, because the product reflects how the company handles quality assurance. Also, a good product itself reflects the quality of the employer's products as a whole.
- The customers' data will be better protected against people who do not invest a lot of time and effort investegating the memory stick for hidden data.

1.8 Result goals

Result goals represents what should be accomplished at the end of the project:

- Make an intuitive GUI, which means the user should not have to read the manual before he can hide or recover files. The GUI is the interface used for communicating with the software.
- Make a product that is of such quality that the employer chooses to share it with their customers.
- Be able to hide and recover files in a safe and efficient manner without the risk of loosing data.

1.9 Scope

The main tasks of this bachelor thesis are to:

- Develop a software which is able to hide and recover files.
- Create a simple GUI, in order to make the software user friendly.
- Analyse the quality of protection provided by the software.

1.10 Description of group members, advisor and employer

All group members are students taking bachelor in Information Security at GUC. The main responsibilities for the members during the project period was:

Knut Borg - Group leader, coding. Øystein Nilsen - GUI, analysis, latex. Rune Søbye - Group leader vicar, coding, secretary.

Our employer was the German company eQ-3 Entwicklung GmbH.

The employers liaison supervisor at GUC: Hanno Langweg. Associate Professor, Dr. rer. nat. Had weekly meetings with us during the project period, where feedbacks of current state of software and suggestions for further work were given.

Contact person at employer: Dirk Stüben. Answered questions related to the software and it's source code.

Supervisor: André Årnes. Adjunct associate professor, PhD. Gave ideas on how to approach different problems in our bachelor thesis, along with advices regarding the bachelor report.

2 Background

We received a proof-of-concept code, shown in appendix H which read every sector from a memory stick. After sectors had been read, the program created a binary file which we could use to inspect the content on the memory stick with a hex editor. The proof-of-concept code was intended to give us an idea about how the interaction with a memory stick works, along with how we could inspect the memory stick, so we did not have to start from scratch.

We were also recommended to read the paper “Methods of Information Hiding and Detection in File Systems” by J. Davis, J. MacLean and D. Dampier [8]. The paper describes several concepts of hiding information on both NTFS and FAT systems. The methods described varies from using “hide file” in WE (Windows Explorer) to creating a separate partition which then could appear as hidden. The paper also mentioned hiding files by using bad clusters and we used it as an inspiration for our development process. The main difference between the paper’s suggestion and our prototype is that we hide everything on the memory stick instead of individual files.

2.1 About the FAT32 file system

FAT stands for File Allocation Table, and 32 means that each entry in the FAT consists of 32 bits [5]. Other FAT file systems are FAT12 and FAT16, but these are not relevant for this assignment. FAT32 only supports files up to 4 GB [5], and a FAT32 volume can not be any larger than 2 TB due to the 32-bit addressing of clusters [5]. More details about clusters and other aspects regarding the FAT32 file system will be discussed in chapter 3.

2.2 The development process

In this chapter we will describe various aspects regarding the development process. We will describe the development used, which software we needed and which internal and external quality control measures we used during the development period.

Choice of development model

We wanted to use a system development model where we could change the system requirements during the development process, and change the source code if unexpected problems occurred. The evolutionary model fitted our project well because it provided the flexibility to switch between specification, development and validation. Also, it made it possible for the employer to give us weekly feedback on our work during the development period. We presented several prototypes to the employers liaison supervisor at GUC (Gjøvik University College) and each of the prototypes were an improvement of the previous ones.

We decided what the prototypes should contain at the beginning of the working process, in collaboration with the employer's liaison supervisor at GUC. By using the evolutionary model, we improved the software through several iterations, until we ended up with a final version.

Software used
We used the C programming language to develop the software. MS Visual Studio 2008 was used as programming environment, since the employer's quality assurance system needed to be compatible with MS VS 2008. The program XVI32 (version 2.4) [9] was used as hex editor for studying image files of the memory sticks. The Sleuth Kit was used for analysis of how much can be detected after files have been hidden. For keeping track of bugs and issues regarding the software development, we used Mantis [10]. Subversion was used as repository and revision control. [http://subversion.apache.org/]

Quality control

Below, we will list different internal and external quality controls measures used during the development period.

External quality controls includes:

- Employer performed a thorough check of the software and gave us reviews. [appendix?]
- Regular meetings with employer, with given feedback of current version of software.

  Internal quality controls includes:

- Mantis was used to keep track of bugs and prioritize them.
- We created a script which executed the software 400 times, used to analyse performance.
- Tested software on different MS OSes, both 32 and 64 bits.
- Bug testing continuously, as we developed the source code.
- Documenting and commenting of code.
- Wrote manual to software.
- Work logs.
- Version control and storage of files on Subversion.
- Group rules.

2.3 Description of the software

Our assignment was to create a software that is able to hide data on a USB memory stick. This will protect the files from being read or copied by people with no experience with forensic tools and knowledge about file systems. The software is also supposed to allow adding files on a memory stick with no danger of overwriting the hidden files, making the operation as risk free, for the user, as possible.

One main requirement from the employer's side was that the software had to be very user friendly. It should have a simple GUI that anyone with minimal computer experience should be able to use. The software is meant to be distributed on the employer's website, and it might be released as open source. The installation program was provided by the employer. We had to develop the software within the standards given by the employer.

The main requirements for the software as we understood it when we wrote the project plan, was:

- The software must be able to be used without any further preparations by the user than installing it and being in possession of a memory stick.
- It must be user friendly. It should be easy and intuitive for all kinds of users to hide files and later recover them.
- It must be reliable. It should work every time, as well as be able to handle unexpected situations.
- It has to be able to hide files and recover files hidden by the software on the FAT32 file system.
- The memory stick must be accessible and able to be used after files have been hidden.
- It is important that the user does not need to wait for a long time in order to hide or recover files, so the process of doing so must be inexpensive. Inexpensive means there will be no use of cryptographic algorithms either by software or hardware implementations, making the process of hiding or recovering files able to finish in a short amount of time which equals no longer than 5 to 6 seconds.
- It should be impossible to detect the hidden files when using computers running either Windows, Mac OS X or various Linux distributions.
- When an illegitimate user tries to access the memory stick, he should not be able to discover hidden contents or suspect that files are hidden. This is only the case if the user is not in possession of any advanced forensic tools or skills regarding the FAT32 file system.

2.4 Description of users

The software can be used by any PC user with a need of protecting his data, without using time-consuming and non-trivial encryption methods. A typical user of the software is a regular person who wants to hide files from colleagues or other people, but do not want to use a lot of time, money or effort hiding it.

2.5 Description of attackers

A typical adversary might be a colleague, a friend or other people without forensic experience or forensic tools at their disposal. A typical attack may be if a person is using the memory stick with, or without, the owner's consent, searching for files and directories via Windows Explorer or the equivalent Finder for Mac OS X or the ls-command in Linux.

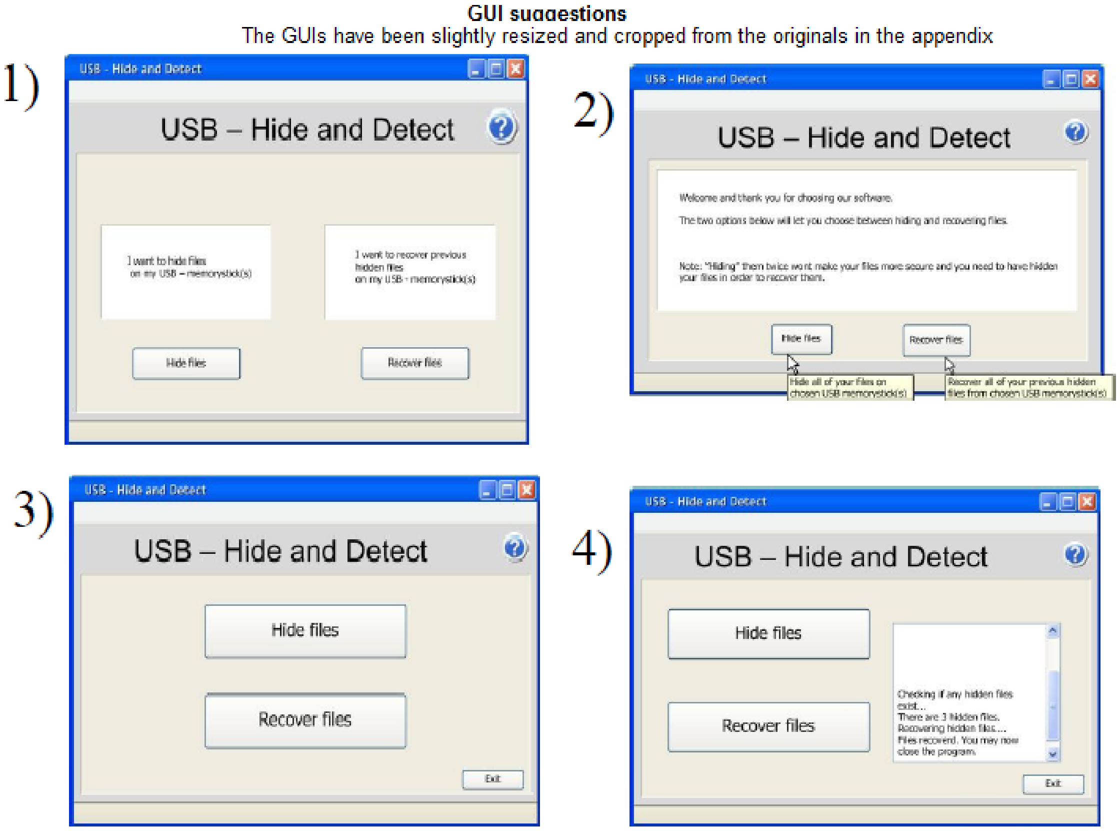
2.6 Limitations

These are the limitations we set for the software we developed:

- The software is developed to be compatible with the FAT32 file system.
- The software is developed to run only on the following operating systems:
  - MS Windows XP
  - MS Vista
  - MS Windows 7
- Due to time limitations, the option of hiding or recovering a given selection of files can not be supported.

2.7 GUI

For user-friendliness, the software needed to have a GUI. We started off with creating four basic looking GUIs, which can be studied in appendix B. Figure 1, shows the four GUI suggestions we made.



GUI suggestions

The GUIs have been slightly resized and cropped from the originals in the appendix

I want to hide files
on my USB – memorystick(s)

I want to recover previous
hidden files
on my USB - memorystick(s)

Welcome and thank you for choosing our software.

The two options below will let you choose between hiding and recovering files.

Note: "Hiding" them twice wont make your files more secure and you need to have hidden your files in order to recover them.

Hide all of your files on
chosen USB memorystick(s)

Recover all of your previous hidden
files from chosen USB memorystick(s)

Checking if any hidden files
exist...
There are 3 hidden files.
Recovering hidden files....
Files recoverd. You may now
close the program.

Figure 1: Our four suggestions for GUI.

GUI number one and two looked quite similar to each other, while GUI number three and four had clear differences between the rest of the GUIs. GUI number four was the most complicated option because of the window with log messages, while GUI number three, with only a couple of buttons, was the GUI we considered to be easiest to implement.

In order to get a different point of view on the GUI suggestions we created, we decided to ask people outside of the group for opinions on which GUI that would suit best. The appendix presents the overall results for each GUI. [Henvisning til meningsresultater om gui Knut] The research in the appendix referenced above, should under no circumstances be considered as a valid survey since we only asked ten friends who works with IT on a daily basis. However, it gave us helpful suggestions when we were about to create the GUI.

To summarize the results; the general first impressions was that most people wanted GUI number four with information of what is going on. After a short discussion, people ended up with wanting a merged solution of GUI number two and GUI number four. Only a few people we asked would rather use the GUI number three, but only if it displayed a progress bar if the hiding and recovering process lasted too long. Our conclusion was to start with implementing a GUI similar to model number two and then add the information log box from GUI number four if the time allowed it. More about the final GUI, including how it looks, can be found in chapter 5.6.

3 Description of the FAT32 file system

This chapter includes definitions of the fundamental components of the FAT32 file system. We will explain how files are stored and represented on FAT32. Various figures will be given during the chapter.

The FAT (File Allocation Table) file system was specified by Microsoft and has three variations, FAT12, FAT16 and FAT32 [11]. FAT32 is the most common, and the two other are rarely used today. The FAT file system is simple in its implementation compared to most other file systems [11]. It is divided into three main regions, the reserved region, the FAT region and the data region. In the reserved region, basic information about the file system on the memory stick can be found. The FAT works like a linked list, where the entries point to the next cluster until end of file. More details and a figure illustrating the linking mechanism will be provided in chapter 3.2.3.

Finally, there is the data region, which is the largest region. In that region, all the file names and their contents are stored.

3.1 Reserved region

The reserved region is the first region on the FAT32 file system. It, amongst others, contains the BPB (Basic input/output system Parameter Block) [5]. The BPB is stored in the very first sector of the reserved region, and contains several important values, used by the file system [5]. Some of these values needed to be extracted for use in our code. The values used by our software will be shown in table 1 in chapter 4.1.

3.2 Sectors

A sector is a logical region on the FAT32 file system. The legal size of the sectors in FAT32 are 512, 1024, 2048 and 4096 bytes [5]. The MS FAT specification, however, states that “if maximum compatibility with old implementations is desired, only the value 512 should be used” [5]. The sector count starts with sector number zero and counting upwards.

3.3 Clusters

A cluster is a collection of sectors in the data region, working as an region where a given amount of data are stored. The number of sectors per cluster may vary depending on disk size, or whether the user has reformatted the disk. However, the number of sectors in a cluster must be a power of 2 and greater than zero, where the legal values are 1, 2, 4, 8, 16, 32, 64 and 128 [5]. The number of sectors on a memory stick is stored in the BPB, in an entry named ’BPB_SecPerClus’ [5]. As with the sectors, the cluster count starts with cluster number zero. This means that cluster number two is the third cluster in the FAT region, but we refer to it as cluster number two.

In this report, we regularly uses the term bad cluster. A bad cluster is a way to prevent the computer to store data in that spesific cluster, in order to prevent disk errors [4]. Bad clusters are marked with the value [F7] [FF] [FF] [0F].

3.4 The FAT

The FAT is a table consisting of several entries. Each entry consists of 32 bits, hence the name FAT32, and represents one cluster in the data region [5]. The main tasks of the FAT are to keep track of which clusters are available for storage, and to represent in which clusters different files are stored [5].

The first two clusters, which is the first eight bytes (starting from 0) in the FAT, are not used for storage of data [5]. Because of that, the first cluster with file-specific contents is cluster number two, which is equivalent to bytes 8, 9, 10 and 11 in the FAT. This cluster also the cluster also serves as the default root cluster in the FAT file system. The default value is stored in the BPB. This means that the storage of file names starts in cluster number two and that the corresponding FAT entry (bytes 8, 9, 10 and 11 in the FAT) is given the value [FF] [FF] [FF] [0F], which indicates the end of the cluster [5]. When cluster number two is filled up with file names, additional files will be stored in another free cluster. The FAT32 file system will then change the entry value in the FAT representing cluster number two, from [FF] [FF] [FF] [0F] to the number of the next cluster used for storage of file names. In addition, the entry of the new cluster where the chain ends will be changed to [FF] [FF] [FF] [0F].

Regarding files and folders, each of them has a field in its directory entry, indicating the cluster where they are stored. In the corresponding FAT entry for a specific cluster, several different values might be stored. The value [FF] [FF] [FF] [0F] means that the cluster is the final cluster containing the file. If the contents of a file or folder cannot be stored in one single cluster, the corresponding FAT entry will contain the value of the next cluster where the file is stored. This way, every file or folder will have a chain of clusters from the cluster where it begins, continuing to the last cluster where the chain ends with the value [FF] [FF] [FF] [0F].

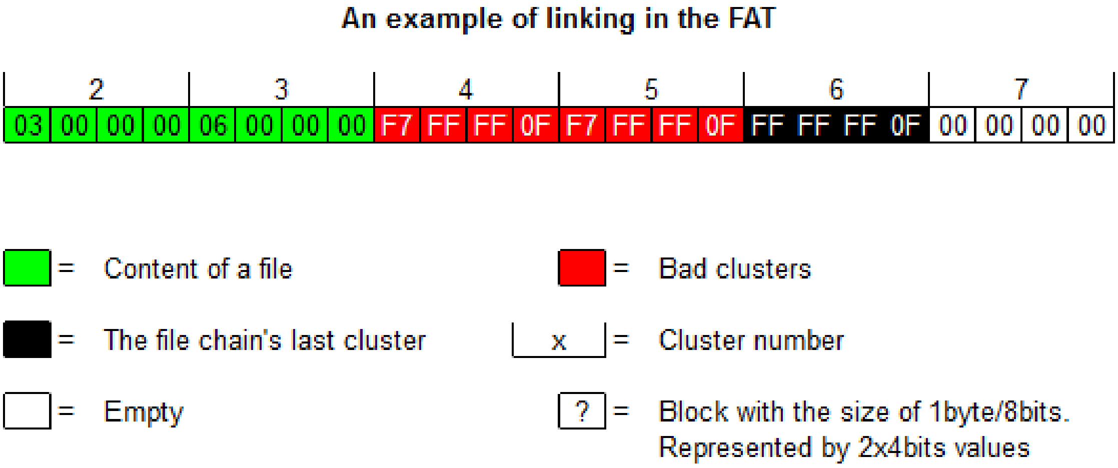

An example of linking in the FAT
2
3
4
5
6
7
03 00 00 00
06 00 00 00
F7 FF FF 0F
F7 FF FF 0F
FF FF FF 0F
00 00 00 00
= Content of a file
= Bad clusters
= The file chain's last cluster
x = Cluster number
= Empty
? = Block with the size of 1byte/8bits. Represented by 2x4bits values

Figure 2: How a file chain works.

As displayed in the Figure 3, cluster number two points to cluster number three. Since both cluster four and five is marked as bad, cluster three points to cluster number six, which contains the value [FF] [FF] [FF] [0F] and hence is the final cluster in the chain.

3.5 Data region

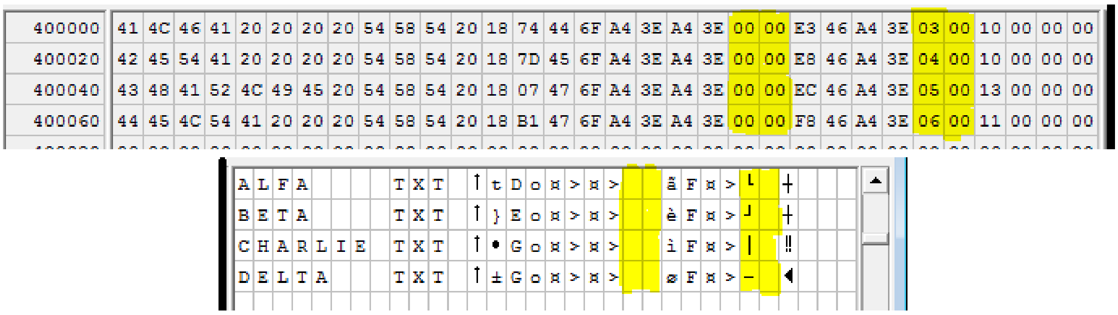
The data region is divided into separate clusters with either a default size or a size chosen by the user. Each cluster has its own address in the FAT. The region stores directory structures, file names and file contents.

3.6 Representation of file names

The FAT32 file system handles long and short file names in different ways. In this section, we will give a description of how short and long file names are represented by the FAT32 file system. How FAT32 handles file names was an important part of this project, since details about the file name representation needed to be understood when implementing the NameCheck() function described in chapter 5.5.

Short file names cannot have more than 11 characters, where up to the 8 first bytes (0-7) contain the file name itself and the three last bytes (8-10) contain the file extension. All files with a short file name have one, and only one, corresponding directory entry [5]. The directory entry for a short file name consists of 32 bytes, where the main contents are the file name, file attributes, various time stamps and meta data about the file [5].

In figure 4, we can see how four sample files with short file names are represented by the FAT32 file system.



```
400000  41 4C 46 41 20 20 20 20 54 58 54 20 18 74 44 6F A4 3E A4 3E 00 00 E3 46 A4 3E 03 00 10 00 00 00
400020  42 45 54 41 20 20 20 20 54 58 54 20 18 7D 45 6F A4 3E A4 3E 00 00 E8 46 A4 3E 04 00 10 00 00 00
400040  43 48 41 52 4C 49 45 20 54 58 54 20 18 07 47 6F A4 3E A4 3E 00 00 EC 46 A4 3E 05 00 13 00 00 00
400060  44 45 4C 54 41 20 20 20 54 58 54 20 18 B1 47 6F A4 3E A4 3E 00 00 F8 46 A4 3E 06 00 11 00 00 00
```

```
A L F A       T X T   ↑ t D o ¤ > ¤ >     ã F ¤ > └   ┼
B E T A       T X T   ↑ } E o ¤ > ¤ >     è F ¤ > ┘   ┼
C H A R L I E T X T   ↑ • G o ¤ > ¤ >     ì F ¤ > │   ‼
D E L T A     T X T   ↑ ± G o ¤ > ¤ >     ø F ¤ > ─   ◄
```

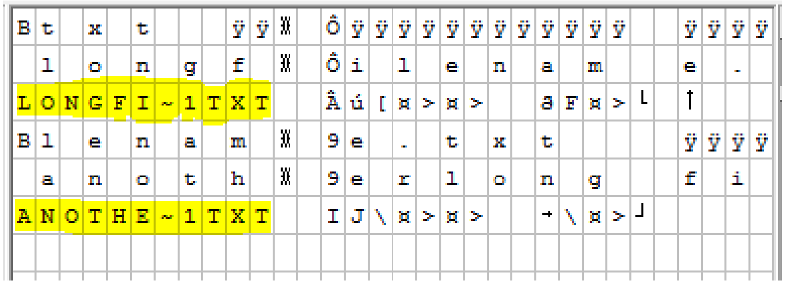
Figure 3: Short file names

In the upper part of the figure, we can see how the files are represented in hexadecimal values, and in the lower part the contents are shown in clear text. In both parts of the figure, one row represents a directory entry for one short file name. In the lower part we can see that all the four file names are stored in the first part of the entries. The yellow marked parts of the upper and lower parts contains the start cluster for the various files. The start clusters are represented by the four bytes 20, 21, 26 and 27. In this example, we can see that the four files are stored in cluster 3, 4, 5 and 6, respectively.

By looking at the figure, we also discover that the '.' (dot) in the file names are not stored. This is because short file names are always represented in the same manner, and thus the file system knows that the contents of bytes 8, 9 and 10 represent the file extension. Finally, we can see that the two directory entries following the "delta.txt" file contains all 0's and hence are available for storage of other files.

Long file names are files where all letters of the file name cannot be stored in one short directory. This means that all files with 12 characters or more, need to be represented in a different way than short file names [5]. Each file with long file name has one short directory entry in addition to a multiple of long directory-specific entries. The reason for this is that the long file name needs to be backward compatible. According to the MS FAT specification [5], "only short directory entries are visible to previous versions of MS-DOS/Windows". Furthermore, the specification also states that "without a short entry to accompany it, a long directory entry would be completely invisible on previous versions of MS-DOS/Windows" [5].

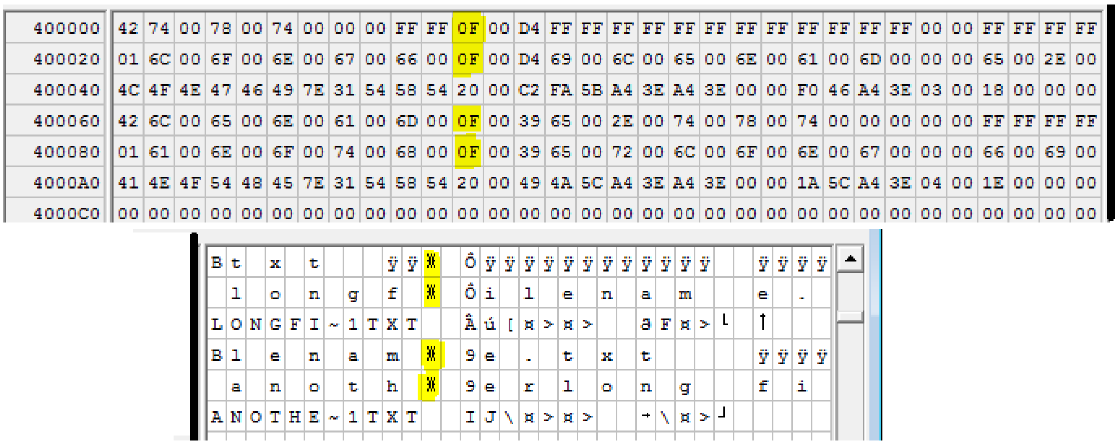
The short directory entry for long file names has exactly the same entry structure as short file names. The difference between short directory entries for short and long file names, is the way the file names are represented. In the short directory entry for long file names, the name is a compressed version (first six letters) of the file name, followed by '~', a number and the file extension [5]. In figure 5, the yellow marked cells show the compressed representations of two files with long file names.



```
Bt x t   ÿÿ☼ Ôÿÿÿÿÿÿÿÿÿÿÿÿ  ÿÿÿÿ
 l o n g f ☼ Ôi l e n a m   e . 
LONGFI~1TXT  Âú[¤>¤>  ðF¤>└ ↑   
Bl e n a m ☼ 9e . t x t     ÿÿÿÿ
 a n o t h ☼ 9e r l o n g   f i 
ANOTHE~1TXT  IJ\¤>¤>  →\¤>┘     
```

Figure 4: Compressed version of two long file names

The long directory entries for long file names each contain various information. Some blocks in the entry contain parts of the file name, one block contains which number it is in the sequence of long directory entries which represent the file, a third is a check sum, and finally there is a value indicating that the directory entry is representing a long file name. This value is stored in byte number 11 of the entry, and is 0x0F if it is a long file name [5]. In figure 6, we can see those values marked in yellow for two sample files.

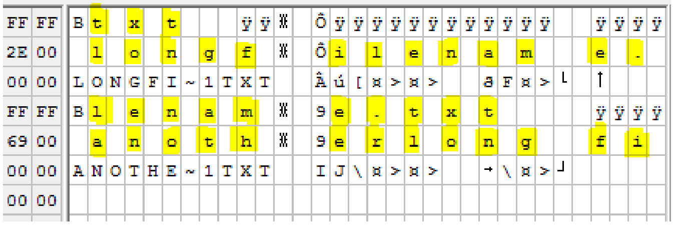


```
400000  42 74 00 78 00 74 00 00 00 FF FF 0F 00 D4 FF FF FF FF FF FF FF FF FF FF FF FF 00 00 FF FF FF FF
400020  01 6C 00 6F 00 6E 00 67 00 66 00 0F 00 D4 69 00 6C 00 65 00 6E 00 61 00 6D 00 00 00 65 00 2E 00
400040  4C 4F 4E 47 46 49 7E 31 54 58 54 20 00 C2 FA 5B A4 3E A4 3E 00 00 F0 46 A4 3E 03 00 18 00 00 00
400060  42 6C 00 65 00 6E 00 61 00 6D 00 0F 00 39 65 00 2E 00 74 00 78 00 74 00 00 00 00 00 FF FF FF FF
400080  01 61 00 6E 00 6F 00 74 00 68 00 0F 00 39 65 00 72 00 6C 00 6F 00 6E 00 67 00 00 00 66 00 69 00
4000A0  41 4E 4F 54 48 45 7E 31 54 58 54 20 00 49 4A 5C A4 3E A4 3E 00 00 1A 5C A4 3E 04 00 1E 00 00 00
4000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
Bt x t   ÿÿ☼ Ôÿÿÿÿÿÿÿÿÿÿÿÿ  ÿÿÿÿ
 l o n g f ☼ Ôi l e n a m   e . 
LONGFI~1TXT  Âú[¤>¤>  ðF¤>└ ↑   
Bl e n a m ☼ 9e . t x t     ÿÿÿÿ
 a n o t h ☼ 9e r l o n g   f i 
ANOTHE~1TXT  IJ\¤>¤>  →\¤>┘     
```

Figure 5: Values indicating long file names

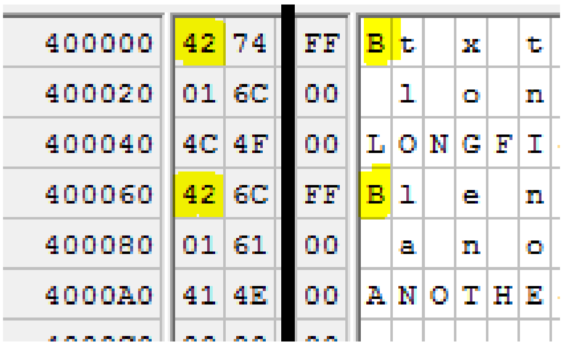
The general structure of directory entries for a long file name is: First a number of long entries and finally one short directory entry. The first long directory entry contains the last part of the file name, followed by an long directory entry with the second last part of the file name, and so on. Each long directory entry contains up to 13 characters of the file name, separated in three different parts of the directory entry. The bytes used to store the file name are 1, 3, 5, 7, 9, 14, 16, 18, 20, 22, 24, 28 and 30, respectively [5]. In figure 7 below, we see how the file names for two sample files with long file names are stored. Here, we can also see that dot is stored for long file names. This is because the representation for the long file names is not always the same, making the file system unable to know where the dot needs to be placed.



```
FF FF  B t   x   t       ÿ ÿ ※   Ô ÿ ÿ ÿ ÿ ÿ ÿ ÿ ÿ ÿ ÿ ÿ ÿ     ÿ ÿ ÿ ÿ
2E 00    l   o   n   g   f   ※   Ô i   l   e   n   a   m       e   .
00 00  L O N G F I ~ 1 T X T     Â ú [ ¤ > ¤ >     ә F ¤ > └   ↑
FF FF  B l   e   n   a   m   ※   9 e   .   t   x   t           ÿ ÿ ÿ ÿ
69 00    a   n   o   t   h   ※   9 e   r   l   o   n   g       f   i
00 00  A N O T H E ~ 1 T X T     I J \ ¤ > ¤ >     → \ ¤ > ┘
00 00
```

Figure 6: Representation of long file names

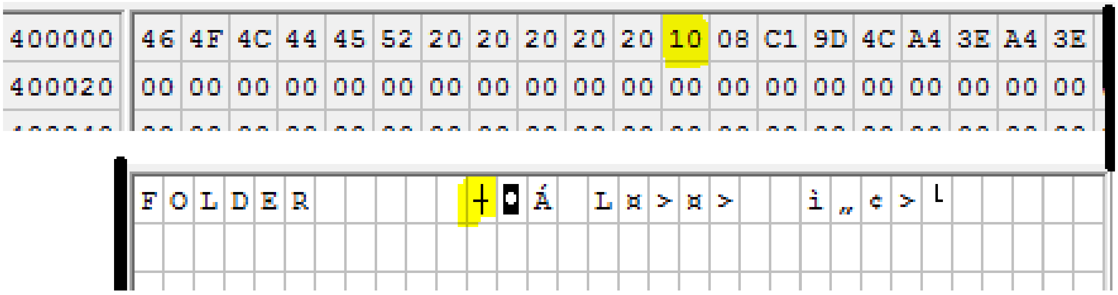
The amount of long entries varies, depending on the length of the file name. In general, a file with a file name consisting of X letters (including the dot and extension) has X/13 (rounded up to the closest integer) long directory entries. Otherwise, we can find the number of long directory entries with the following formula: (the first value in the first long directory entry - 64). In the sample files shown in the figure 7, we see that the first directory entry for both the file names starts with a value of 0x42. This is equivalent to 66 in decimal, and thus we can find the number of long directory entries with (66-64 = 2).



```
400000  42 74  FF  B t   x   t
400020  01 6C  00    l   o   n
400040  4C 4F  00  L O N G F I
400060  42 6C  FF  B l   e   n
400080  01 61  00    a   n   o
4000A0  41 4E  00  A N O T H E
```

Figure 7: First entries in long file names

Regarding folders, the directory entry has the identical basic structure as that of files [5]. The representation of folders is also near the same. The two main differences are that folders do not have an extension and hence do need the dot, and that the directory entry needs to indicate that it is representing a folder. This is done by adding the value 0x10 in the byte 11 of the directory entry for short folder names [5]. For folders with long names, this value is added in byte 11 of its short directory entry. Figure 9 shows a representation of a folder, where the values indicating that a folder is stored, are marked in yellow.

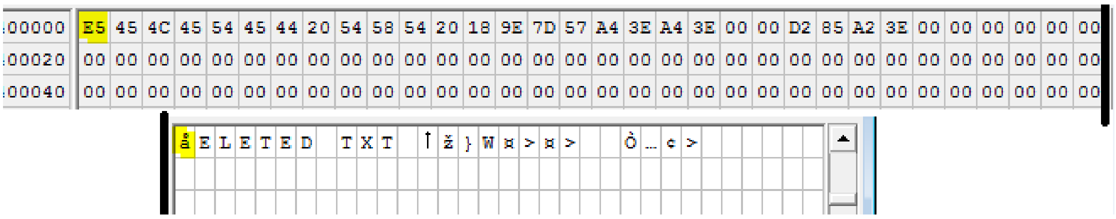


```
400000  46 4F 4C 44 45 52 20 20 20 20 20 10 08 C1 9D 4C A4 3E A4 3E
400020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

F O L D E R         ┼ ■ Á   L ¤ > ¤ >   i „ ¢ > └
```

Figure 8: Representation of a directory

One essential question for both long and short file names is the following: What happens with deleted files after deletion? The answer is that the deleted file still exists on the memory stick. Then, two other questions arise. If deleted files still exists on the system, why are they not visible and what happens with the available space on the memory stick when files are deleted? The answer to those questions is that when a file is deleted, it will be marked with one special character in the first byte of its directory entry. This character is E5 in hexadecimal, which is 229 in decimal, and when the file system finds this value, it knows that the file has been deleted and that the cluster(s) used to store the deleted file may be used for storage of other file(s). This way, deleted files will not cause problems when it comes to available space, and it will also lead to deleted files and their contents not showing up when the memory stick is accessed.

In figure 10, we can see the representation of a deleted file, “deleted.txt”. The yellow marked cells show that the first cell in the directory entry has been replaced by the value 0x E5.



```
:00000  E5 45 4C 45 54 45 44 20 54 58 54 20 18 9E 7D 57 A4 3E A4 3E 00 00 D2 85 A2 3E 00 00 00 00 00 00
:00020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
:00040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

å E L E T E D   T X T   ↑ ž } W ¤ > ¤ >   ò … ¢ >
```

Figure 9: Deleted file

4 Development and testing

In order to complete our assignment, we had to implement the following functionality:

- Write sectors to the memory stick.
- Hide files on the memory stick.
- Recover the same files from the memory stick.
- Securing the hidden files by preventing new files to overwrite the hidden ones.
- Be compatible with a GUI developed by us.

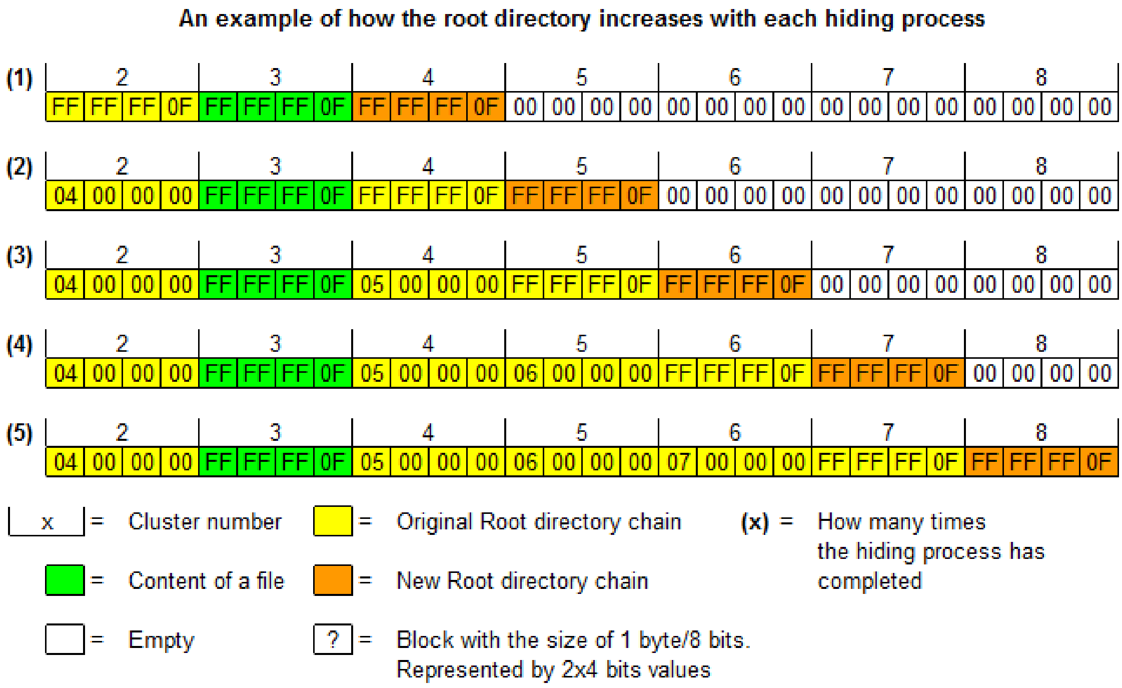
Prior to the official start of the bachelor thesis, the employer handed us a proof of concept code, which can be seen in appendix H, on how to read one sector from an USB (Universal Serial Bus) memory stick while also creating a binary file with the contents of the memory stick. We were recommended to familiarize us with the code and how to use a software called XVI32 [**?**] to inspect the data in the binary file.

Analysing some binary files against the information in the FAT (File Allocation Table) system specification [5] was necessary to get started with development of the software. This information taught us for instance in which cluster the root cluster starts, the cluster size, the sector size and how many sectors the FAT region contains [5].

Since the proof of concept source code did not contain any write functions, we needed to develop or own in order to continue with the software. We created a function called DriveWriteSector() (DWS()), which partially worked, but had a limitation on which regions data could be written to. DWS() could not write bytes into the FAT or data region and we struggled trying to figure out how to solve the problem. The solution to the problem was that DWS() needed an additional I/O (Input/Output) control function to enable writing to any given byte on the memory stick. The I/O control function in question needed to be called with the two parameters FSCTL_DISMOUNT_VOLUME [12] and FSCTL_LOCK_VOLUME [13] to lock the memory stick, which means that no other process could use it while the software was running.

The development of functionality to hide files started when we were able to write data to any region we wanted. One easy solution to hide files, was by creating a new root directory in a free cluster. The free cluster did not contain any file entries and therefore Windows Explorer (WE) displayed the memory stick as empty. By editing the root cluster value in the BPB and adding the value [FF] [FF] [FF] [0F] to the new root cluster address in the FAT region, we had successfully manipulated the FAT system in order to hide files. One side effect was that the memory stick showed the space allocated by the file, even though the file name was hidden from WE. This problem will be discussed later in this chapter.

In order to recover the hidden file(s), we considered copying the directory entries in the new root directory back to the original root directory, and then set the root directory entry in the boot sector to its original value. However, after consulting with the employer's liaison supervisor, we agreed upon that copying directory entries would be too time consuming and that we rather could link the root directories into one big cluster chain. This method is more space consuming, since it will allocate an extra cluster for each initiated hiding process. However, with normal use-frequency of the software, we did not considered the additional space used to make a significant impact.



An example of how the root directory increases with each hiding process

	2				3				4				5				6				7				8			
(1)	FF	FF	FF	0F	FF	FF	FF	0F	FF	FF	FF	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
(2)	04	00	00	00	FF	FF	FF	0F	FF	FF	FF	0F	FF	FF	FF	0F	00	00	00	00	00	00	00	00	00	00	00	00
(3)	04	00	00	00	FF	FF	FF	0F	05	00	00	00	FF	FF	FF	0F	FF	FF	FF	0F	00	00	00	00	00	00	00	00
(4)	04	00	00	00	FF	FF	FF	0F	05	00	00	00	06	00	00	00	FF	FF	FF	0F	FF	FF	FF	0F	00	00	00	00
(5)	04	00	00	00	FF	FF	FF	0F	05	00	00	00	06	00	00	00	07	00	00	00	FF	FF	FF	0F	FF	FF	FF	0F

x = Cluster number

🟨 = Original Root directory chain

(x) = How many times the hiding process has completed

🟩 = Content of a file

🟧 = New Root directory chain

□ = Empty

? = Block with the size of 1 byte/8 bits. Represented by 2x4 bits values

Figure 10: How the root directory increases in size when hiding files

Figure 11 shows how the space allocation for the root directory increases by each hiding process. Each row in the figure displays the same FAT region. Each row shows the FAT after a hiding process has finished. When the recovering process runs, the original and the new root directory get linked together and appear as one root cluster. When the hiding process runs one more time, a new free cluster get allocated as the new root directory and so it goes on.

Hiding and recovering of files did not require a complex algorithm, as opposed to a function which prevents the occurrence of duplicated file names when recovering files, and a function which reduces the space shown as allocated in WE.

A further non-trivial problem that needed to be addressed, was to eliminate the risk of overwriting files. The problem was that if new files were added to the memory stick after a hiding process, the new files might overwrite the hidden files.

To eliminate the problem with overwriting files, we decided to implement a safe mode in the software. If the software is executed in safe mode, the cluster containing the contents of stored files will be marked as bad. Marking clusters as bad means that no other files can be stored in the same cluster, and hence it prevents the risk of overwriting files.

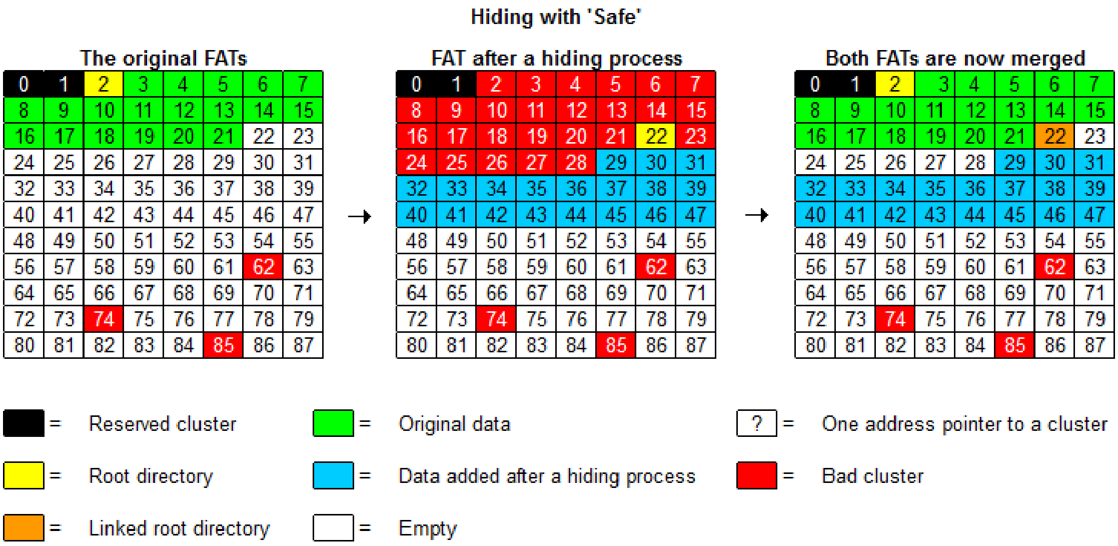


Figure 11: Hiding files with safe mode

As shown in figure 11, the hidden data is safe from being overwritten due to the clusters are marked as bad.In the box on the left side, the cluster numbers from three to 21 are the files the software is going to hidden. The box in the middle shows that the content of these clusters are safe because the new files are being stored from cluster 29 and upwards. The box on the right side shows that no conflicts between the original and new files takes place.

In addition to the safe mode, we also decided to develop a light mode. This mode will decrease the space shown as allocated, but does not eliminate the problem with overwriting of files. The space allocation will only decrease if the files already stored on the memory stick are larger than the FAT copy which is also stored on the memory stick.

The risk of overwriting hidden files when using light mode is shown in figure 12. The box in the middle shows that new files are stored in the same clusters which contained the hidden files, while the box on the right shows which clusters are overwritten. Because of this, the owner of the memory stick will now only be able to access the files added after the hiding process.

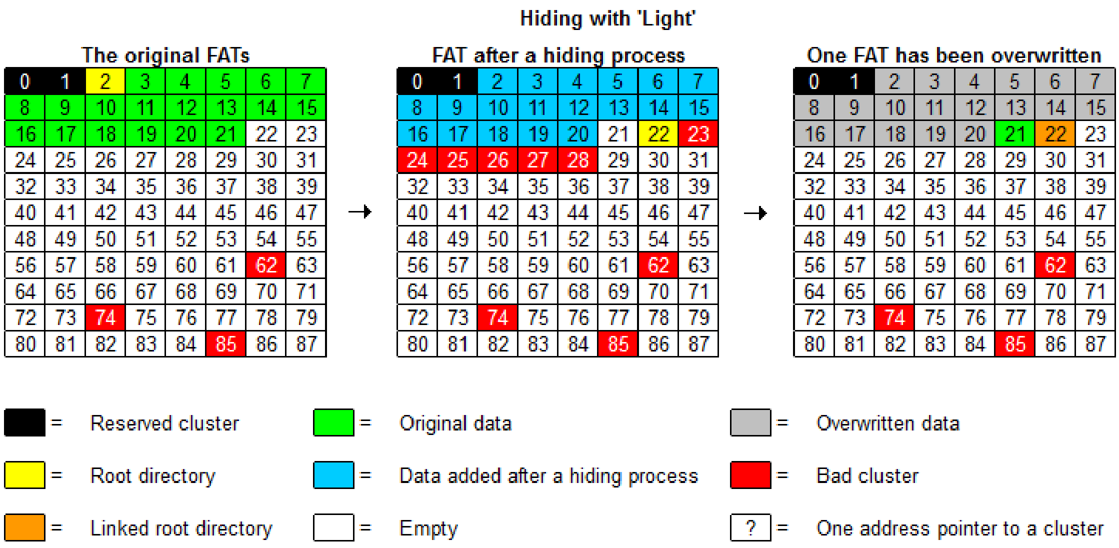

Hiding with 'Light'
The original FATs
FAT after a hiding process
One FAT has been overwritten
⬛ = Reserved cluster
🟨 = Root directory
🟧 = Linked root directory
🟩 = Original data
🟦 = Data added after a hiding process
⬜ = Empty
= Overwritten data
🟥 = Bad cluster
? = One address pointer to a cluster

Figure 12: Hiding files with light mode

Both figure 11 and figure 12 shows that the three bad clusters 62, 74 and 85 which are not manipulated as bad by our software, regardless of the users choice, will not be changed.
While developing the software, we differentiated the versions from each other by numbering them 'x.x.x'. The first 'x' displays the release candidate number. This number will only change when major changes have been made to source code, for instance when a major overhaul have been done to the software and the GUI. The second 'x' displays any huge difference between two source codes, for instance by developing a new method of hiding files. If only small bug fixes were added to the source code, the third x would increase the count by one, e.g. '0.2.1' to '0.2.2'.

During the development period, we operated with three different main versions. The differences between them are show below:

- 0.1.x
  - Reads/writes sector by sector.
  - Reads/writes several times during the hiding process to the memory stick.
  - Did not contain the entire recovery process.
- 0.2.x
  - Still Reads/writes sector by sector.
  - Reads/writes to the FAT only once during a process which limits the need for I/O-calls.
  - Contained the entire recovery process.
- 0.3.x

- Reads/writes a cluster or the whole FAT region at once, thus limiting the need for I/O-calls.

The most important difference between the version ’0.1.x’ and version ’0.2.x’ was that the software did not read the FAT region into a buffer and passed the same buffer to functions who relied on that information. Instead of having to read the FAT region from the memory stick every time the software needed information, or to write the changes back to the memory stick at the end of every function which utilized the FAT, we decided to store the FAT region into memory. By doing so, we limited the need for reading and writing from/to the physical drive to one time. Basically, we allocated a buffer large enough to contain the two FATs, then the FATs are read into the buffer. Finally, the buffer is passed as parameter to functions like RemoveTrace(), RecoverTrace(), HideAll() etc.

4.1 The foundation of the software

The first thing the software does is to gather important data about the memory stick the user wants to operate on.

Table 1 shows the data we extracted from the BPB [5].

Table 1: Data found in the BPB

Name in MS FAT spesification	**Name in code**	**Description**	**Stored in blocks**
BPB_BytsPerSec	BytesPerSec	Total bytes in each sector	11 and 12
BPB_SecPerClus	SecPerClus	Total sectors in a cluster	13
BPB_RsvdSecCnt	ResSec	The size of reserved area in sectors	14 and 15
BPB_FATSz32	SecPerFat	Total sectors in a FAT	36, 37, 38 and 39
BPB_RootClus	RootClus	Contains the beginning of the root directory	44, 45, 46 and 47

Other variables we used, but had to compute by the information already extracted, was:

- StartDataregion: The beginning of the data region. We computed this variable by “ResSec + (SecPerFat * 2)”
- SecForBothFat: Used to create different kinds of buffers containing two FATs or for scanning through FATs. The variable is computed by “SecPerFat * 2”
- TotBytesInFat: Same as above. Computed by “SecForBothFat*BytesPerSec”.

After the software has extracted important information from the BPB, the user will be presented with a choice to either hide or recover files. This was solved by creating a ’case’ operation. We could have easily avoided the ’case’ and created a simple ’if’ solution instead, but it is easier to add new functionality or options later, when using the ’case’ option. This is because we considered lots of ’if’ argument to be confusing compared to using cases. When we developed the software, it was a possibility that it may become an open-source project, which was another reason for writing a legible source code.

4.2 Development of the RemoveTrace() function

Hiding files and folders was easy and did not take a lot of effort to develop. However, we encountered a problem when we discovered that the memory stick was displaying the space allocated for hidden files. If anything will raise suspicion, it will be when an attacker is able to see that parts of the memory stick are allocated, even though no files or folders are visible.

We had to hide the tracks of allocated space and decided to create a copy of the FAT region to store it on the memory stick. The reason for this is because we noticed the computer did not use a value in the reserved region to compute how much space was available. However, we noticed that the amount of allocated space changed when we manipulated the FAT region. We figured out that if we set every cluster except the root directory, the first two clusters in the FAT and the bad clusters to zero, then we could see a significant decrease in allocated space. The difference in space allocation depended on how much space was originally allocated. We agreed to develop a solution which creates a copy of both FATs and stores it in the data region on the memory stick. After the copy process is done, the software will mark clusters that contains the files the user has hidden. Those clusters will be marked as either bad or free, depending on the user's choice.

Figure 13 describes how the memory stick looks after the hiding process has finished with safe mode.

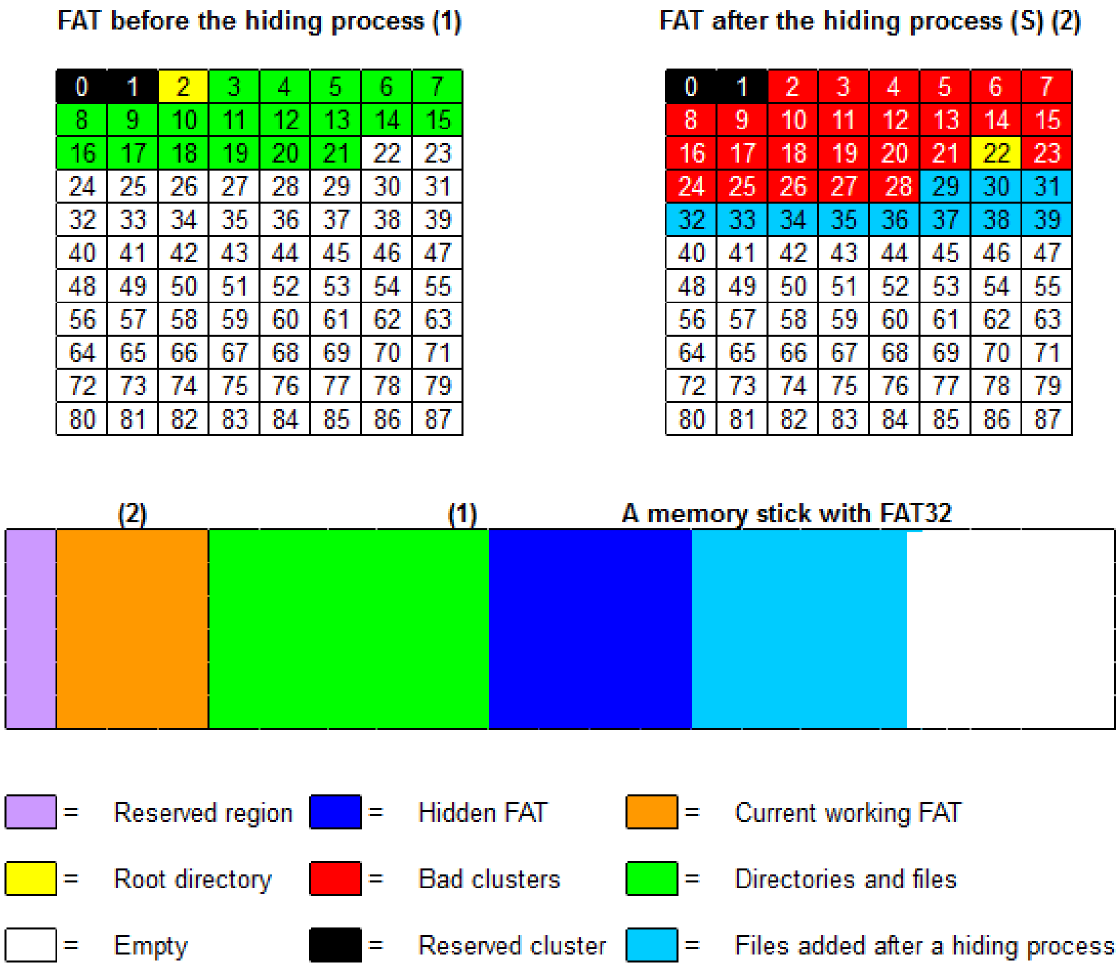


Figure 13: How the memory stick looks after the hiding process has finished.

The long box in the middle displays how the memory stick looks like after the hiding process is complete. The FAT on the upper left side is the original FAT and is marked with ’1’. The cluster number two until 21 is shown as the green box in the memory stick’s data region, which contains the files and directories before the hiding process begins. The dark blue box to the right of the green box contains the copy of the original FATs which are stored in the data region, after a hiding process. That FAT is a copy of the original FAT and will no longer be in use. The FAT on the upper right side is displaying the currently working FAT and is marked ’2’. The orange box describes where the currently working FAT is stored on the memory stick, i.e. in the FAT region. The light blue box on the memory stick displays files and folders added after the hiding process was completed.

The RemoveTrace() function consists of two main parts. The reason for separating the two main parts of RemoveTrace() was to make a less complicated code by doing different operations separate from each other. It was also easier to discover coding errors by checking which operation that failed. The first part creates a large data buffer will contain the FAT copy and predefined sectors [source code 1], while the second part finds free clusters and writes one cluster chunk of data to the data region of the memory stick.

Figure 14 illustrates how RemoveTrace() prepares a cluster with data from the FAT copy so that when the FAT copy is written to disk, the RemoveTrace() will take one cluster size of data from the ’buffer’ and then write it to a given free cluster on the memory stick in the data region. For instance, will sector 0 to 6 be written to one cluster in the memory stick, while sector 7 to 13 will be written to the next free cluster, and so on. The number inside the blocks in ’Buffer’ refers to the sector number in ’FATBuffer’. The predefined sector is explained in details below.

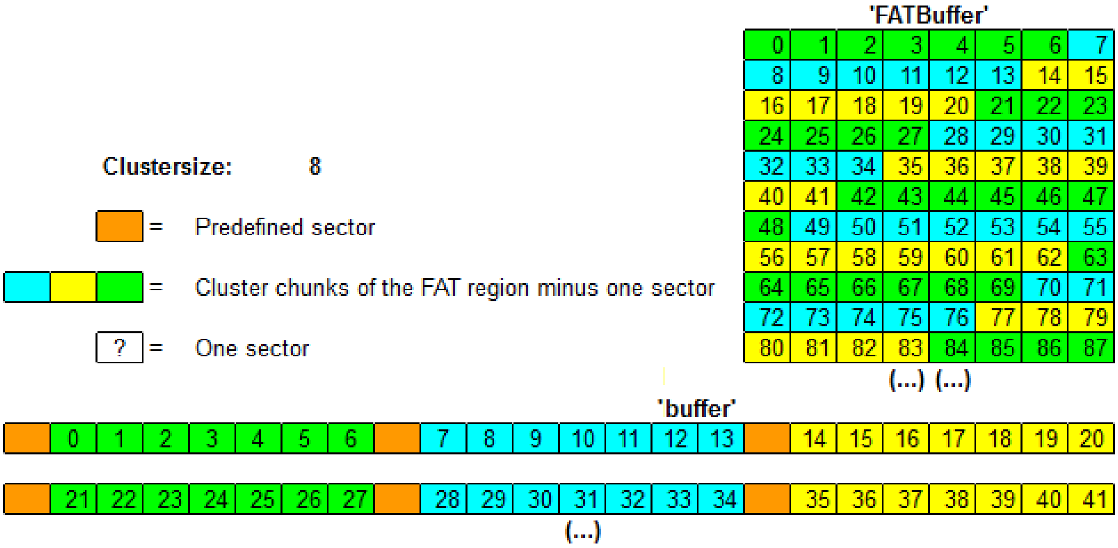


Figure 14: Predefined sectors

A side effect of hiding the FATs with bad cluster is that there could be legitimate bad clusters around on the memory stick before the hiding process started. In order to differentiate between legitimate and manipulated bad cluster, we created a predefined sector which would be used by the software to recognise manipulated clusters.

We developed the algorithm for predefined sector by first creating the predefined sector, then reading X amount sectors from the FAT region depending on the cluster size. The one of the reason for why we created a predefined sector, besides recognising manipulated clusters, was that we needed to save different kinds of data for the recovering process. One of these was the original root cluster. The source code below shows that the predefined sector had four ’g’ in order to recognise a manipulated cluster. The original root cluster value is also stored for recovery purposes.

```
for(i=0; i< SecForBothFat; i++)                    // Creation of the FAT copy
   {
     if(i % ExtraSector == 0)
       {
         buffer[0+temp]= ’g’; buffer[1+temp]= ’g’;         // Predefined sector
         buffer[2+temp]= ’g’; buffer[3+temp]= ’g’;
         buffer[4+temp]= byte1; buffer[5+temp]= byte2;   // Original root cluster
         buffer[6+temp]= byte3; buffer[7+temp]=byte4;
         for(j=8; j< BytesPerSec; j++)
             buffer[j+temp]= ’D’;
           temp+=BytesPerSec; test++;
           for(j=0; j < BytesPerSec; j++)             // Copies the FAT by
                 buffer[j+temp]= FATBuffer[j+temp2];          // cluster -1 sector.
       }
       else
       {
          for(j=0; j < BytesPerSec; j++)               // Copies the FAT
             buffer[j+temp]= FATBuffer[j+temp2];
       }
       temp+=BytesPerSec, temp2+=BytesPerSec;          // Gets ready for a new sector
   }
```

After the ’buffer’ is filled with FAT and predefined sectors, the RemoveTrace() needs to know where it can store a part of the FAT copy. First the it checks for free clusters in the FAT and then write one cluster of data of the FAT copy, sector by sector, to the data region of the memory stick. After the whole FAT region has been copied to the memory stick, the function WipeFatregion() sets every cluster pointer in the working FATs to zero, with the exception of bad clusters, root directory and the first 8 bytes in the FAT.

WipeFatregion() was a huge timesink. In version ’0.1.x’ it began with reading the FAT region into memory, cleaning it and then write it back, sector by sector.

As mentioned above, both reading and writing one sector at the time caused a lot of unnecessary I/O-calls which consumes a huge amount of time. In version ’0.2.1’ however, the FAT region was saved in a buffer when the software began its loading process and then it got passed to the functions who needed to operate with data from the FAT region. This decreased the time spent on I/O-calls significantly since the software then operated with data stored in the memory, instead of dealing with a physical memory stick. It also enabled the software to only read and write to the FAT region once while the hiding process was in progress.

Version ’0.3.1’ is improved even further by eliminating the need to read or write only one sector at the time. We allocated two buffers which were used to contain the data that was going to be read from, or written to, the memory stick. One of the buffers has a size of the FAT region and the other one has a size of a cluster. When the software is ready to write the FAT copy to the memory stick, it copies a cluster worth of data into the cluster-sized buffer. Then the RemoveTrace() writes the data to the memory stick in one single operation, unlike version ’0.2.x’, where the software had to call the write functions as many times as it was sectors in a cluster. By writing bigger pieces of information at once, the need for I/O-calls will be reduced by a significant amount. For instance, on a 4 GB memory stick with 1024 bytes cluster size, the time difference between version '0.2.x' and version '0.3.x' for hiding files is 706 seconds. That is a significant decrease in time used. However, as seen in table 2, the time difference decreases as the cluster size increases.

	Hiding 4 GB		**Recovering 4 GB**	
Cluster Size	0.2.x	0.3.x	0.2.x	0.3.x
512 kb	N/A	N/A	N/A	N/A
1024 kb	897	191	258	52
2048 kb	355	43	114	13
4096 kb	170	11	58	5
8192 kb	84	4	29	2
16384 kb	41	2	14	1
32768 kb	21	1	8	0
65536 kb	N/A	N/A	N/A	N/A

Table 2: Time used

Problems during the development of RemoveTrace() Calculating the exact size of the buffer which should contain the entire FAT region and predefined sectors was hard and led to many buffer overflow errors. Trying to figure out the exact size for any cluster sizes was had hard and took almost the same time as developing the algorithm for RemoveTrace() itself.

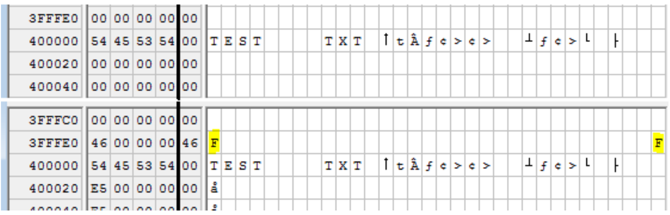
4.3 Development of the RecoverTrace() function

Before we started to develop the RecoverTrace() function, we realised that we could read the entire FAT into memory which we could access from any function and thus decrease the amount of time spent on either hiding or recovering files. When we developed the new and improved RemoveTrace() we decided to upgrade the version number to '0.2.1'.

RecoverTrace() was an easier function to develop than RemoveTrace() due to the that RecoverTrace() only had to search for bad clusters and read the content. The RemoveTrace(), on the other hand, needed to do a lot of calculations in order to make sure everything worked correct. The function scans through the currently working FAT and checks whether the bad clusters contain parts of the original FAT. If a manipulated bad cluster is found, the function reads the rest of cluster into a temporary buffer until both FATs have been recovered from the memory stick. When both FATs have been recovered, the RecoverTrace() will overwrite the currently working FAT with the one it recovered.

Problems during the development of RecoverTrace() One problem we encountered during the development of RecoverTrace() was that it did not read the entire hidden FAT. Usually the problem occurred with FAT number two because the variables we used to determine where, and how much data, the function needed to read from the memory stick were wrong. The variables were not calculated correctly and hence displayed wrong values after passing a certain value.

One way to make sure that the entire FAT was recovered from the data region on the memory stick was to mark the last byte in the working FAT with a letter, for instance the letter ’F’. We marked the last byte in the with an ’F’ before the hiding process started, and checked that the WipeFatRegion() removed the ’F’ before we started the recovering process. Now the FAT copy stored in the data region would contain a ’F’ in the last byte of the FAT, but the current working FAT would not have it. After the recovering process had copied the original FAT back to the FAT region, we checked whether the last byte in the FAT region contained an ’F’. If it did and the following sector still contained the original root cluster, then we knew that the RecoverTrace() managed to read back the exact amount of data and stopped reading when it reached the end of the FAT.



```
3FFFE0  00 00 00 00 00
400000  54 45 53 54 00  TEST    TXT ↑tÂƒ¢>¢>  ⊥ƒ¢>└ ├
400020  00 00 00 00 00
400040  00 00 00 00 00

3FFFC0  00 00 00 00 00
3FFFE0  46 00 00 00 46  F                              F
400000  54 45 53 54 00  TEST    TXT ↑tÂƒ¢>¢>  ⊥ƒ¢>└ ├
400020  E5 00 00 00 00  å
```

Figure 15: The letter ’F’ occurred after the hiding process had finished.

The figure 15 shows that the letter ’F’ occurred after the recovery process had finished, which concludes that both FATs have been placed in their original position. The box on the top displays the end of the working FAT after the RemoveTrace() has finished, while the box below shows how the end of the same FAT looks like after RecoverTrace has been finished.

Manually setting the last byte in the FAT as ’F’ did not cause any conflicts because there was no cluster which linked to the last cluster on the memory stick. This is because we can store any information we want in the FAT as long as it does not interfere with the cluster chains already there.

Another problem we encountered during the development process of RecoverTrace(), was the use of pointers. To copy the FAT into a temporary buffer we wrote TempBuffer = FATBuffer. What happened, was that TempBuffer pointed to FatBuffer’s address instead of being an individual copy. The big disadvantage with this was that changes done to TempBuffer also effected FATBuffer. In order to create a real copy instead of having two variables pointing to the same address, we used a for-loop to make an exact byte for byte copy.

4.4 Development of the GUI

The employer wanted a user friendly software. We decided to create a graphical user interface (GUI) since it would be much more user friendly than a console interface. The GUI was written in C#, a managed programming language made by Microsoft. The reason for choosing C# when creating the GUI was that it would be simpler than using just the Windows API in C or MFC in

C++. Drawing the GUI in C# is trivial compared to developing it in C or C++, where it would require a lot of time spent just creating buttons and text.

Programming the GUI in another programming language meant that we had to make our C code into a DLL that could be called from the C# GUI code. This was not an easy process. We had to find out how C# calls DLLs, and changing our C code so we could use it as a DLL. By using C# we could also more easily extract information about all plugged-in memory sticks, only allowing to hide files on those with a FAT32 file system.

We had some problems in the beginning with getting the DLL to work properly, but all things considered, it was probably a better choice than having to create the GUI from scratch with pure C. In MS VS C# we could change the appearance of the GUI easily without having to write any code.

5 Development results

In the following sections, we will describe the main functionality of the most important functions used by the software. The functions will be presented in the order which they are employed, beginning with the hiding process and ending with the recovering of hidden files. All functions will be presented with some examples and figures, in order to be easier understood by the reader. The main functions presented are HideAll(), RemoveTrace() and RecoverTrace(), as well as different functions for recovering files and handling problems with duplicate file names.

5.1 How HideAll() works

This function serves as the main function for hiding of data. The method for hiding files is based on creating a new root directory, making the file system driver think that the old one does not exist, and hence showing no files.

The function first calls FindNextFreeCluster() (FNFC()) to obtain a free cluster where the new root directory can be stored. FNFC() scans through every cluster entry in the FAT (File Allocation Table) and checks whether any of them have a value of zero. If that is true, the function returns the cluster number of the first free cluster found. If it is not the case, it will return 0.

Then, the function ClearCluster() will erase any garbage the found cluster might contain, by setting every byte inside the cluster to zero. The reason why we want to remove possible garbage data is that it might become visible when the memory stick is inspected through WE (Windows Explorer). Any visible garbage will lead to the attacker becoming suspicious and might want to put in some extra effort in order to extract information from the memory stick.

5.2 How RemoveTrace() works

The first thing RemoveTrace() does is to create a large buffer called ’buffer’. This buffer is going to contain the FATs and predefined sectors, as seen in figure 14, that will be stored in the data region on the memory stick. Next, the RemoveTrace() creates a predefined sector which the software will use to recognise manipulated clusters. Then, the function will copy the amount of sectors in a cluster, minus the predefined sector, into ’buffer’. The RemoveTrace() will continue to copy the FATs into the buffer until there are no more sectors left to copy.

When ’buffer’ has all the data needed, RemoveTrace() calls FNFC() to find a free cluster, copies a cluster of data from ’buffer’ to the character array ’ClusBuff’, which is used when reading or writing to the memory stick is necessary. The RemoveTrace() loops until every cluster in ’buffer’ has been written to the data region.

Table 3: Space allocated

	2 GB	**4 GB**	**8 GB**	**16 GB**	**32 GB**
Cluster Size (bytes)	**Space allocated (MB)**				
1 024	**29,56**	**59,29**	N/A	N/A	N/A
2 048	**9,89**	**19,85**	**39,71**	N/A	N/A
4 096	4,25	8,53	**17,05**	34,05	N/A
8 192	1,99	**3,99**	**7,97**	**15,91**	32,22
16 384	0,98	1,95	**3,87**	7,72	**15,60**
32 768	N/A	1,00	1,93	3,84	7,72
65 536	N/A	N/A	1,06	2,06	3,88

After the entire FAT region has been copied to the data region, RemoveTrace() calls the WipeFatregion() function which edits the working FAT based on the user's choice. The user has the option to select a light mode through the command line by having the letter 'e' as a parameter when he starts the program. The GUI will then in “advanced” mode. The difference between the safe mode and the light mode is how much space is displayed as allocated and whether adding new files after a hiding process possibly will overwrite the old files or not.

The light mode option in the WipeFatregion() checks whether any cluster address, except the first eight bytes in the working FAT is different from [F7][FF][FF][0F] and hence not marked as bad. If the cluster address is not marked as bad, then the WipeFatregion() marks the cluster as free. The result is a significant decrease in displayed allocation space, depending on how much data the memory stick originally contained. The table below shows the space allocated after files have been hidden, for various cluster and memory stick sizes. The numbers in bold indicates when an attacker can see that the space is allocated from WE, by just looking at the icon representing the memory stick.

The only difference between light and safe mode is that the safe mode mark the clusters, which the light mode marks as free, as bad. Figure 16 displays the difference between safe and light mode regarding the amount of bad clusters. The box in the middle shows that the safe method marks the used clusters, ranging from 2 to 21, as bad, while the box on the right shows that the light mode marks them as free.

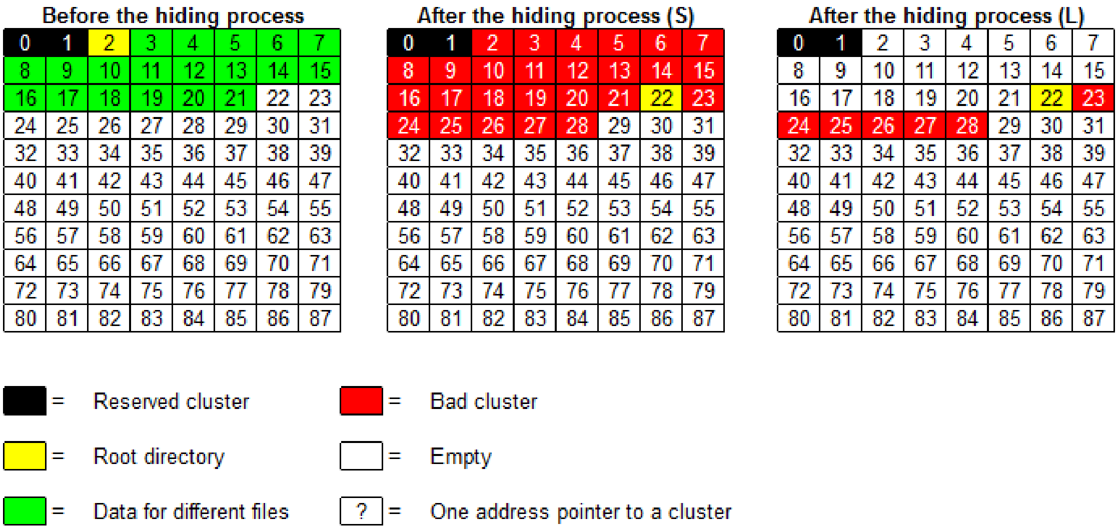

Before the hiding process
After the hiding process (S)
After the hiding process (L)
= Reserved cluster
= Root directory
= Data for different files
= Bad cluster
= Empty
? = One address pointer to a cluster

Figure 16: The difference between safe and light mode with the amount of bad clusters.

5.3 How RecoverTrace() works

The first thing RecoverTrace() does is to create a temporary buffer, called 'Buffer', which is used to store the entire FAT extracted from the 'FATBuffer'. We created 'TempBuf' to store the FAT which was read from the data region on the memory stick. Instead of checking 'FATBuffer' for bad clusters, we checked 'Buffer' and stored the FAT in 'TempBuf' instead of 'FATBuffer'. The reason for this was that we did not want to change the values in 'FATBuffer' untill we had to. If we operated on the 'FATBuffer' directly, we would end up checking the wrong FAT as the FAT from the data region would overwriting the FAT we used to check for bad clusters in the 'FATBuffer'.

We did actually encounter such an error by writing 'Buffer = FATBuffer', where 'Buffer' is the buffer we used for reference and 'FATBuffer' is the FATcopy of the currently working FAT. The issue is explained in detail in section 4.4.

The RecoverTrace()runs through the 'Buffer' and checks for any manipulated clusters. For each manipulated cluster it finds, it sets two bytes in the array 'CorruptCheck' to one, one byte for each FAT. If the cluster was not manipulated, it will be set as zero. The reason why RecoverTrace() needs 'CorruptCheck' when it merges the original FAT with the new one, is because it needs to know which clusters that have been marked as bad. The merging process will be explained later in this chapter.

When RecoverTrace() encounters a bad cluster, it checks whether the first sector is predefined by the RemoveTrace() function. If the cluster is used to hide the original FAT, then the values, for the given cluster, in CorruptCheck is set to 1. The source code below shows how the RecoverTrace scans through the FAT, finds a bad manipulated cluster and then marks the 'CorruptCheck'.

```
void RecoverTrace(Drive* dri, SectorBuffer* buf, unsigned char* FATBuffer) {
(omitted code)
for(i=0; i< (SecPerFat*BytesPerSec)-4; i+=4)            //Scans throug FAT
{
 CorruptCheck[i]='0';
 CorruptCheck[i+(TotBytesInFat/2)]='0';               // Resets the array
 if(Buffer[i]== 247 && Buffer[i+1]== 255         // Checks for bad cluster
      && Buffer[i+2]== 255 && Buffer[i+3]== 15)
 {
    ClusNo= i/4;
    SecToRead = ClusToSec(ClusNo);      // Calculates which sector to read
    ReadSector(dri, ClusBuff, SecToRead);
    if(ClusBuff->data[0] == 'g' && ClusBuff->data[1] == 'g'
       && ClusBuff->data[2] == 'g')        // Checks if the cluster is manipulated
       {
          CorruptCheck[i]='1';     // If so, mark it in the array
         CorruptCheck[i+(TotBytesInFat/2)]='1';

(omitted code)

}
}
```

When RecoverTrace() has found a manipulated bad cluster, it reads the whole cluster and copies everything but the first sector into a temporary buffer. When the cluster has been read, the function ClearCluster() removes the data from the manipulated cluster. This is to ensure that RecoverTrace() not will read any old hidden cluster data by accident if the cluster turns out to be a legitimate bad cluster in the future.

Before the software updates the FAT, it makes sure that the new root directory, with its files, will still be intact and ready to use. This part applies if the user has chosen the safe method, but will also work if the user have chosen the light method. However, it is not recommended to add new files if the user have chosen the light mode.

The next operation RecoverTrace() does, is to merge the original FAT with the currently working FAT. The two FATs needs to be merged when the user have chosen to hide files with the safe method. This is because the user will want to access the files added to the memory stick after the hiding process is finished. The user will not get access to the files if there is not any merging process taking care of both old and new files.

Figure 17 shows what happens when merging the FATs after using the safe method, and what will happen if the merging process does not take place. If the merging process did not happen, like in the upper right box, then the user will not be able to access the files he/she added after the hiding process had finished.

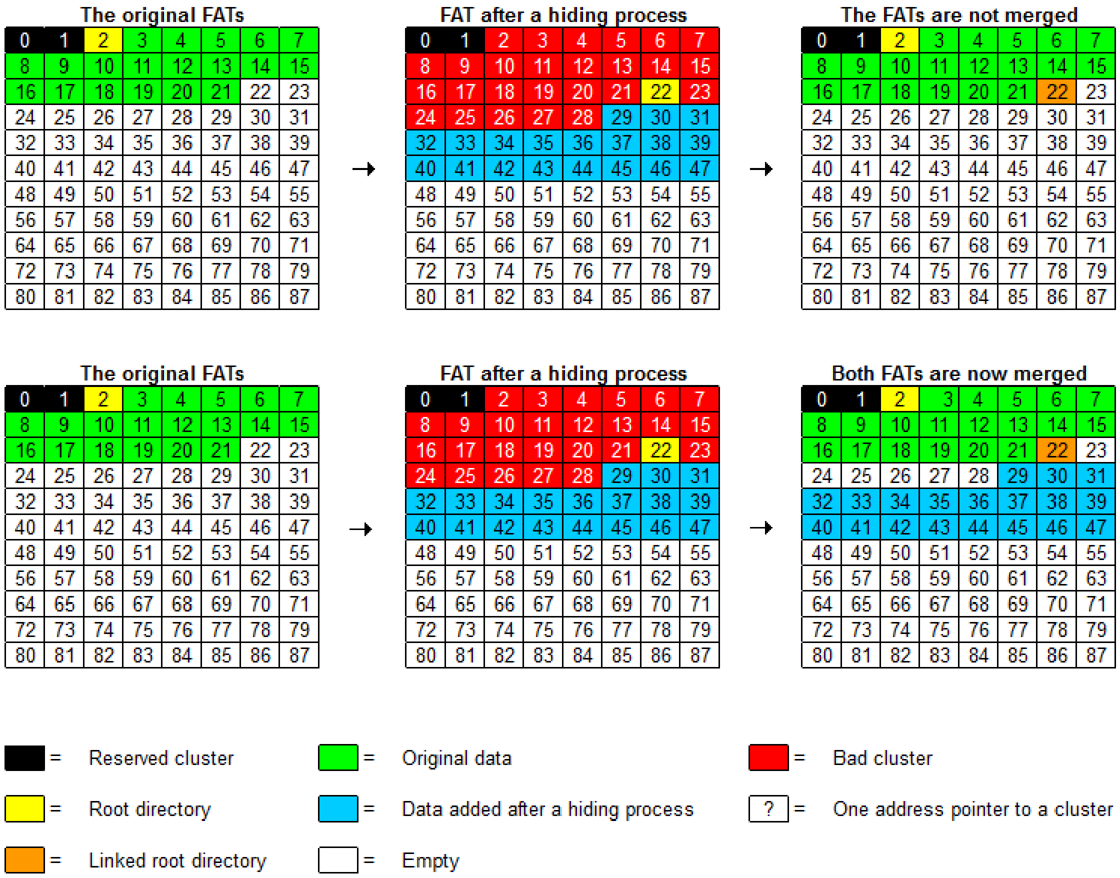


Figure 17: “Merging” the FATs in safe mode.

The last operation of RecoverTrace() writes the original FAT, with a few modifications from the merging process, back to the FAT region.

5.4 Root cluster and file handling

After the FAT region has been restored by RecoverTrace(), the next task is to recover the hidden files. This process is done with the use of three different functions, named FindAll(), MergeClus() and LinkCluster().

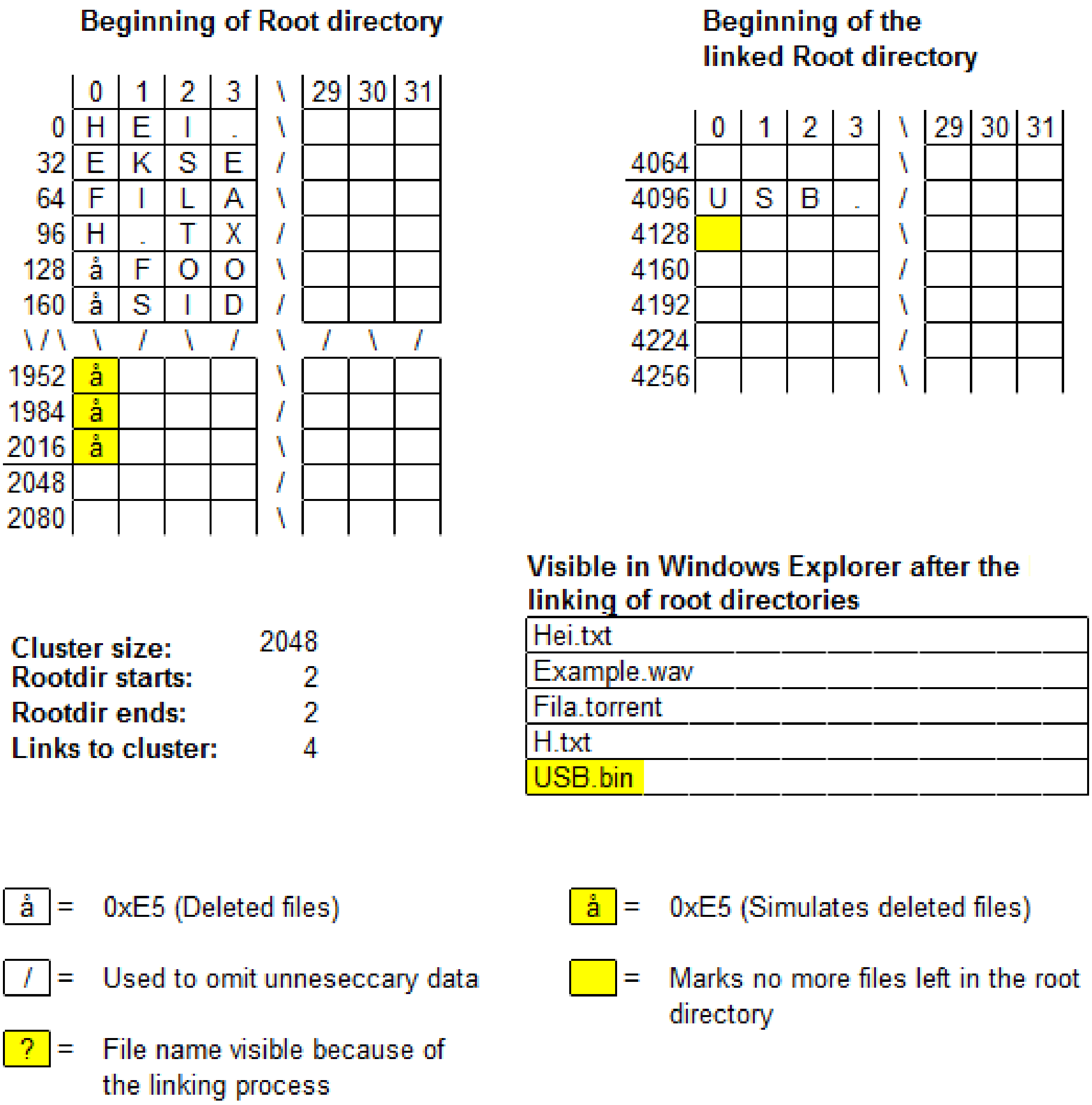
In order to recover hidden files, the root cluster chains for hidden files and for files added after the time of hiding, needs to be linked. This is done with the functions MergeClus() and LinkCluster(). What MergeClus() does, is to traverse the original root cluster chain, searching for the last cluster in the chain. The function scans through the original root cluster chain in the FAT, until the value [FF] [FF] [FF] [0F] is found. (As mentioned earlier in chapter 3.2.4, the value [FF] [FF] [FF] [0F] indicates the end of cluster chain.) Since each cluster is represented by 4 bytes in the FAT, we calculate the value for four bytes at a time, and then continue the operation in the value we found in the four bytes. For example, if cluster 2 points to cluster 4, which points to cluster 6, which points to cluster 10, which contains the value [FF] [FF] [FF] [0F], the traverse will be 2-4-6-10.

After the last cluster in the original root cluster chain has been found, MergeClus() calculates in which sector, in the data region, the cluster containing [FF] [FF] [FF] [0F] starts. This is done by using the formula shown below:

SecToLink = StartDataregion + ((temp - 2) * SecPerClus);

Here, temp is the number of the last cluster in the original root cluster chain. StartDataregion indicates the start of the data region, and when adding it with (temp-2), the two reserved clusters in the FAT are skipped. Finally, by multiplying it with SecPerClus, we convert from cluster number to sector number.

By knowing the sector number for the cluster where the original root cluster chain ends, it is possible to link the original chain with the cluster chain, which is representing files added after a hiding process. In order to link those two chains, two things need to be done. First, all directory entries in the last cluster (marked [FF] [FF] [FF] [0F]) of the original root cluster chain not containing files, need to start with the value 0xE5. This is because the OS will stop showing files when it finds a 0 in the beginning of a directory entry. The LinkCluster() function fixes this issue by scanning through all sectors in the cluster, replacing any 0's it finds with the value 0xE5. By doing this, the OS will continue to search through the old root cluster and then continue to the new root cluster.



Beginning of Root directory

	0	1	2	3	\	29	30	31
0	H	E	I	.	\			
32	E	K	S	E	/			
64	F	I	L	A	\			
96	H	.	T	X	/			
128	å	F	O	O	\			
160	å	S	I	D	/			
\ / \	\	/	\	/	\	/	\	/
1952	å				\			
1984	å				/			
2016	å				\			
2048					/			
2080					\			

Beginning of the linked Root directory

	0	1	2	3	\	29	30	31
4064					\			
4096	U	S	B	.	/			
4128					\			
4160					/			
4192					\			
4224					/			
4256					\			

Cluster size:	2048
Rootdir starts:	2
Rootdir ends:	2
Links to cluster:	4

Visible in Windows Explorer after the linking of root directories

Hei.txt
Example.wav
Fila.torrent
H.txt
USB.bin

å = 0xE5 (Deleted files)

å = 0xE5 (Simulates deleted files)

/ = Used to omit unneseccary data

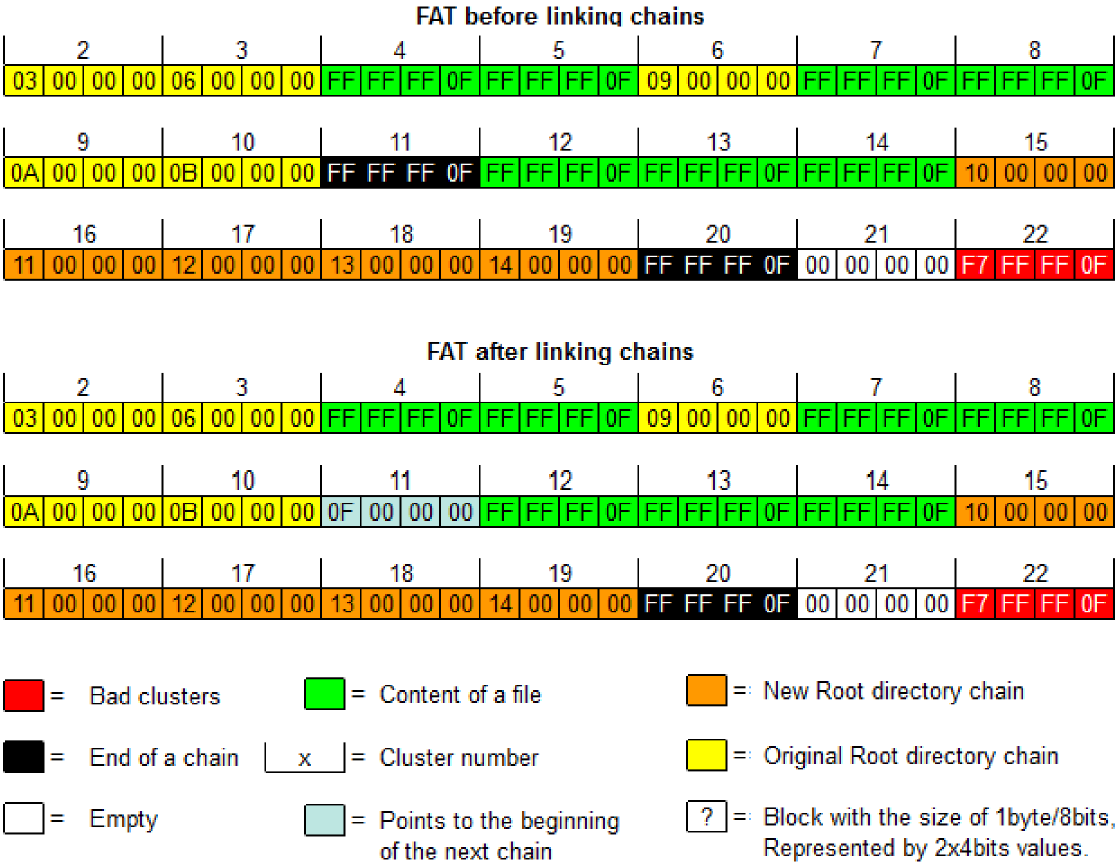
= Marks no more files left in the root directory

? = File name visible because of the linking process

Figure 18: How the linking between two clusters works.

Figure 18 shows how the linking between two clusters work within two given clusters. The original root directory is displayed in the table on the left side with several files, while the new root directory is on the right side and contains one file. In the table on the left, we can see several boxes marked in yellow with an 'å' inside. This simulates a deleted file and the computer will keep looking until the end of the cluster. When the computer have reached the end of the cluster, it jumps to the next cluster which is the new root directory. The computer finds the USB.bin file and checks the next first block which is marked in yellow. That block is zero meaning that there are no more files stored on the memory stick.

The second thing that needs to be done in order to link the clusters, is to edit one value in the FAT. This way, the end of the old root cluster chain will point to the first cluster in the new root cluster chain. This is done by adding the value for the new chain, in the entries in the FAT, which represents the last cluster of the old chain. For instance, as seen in figure 19, when the old root cluster chain ends in cluster 12 and the new starts in cluster 16, the four bytes in the FAT representing cluster 12 will be set to 16.



FAT before linking chains

2	3	4	5	6	7	8
03 00 00 00	06 00 00 00	FF FF FF 0F	FF FF FF 0F	09 00 00 00	FF FF FF 0F	FF FF FF 0F

9	10	11	12	13	14	15
0A 00 00 00	0B 00 00 00	FF FF FF 0F	FF FF FF 0F	FF FF FF 0F	FF FF FF 0F	10 00 00 00

16	17	18	19	20	21	22
11 00 00 00	12 00 00 00	13 00 00 00	14 00 00 00	FF FF FF 0F	00 00 00 00	F7 FF FF 0F

FAT after linking chains

2	3	4	5	6	7	8
03 00 00 00	06 00 00 00	FF FF FF 0F	FF FF FF 0F	09 00 00 00	FF FF FF 0F	FF FF FF 0F

9	10	11	12	13	14	15
0A 00 00 00	0B 00 00 00	0F 00 00 00	FF FF FF 0F	FF FF FF 0F	FF FF FF 0F	10 00 00 00

16	17	18	19	20	21	22
11 00 00 00	12 00 00 00	13 00 00 00	14 00 00 00	FF FF FF 0F	00 00 00 00	F7 FF FF 0F

🟥 = Bad clusters

⬛ = End of a chain

□ = Empty

🟩 = Content of a file

x = Cluster number

🟦 = Points to the beginning of the next chain

🟧 = New Root directory chain

🟨 = Original Root directory chain

? = Block with the size of 1byte/8bits, Represented by 2x4bits values.

Figure 19: How the linking process works.

When MergeClus() and LinkCluster() have completed their tasks with linking of clusters, the function FindAll() will finish the recovery process. This is done with restoring the root cluster to its original value, which by default is the value 2. Now, all clusters containing files are linked as a single root cluster chain, starting from cluster 2. As a result of this, all files on the memory stick are now visible.

In addition to link clusters and editing root cluster values, the file recovery process also involves a function to handle problems with duplicate file names. This function is called NameCheck(), and will be described in detail in the following section.

5.5 Elimination of duplicate file names

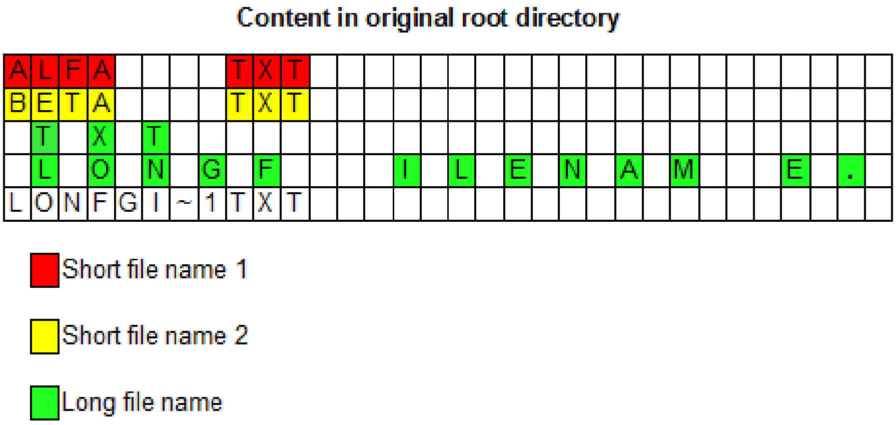
One major issue while recovering files, was the handling of duplicate file names. With visible files it is not possible to create two or more files with identical names, but this is not the case for hidden files. The issue with duplicate file names is due to occur when the user creates a file with an identical name as a currently hidden file. When recovering the hidden file, a possible outcome will be two identical file names showing up, but unfortunately with the contents of one of the files stored on both of them.

The main principle of the name check is to compare the hidden files name with the unhidden ones. Whenever two identical file names are found, there is a need to change one of them. Our implementation will always change the name of the most recently added file (the unhidden), and the name changing is done by adding a number to that file name.

Our first implementation scanned through the root directories, searching for both hidden and unhidden files on the memory stick. However, we considered that to be rather inefficient and concluded that another method was required. After a suggestion from our employer, we found a more efficient way. In the new method, all the directory entries for the hidden files are copied into a local buffer before any name comparisons are done. By doing so, there is only need to read the root directory with the hidden files once, instead of reading it from the memory stick for every unhidden files.

The implementation of preventing files with identical file names involves five different functions: CopyFiles(), NameCheck(), NameCheckShort() (hereby NCS()), NameCheckLong() (hereby NCL()) and ChangeName(). Below, we will explain the main idea and functionality of each of these functions.

The CopyFiles() function is the first function involved in the process of handling duplicate file names. Here, the function scans through the root directories containing the hidden files and copies the directory entries for each of them, to a local buffer. In the figure 20, we see an example of how the buffer may look after a copying process is finished. In this example, we can see that two files with short file name and one with a long file name have been hidden.



Content in original root directory

A	L	F	A					T	X	T																					
B	E	T	A					T	X	T																					
	T		X		T																										
	L		O		N		G		F					I		L		E		N		A		M				E		.	
L	O	N	F	G	I	~	1	T	X	T																					

🟥 Short file name 1

🟨 Short file name 2

🟩 Long file name

Figure 20: Buffer with hidden files.

The main difficulty with implementing the CopyFiles() function was to keep track of values regarding the root directory and the buffer. The problem was that CopyFiles() was called several times, and for each it needed to operate with different values representing how much was copied from the root directory and where to copy it in the buffer, respectively. The problem was solved by passing those values as arguments to the function, with the help of the int array cop[]. The first value in cop[] represents how far the copy process had reached in the root directory, and was updated in MergeClus(), before sending it to CopyFiles(). The second value in cop[] represents where to copy a directory entry in the buffer, and this value was updated in CopyFiles itself.

Next, the NameCheck() function is called. This function scans through the new root cluster chain and searches for unhidden files. For each cluster it traverses in the new root directory chain, the function scans through the directory entries stored in the current cluster. For each directory entry, the function decides whether the directory entry represents a short or a long file name. This is done by checking whether the value in byte number 11 of the various directory entries contains the value 0x0F (15 in decimal) or not. If it is so, the file name is long and NCL() needs to be involved. In the opposite case, the directory entry represents a short file name, and hence, the NCS() function needs to be called. In figure 21, the files added after hiding is shown. Here, it can be seen that three files with short file names and one with a long file name (identificators marked in yellow) have been added after time of hiding. As a result of this, NCS() needs to be called three times and NCL() once.

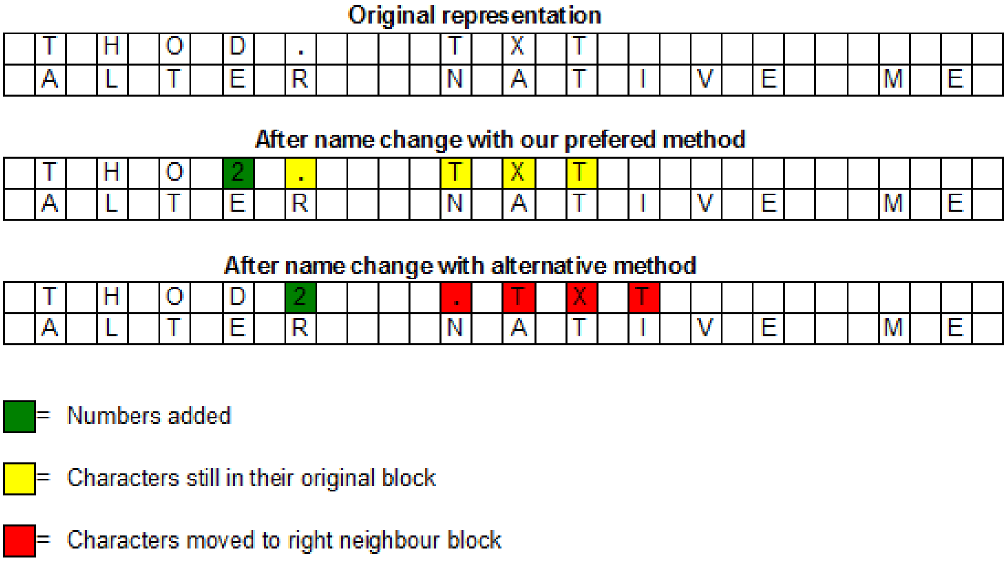

Original representation
After name change with our prefered method
After name change with alternative method
= Numbers added
= Characters still in their original block
= Characters moved to right neighbour block

Figure 21: Short or long file name check

The check in NCS() and NCL() is done by comparing byte by byte in the directory entry for the unhidden file, with the directory entry for hidden files. If the function manages to scan through all hidden files' directory entries without any matches, the unhidden file name currently checked is unique and do not need to have its name changed. In the opposite case, two file names are identical and ChangeName() needs to be involved. In figure 22, we can see that the two file names “longfilename.txt” and “alfa.txt” are duplicates and need to be changed.

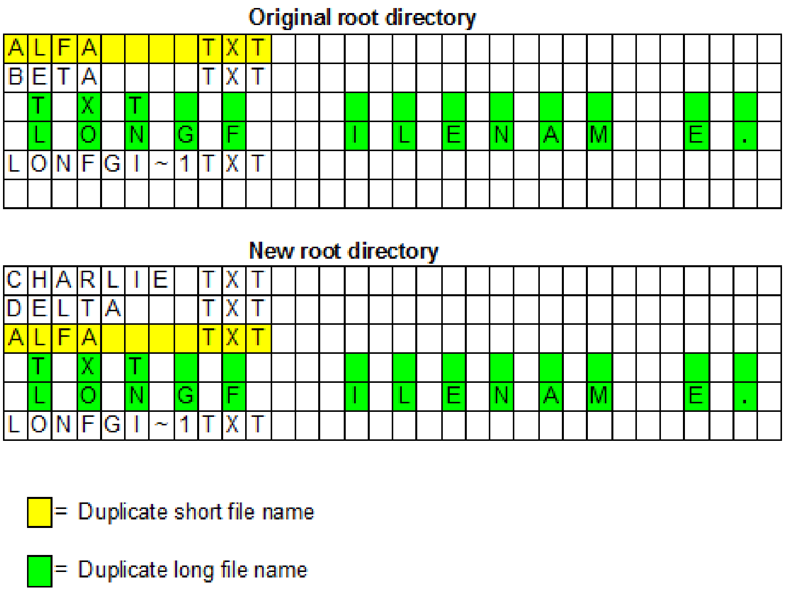


Figure 22: Namecheck

In ChangeName(), duplicated files are given a new name. The naming convention used is that the function appends a number to the file name, and then performs a new check to discover whether the new file name is unique. For short file names, the number is added at the end of the file, with one exception; If the number forces a new directory entry to be created for the renamed file and hence must be converted to a long file name, the number is stored in the last character of the file name.

Regarding long file names, the number replaces the character preceding the dot. Another, but more complicated, solution would be to keep the original file name and adding a number after the last letter. As shown in the middle table of figure 23, this would cause replacing the dot with the number, and then move the dot and the file extension characters one place; to their neighbour name block to the right. As mentioned, we considered that to be a complicated case, especially if not all characters would have fit in the existing long file name directory entry, and hence, forcing a new directory entry for the long file name to be created.

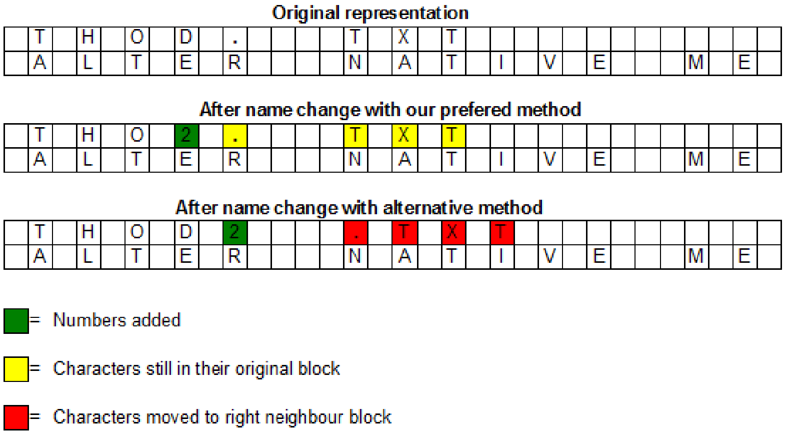

Original representation
T H O D . T X T
A L T E R N A T I V E M E
After name change with our prefered method
T H O 2 . T X T
A L T E R N A T I V E M E
After name change with alternative method
T H O D 2 . T X T
A L T E R N A T I V E M E
🟩= Numbers added
🟨= Characters still in their original block
🟥= Characters moved to right neighbour block

Figure 23: Two methods of renaming files with long file names.

In our sample case, the two files named “alfa.txt” and “longfilename.txt” need to be changed. As shown in figure 24, they are renamed to “alfa2.txt” and “longfilenam2.txt” after ChangeName() is done.

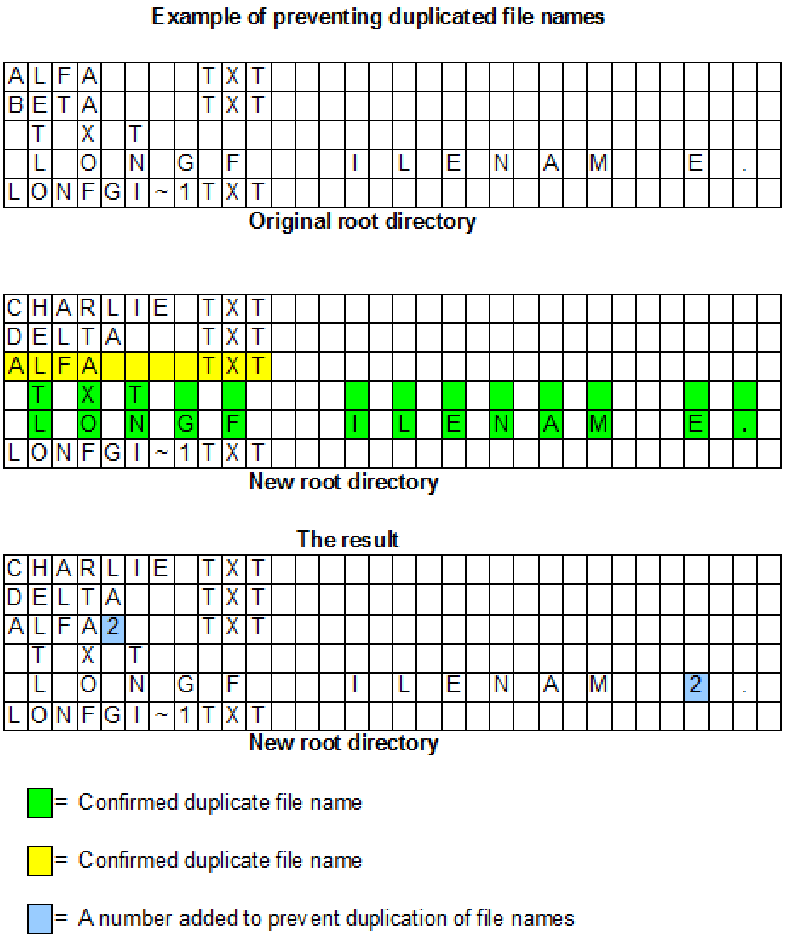

Example of preventing duplicated file names
ALFA TXT
BETA TXT
T X T
L O N G F I L E N A M E .
LONFGI~1TXT
Original root directory
CHARLIE TXT
DELTA TXT
ALFA TXT
T X T
L O N G F I L E N A M E .
LONFGI~1TXT
New root directory
The result
CHARLIE TXT
DELTA TXT
ALFA2 TXT
T X T
L O N G F I L E N A M 2 .
LONFGI~1TXT
New root directory
🟩= Confirmed duplicate file name
🟨= Confirmed duplicate file name
🟦= A number added to prevent duplication of file names

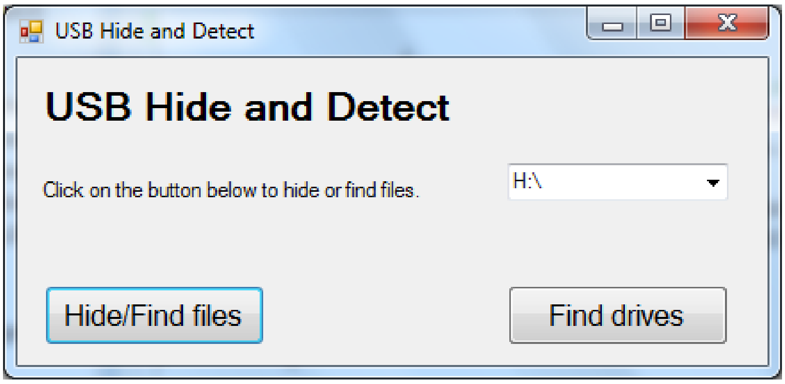
Figure 24: ChangeName() function

However, it must also be checked whether files named “alfa2.txt” or “longfilenam2.txt” exists among the hidden files. If that is the case, the files will be renamed once more, this time with replacing 2 with the number 3, which will rename the unhidden files to “alfa3.txt” and “longfilenam3.txt”. After this, a further check to see whether any of these file names exists, needs to be performed. In this case, “alfa2.txt” and “longfilenam2.txt” are unique, so no more than one name change needs to be done.

The implementation of the five functions to handle duplicate file names was challenging. Especially the combination of the NameCheck() and ChangeName() functions required a lot of bug testing and changes. Some problems we encountered was getting the name changing correct and to change file names only when needed. Common errors were that file names were renamed even though no identical file names existed, or that no changing were done even when required. Also, we struggled to implement new checks after a file had been renamed once. In general, handling duplicate long file names was more difficult to implement than short file names, because long file names have more directory entries to take into consideration. There was also a problem with placing the number in the correct position when handling long file and directory names.

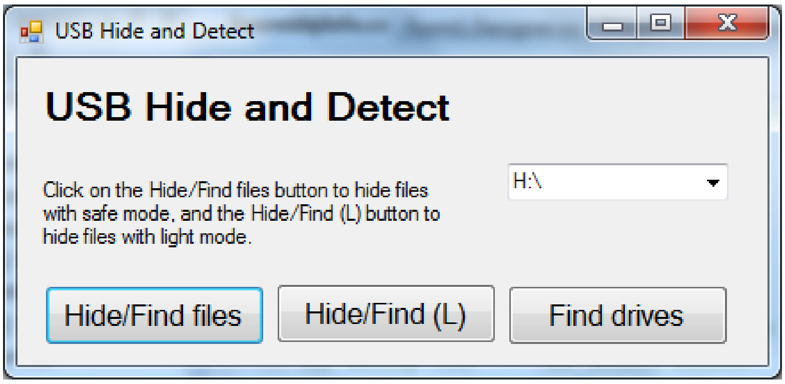
5.6 GUI

With the GUI, the user gets a window on the screen which shows a hide/find button for the safe method. The button will not work until the user has selected what drive to hide or find files on. The software will find all the drives that are removable and has a FAT32 file system and select one automatically. When the user presses the Hide/find button, the software will execute the safe hide. If the user has already hidden files with the software, the find function will execute. If the program is executed from the command line with the parameter "e", the GUI will offer a choice to hide or find files with the light button. The figures below show how the two different versions look like


USB Hide and Detect

Click on the button below to hide or find files.

Figure 25: Normal version


USB Hide and Detect

Click on the Hide/Find files button to hide files with safe mode, and the Hide/Find (L) button to hide files with light mode.

Figure 26: Advanced version

The .exe file needs access to the DLL to work properly. The GUI calls the run() function from the DLL, and executes it with the parameters provided.

6 Software Analysis

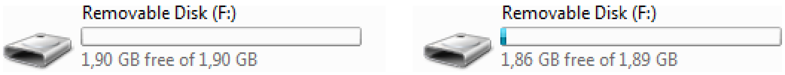
In this chapter, we will present an analysis of our software. We will first show an analysis of the level of protection the software provides, then a performance analysis which compares the efficiency of the software with a cryptographic mean of protecting files. Finally, we will discuss bugs that appeared through testing the software.

6.1 Level of protection

In this section, we will show an analysis of the level of protection provided by our software. This means that we will take a look at how hard or easy it is to find out that files are hidden by our software. The analysis has been done using two different scenarios. In the first, we analysed what might be discovered on first sight when just studying the memory stick. In the second part we will see what might be concluded when using the tool dd (data description) and the file forensic tool kit The Sleuth Kit (TSK) [6].

6.1.1 First sight discovery

When an attacker opens the memory stick in Windows Explorer (WE), the first thing he will see is the space allocated on the memory stick.



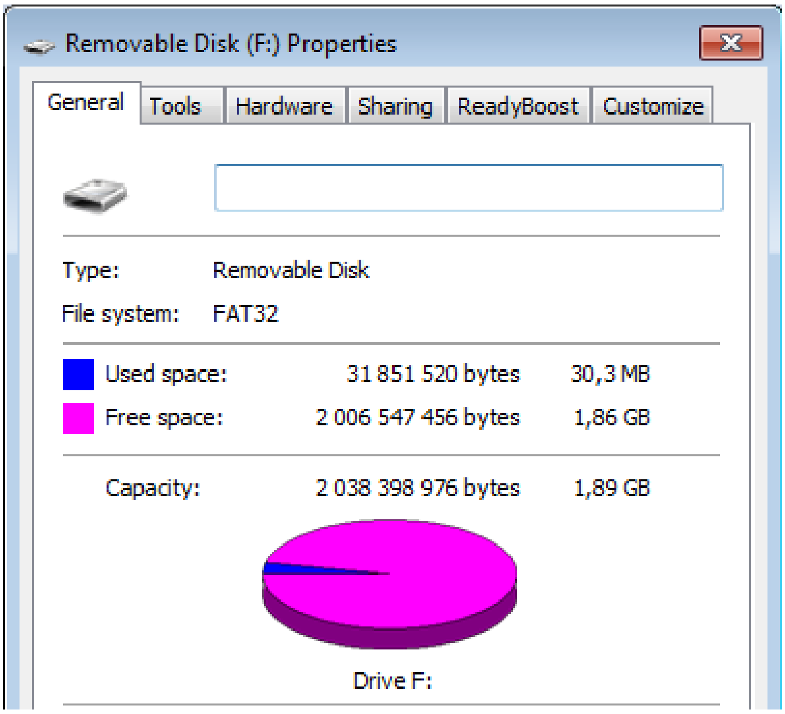
Removable Disk (F:)
1,90 GB free of 1,90 GB

Removable Disk (F:)
1,86 GB free of 1,89 GB

Figure 27: Space allocation viewed in WE

As explained in figure 27, some cluster sizes will show that space have been allocated while other sizes will not. In figure 27, the memory stick on the left side uses 4096 bytes cluster size, while the same sized memory on the right side used stick 1024 bytes as cluster size.

When the attacker checks the content of the memory stick, he will not find any files or the files he is looking for in particular. Some attackers might leave the memory stick after not discovering any files, while others may be tempted to check the memory stick's properties. A supposed "empty" memory stick will show the space allocated. In figure 27 the memory stick is using 1024 bytes as its cluster size.



Type: Removable Disk

File system: FAT32

🟦 Used space: 31 851 520 bytes 30,3 MB

🟪 Free space: 2 006 547 456 bytes 1,86 GB

Capacity: 2 038 398 976 bytes 1,89 GB

Figure 28: The space a FAT copy allocates

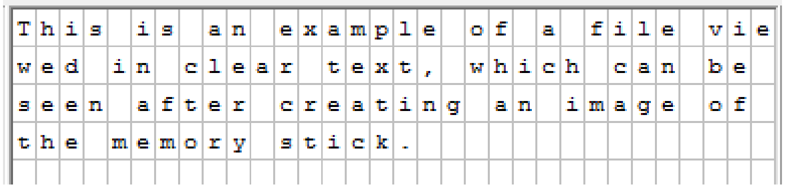
If WE does not show any space allocated, like in figure (WE_space_allocation), the chances of an attacker would pursue a more advance analysis is low. It depends on the intentions of the attacker. For instance, a nosy colleague and a spy will probably have different intentions. While the nosy colleague might be more interested in the memory stick owner's private life and probably just quit his attempt to find files when nothing is visible on the memory stick, the spy will try to do anything in order to get his hands on secret information.

6.1.2 Analysis using advanced tools

For a deeper analysis, we installed Autopsy 2.24 [6] with TSK 3.2.1 and on a virtual Ubuntu Linux machine in VMware Player [14] for testing what a forensic tool might discover. TSK is an open source and free forensic tool which is basically a collection of different forensic software. Autopsy is an open source web based (running as localhost) GUI (Graphical User Interface) for TSK.

TSK/Autopsy requires an image file of the memory stick for processing. This we extracted with the Linux/Unix program dd. It can copy a drive, byte by byte, to a file, which is then examined using Autopsy. In an image, a text file will be written out in readable format, and therefore not safe from reading, even when hidden. This way, we can conclude that hidden files are not safe from reading, and that the best way to protect a text file from reading is cryptography. However, a regular PC user would probably not be able to produce an image of the memory stick and find the information.

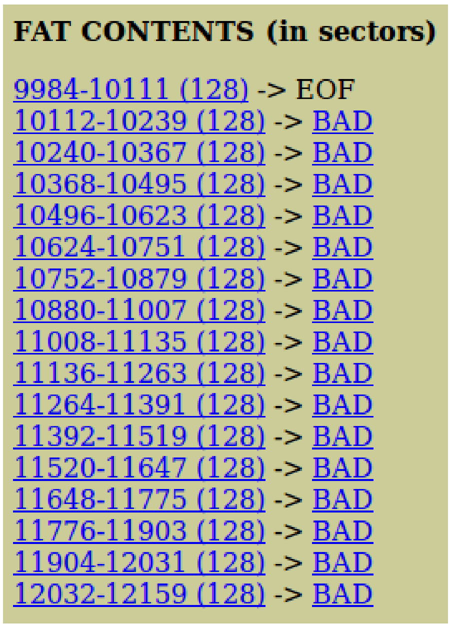
Figure 29 shows that by saving a text (.txt) file on the memory stick and then creating an image of the memory stick, the attacker will be able to see the content regardless of whether the files have been hidden or not. The image is a binary file we created by our software after a file was hidden. Another method to create an image of the memory stick is to use the dd (data description) program in Linux, as explained above.



```
This is an example of a file vie
wed in clear text, which can be 
seen after creating an image of 
the memory stick.
```

Figure 29: file contents in clear text

The Sleuth Kit is not able to find files hidden with our method. It will however find, as shown in figure 30, that a lot of sectors are marked as bad, which might raise suspicion.



FAT CONTENTS (in sectors)

9984-10111 (128) -> EOF
10112-10239 (128) -> BAD
10240-10367 (128) -> BAD
10368-10495 (128) -> BAD
10496-10623 (128) -> BAD
10624-10751 (128) -> BAD
10752-10879 (128) -> BAD
10880-11007 (128) -> BAD
11008-11135 (128) -> BAD
11136-11263 (128) -> BAD
11264-11391 (128) -> BAD
11392-11519 (128) -> BAD
11520-11647 (128) -> BAD
11648-11775 (128) -> BAD
11776-11903 (128) -> BAD
11904-12031 (128) -> BAD
12032-12159 (128) -> BAD

Figure 30: Bad clusters viewed from Autopsy

Even though TSK does not find the files, the files are still present in their original form in the data region and can be found during a "manual" search through an image. Especially contents of text files will be easy to find and read with merging different text found in clear text in the image. It is also easy to discover "magic numbers", which are hex strings identifying different file formats [15]. For example, the hex string "FF D8 FF E0" identifies .jpg files [15]. To create an image in dd, the following has to be typed in the terminal:

```
sudo dd if=/dev/sdb of=image.bin.
```

The result of the analysis was as anticipated. Because we do not move or encrypt any data, it will not be difficult to get hold of the hidden data for anyone with advanced forensic tools. If the sort files function in Autopsy is used, no files will be found.

6.2 Performance analysis and comparison

Below, we will present an analysis on how fast our software performs hiding and recovering of files. We will present separate results for the software, and compare the performance with a cryptographic protection method.

For the measurement of the speed of our software, we considered three different parameters to be relevant. All the three parameters are chosen because the performance of the software varies depending on these parameters. The first parameter used, is the size of the memory stick. The second is the cluster size used on the memory stick, while the third parameter is which method the software used for hiding files.

We have done one hundred measurements for each combination of parameters. For simplicity and to avoid human errors, we decided to create a script which automatically executed the software the desired number of times and with the various combination of parameters. All results for hiding and finding was continuously written to two separate text files. The results were imported into spreadsheets for analysis. We calculated average and median values for each cluster for each memory stick. The median analysis was made to discover if any data was skewing the average value. Most of the time the average and median was relatively close, indicating that there was little skewing.

The performance tests were performed on four different memory sticks of different sizes: 4,8,16, and 32 GB (GigaByte). The tests were run on computers at GUC with the same hardware:

- Intel(R) Core(TM)2 Duo CPU (Central Processing Unit) E6750, 2.66 GHz (GigaHertz)
- 4 GB RAM (Random Access Memory)
- USB (Universal Serial Bus) 2.0
- 32-bit Windows 7 Professional OS

The hide operation was, as expected, slow on small cluster sizes and fast on large ones. The find operation, however, seemed to go slower and slower on large cluster sizes, after first having a drop in time used. The three figures shown below, illustrates the average test results for hide with safe and light mode and find for a 32 GB memory stick.

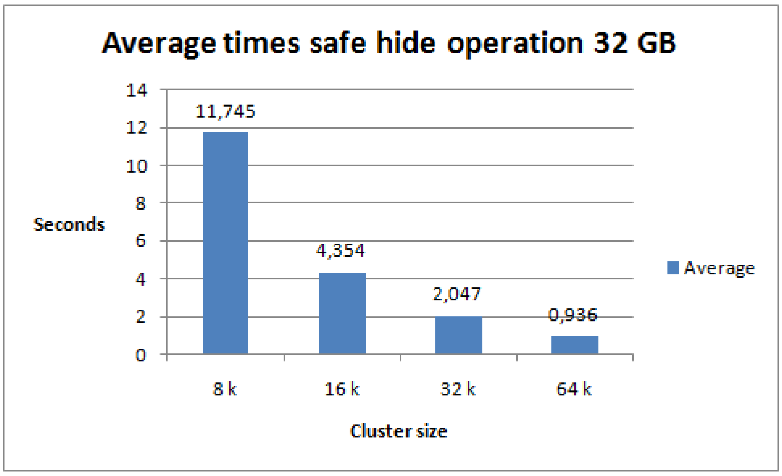


Figure 31: Average times of the safe hide function on a 32 GB memory stick

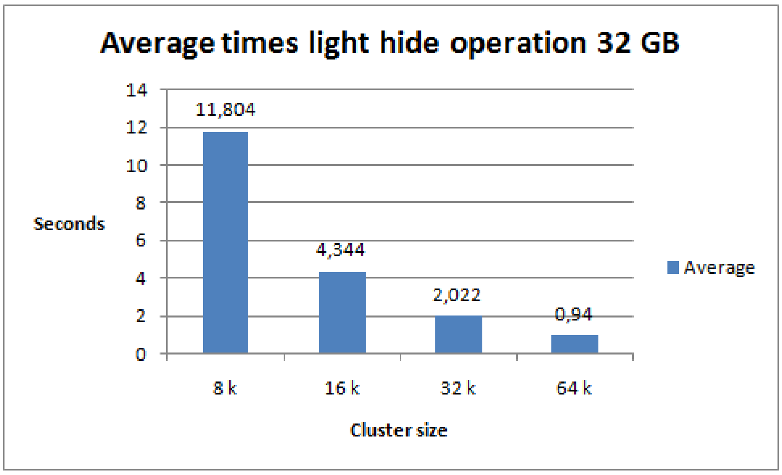


Figure 32: Average times of the light hide function on a 32 GB memory stick

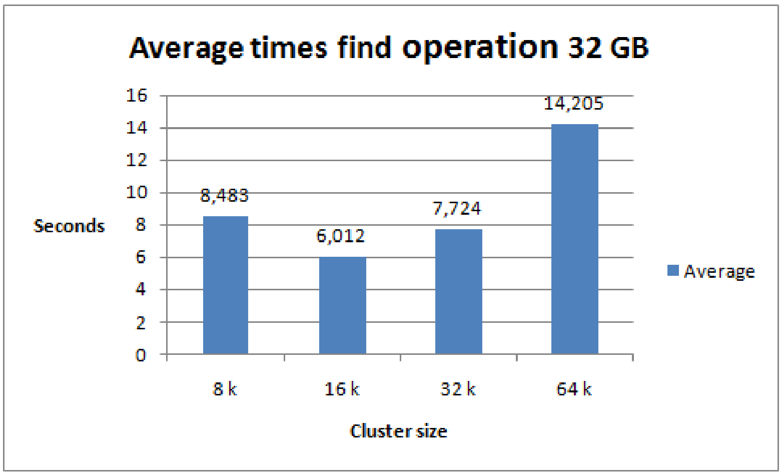


Figure 33: Average times of the find function on a 32 GB memory stick

More results can be seen in the appendix A.

We also wanted to test how fast an encryption program was compared to our software. One suitable program for testing encryption is the popular TrueCrypt [3]. It is free and open source software, but it can be difficult to use, since it has a lot of options [?] and requires management of cryptographic keys. The main problem with cryptographic keys is that it might be unfortunate if the user encrypts his hard drive and later forgets the password needed to decrypt it.

When the cryptographic tool was tested, we first checked how long it took to encrypt an entire memory stick with a computer with a 2.66 GHz processor and 3.25 GB RAM, with regular AES encryption through TrueCrypt. The result we found, was that it took 1 hour and 40 minutes to encrypt a 32 GB memory stick with in-place encryption, which means that the files on the memory stick are encrypted. The default option in TrueCrypt is to create an encrypted volume and format the disk with random data, so it can be used as an encrypted volume [4]. This method is faster, and it took only 20 minutes to create a 32 GB volume with a cluster size of 16384 bytes. The average speed of the encryption was 24.9 MB/s. With our method, as shown in figure 31 and 32, hiding files on 32 GB memory stick takes between 1 and 12 seconds, depending on the cluster size used.

Even if encryption was infinitely fast, the transmission speed of USB 2.0 is 60MB/s [16], which means that it would take about 9 minutes to encrypt a 32 GB harddrive.(32 768 MB / 60 MB = 546 seconds = 9 minutes). The encryption that was tried out on the same type of computer which we tested our software, and the encryption took about 25 MB/s in one of the fast options. This means that the encryption speed is lower than the transmission speed.

Encrypting entire volumes is just one of the options that TrueCrypt offers. A more practical way to do encryption, is to make a virtual hard drive. This is because virtual drives are handled like ordinary files and can be mounted on any computer with TrueCrypt installed on it [4]. This file can be of any size the user wants. Making a virtual hard drive 10 MB in size, takes only a couple of seconds. Making a volume the same size as a memory stick will take about as long as encrypting the memory stick normally.

6.3 Conclusions of the analysis

The first part of analysis showed that files hidden files are easy to find with forensic tools. Furthermore, since the software does not use any encryption when hiding files, it is also easy to read the contents of files found. However, to find files and later read the contents of them is relatively complicated for common PC users. If the mean of using the software is to protect files against nosy colleagues or regular person finding a lost memory stick, our software might be sufficient.

The major advantage with using our software, as shown in chapter 6.2, is that it is faster than using encryption.

6.4 Known bugs

Below, we will describe possible known bugs found in our software. See chapter 7 for a brief discussion about the chances of having another bachelor thesis and possible improvements we recommend.

- When a certain amount of data is hidden by using the safe method, the RemoveTrace() funtion writes the FAT copy into the reserved region. This should not be possible and we think the bug is happening when the RemoveTrace() function is computing the next free cluster. For some reason, the FreeCluster variable might become a negative number, which may cause the function to write in the reserved and FAT region. If the free cluster variable is 2, then the data would be written at the beginning of the data region. Any number smaller than 2 would cause the function to write to a sector belonging to either the FAT region or the reserved region. Smaller files hidden with the safe hide function will be hidden and found correctly. However, several big files (greater than 200 MB) will cause problems, resulting in that the user will see a lot of garbage when viewing the contents on the memory stick. Even if the hiding function appears to have worked correctly, by showing no files, the find function will most likely result in an error about protected memory violation. The light mode works correctly on all file sizes.
- A bug that triggers for an unknown reason, is causing the files to stay hidden, even though the allocated space can be seen through WE. Something has to trigger the bug and we will recommend taking a closer look on which number the SetValue() function receives, because the number it receives is the value of the root directory.
- The software will not be able to run if the memory stick has a cluster size of 512 bytes. The problem may be solved if point number five in section 7.2 - ’Possible improvements’ is implemented as the new method of hiding and recovering files.

- The predefined sector used to decide whether a cluster is legitimately bad or not, is recognised by four consecutive 'g's. The bad thing is that if a user creates a .txt document with the first four letters as 'g' and then hides the file with the safe mode, then the software will crash. The chances of this happening is very small and can be even more reduced by checking up to 504 more bytes for 'g' or any other letter/symbol.
- The best thing to do is to use a symbol with a value greater than nine (ASCII) which is infrequently used in a .txt file like any letter in the alphabet. Expanding the number of bytes used to check for a manipulated cluster is also recommended.

7 Future work

If a new bachelor thesis could be created, then it may be about creating a more efficient and secure version of the software. More functionality, like hiding individual files and the possibility to be able to see hidden files through the GUI (Graphical User Interface) before the user recover hidden files, may also be considered in a possible future work. Also, implementing the software for use on Linux and Mac OS X, may be a possible future implementation.

To conclude, an analysis of how the current source code works and make suggestions for a new and improved version both in terms of security, methodology and efficiency, may be a starting point for creating a future bachelor thesis regarding hiding files on the FAT32 file system.

7.1 Possible improvements

- Rollback feature: As of today, the software tries to hide or find the FAT (File Allocation Table) region as if the FAT works normally. There is no rollback feature which will recover the damages if the hiding or recovering processes gets interrupted. A way to solve this is to create a function which scans through the FAT and then recover the parts of the FAT the software managed to copy.
- Currently there are two separate versions of the software, one for 32 bits and one for 64 bits OS. The reason for this is that the Bdll.dll file have to be compiled in either a 32bits or 64bits, caused by an error message if the software is executed with the wrong .dll file. Instead of having one 32 bit and one 64 bit version, we should be able to call the specific .dll file, depending on whether the user runs a 32 bits or 64 bits OS.
- Compatibility to run the software on Linux operating systems and on Mac OS X.
- Improve recovering speed by having one or more clusters dedicated to save which bad clusters the software has manipulated. As of today, choosing the safe mode to hide several gigabytes will result in a much higher recovering time than light mode. This is because the software checks every bad cluster by reading one cluster from the memory stick. With several GBs stored, every cluster address used to store those will be marked as bad. Both the RemoveTrace() and RecoverTrace() would have to be rewritten because if the dedicated cluster is going to be implemented, there would be no reason to have one predefined sector in the beginning of each cluster as it is today.
- Decrease the time used by the hiding process by letting the computer check how many consecutive free clusters the memory stick have. Then, instead of writing one cluster at the time, write and then write data for 'x' amount of clusters in a single operation.
- Improve security by randomly selecting free clusters instead of selecting the next free cluster. This is done to prevent having one big group of bad clusters on the memory stick. Spreading

the bad clusters all over the memory stick makes it less suspicious to analyses.

- Improve security by forcing the user to enter a pin code in order to prevent other users using the same software to recover hidden data. The pin code may be hidden in a predefined sector the same way as we store the original root cluster value.
- Make the hiding with safe and light mode less suspicious by creating a dummy file with the size of the space the bad clusters allocated. The user may rename this dummy file to anything he wants, because it will be removed when the user recovers the hidden files.
- Improve GUI by adding a status bar/text box which shows the user what the software is doing at any given time.
- The result of any hiding process, regardless of choosing the safe or light method, will show lots of bad clusters in one certain region on the disk. One feasible improvement may be to make the software randomly write the bad clusters on the memory stick.
- After each hiding process, the time spent by the recovery process will increase slightly. Since we believe it is the process of preventing duplicated file names that is the time sink, it should be possible to improve the process by moving new file names to the original root cluster and thereby decrease the size of the root directory chain.

8 Discussions and evaluation

In this chapter, we will evaluate our performance during the work with the bachelor thesis and the thesis itself.

8.1 Evaluation of the work done by the group

The group members have cooperated well during the work with this bachelor thesis, and the workload has been evenly scheduled between the members. The cooperation and communication between the group and our employer and supervisors also worked well.

This was the first time any of the group members participated on a project of this size. Developing a bigger software was also new to us, which lead to many challenges. Certain bugs and flaws in the software and GUI were especially hard to solve, which caused extended working hours on several occasions.

No major disagreements occurred during the thesis period, and all minor disagreements were solved in a satisfying way. The group leader however used his right to veto on different occasions, for instance when he declined turn-ins of the report due to the group having a too heavy workload at that time.

All group members have worked more hours than recommended, which is more than 30 hours a week per person. The work load was evenly spread during the thesis period, but naturally it got more hectic in the end.

8.2 Criticism of the bachelor thesis

We now realise that we should have been more active in the start-up period of the work with the bachelor thesis, in order to quickly understand how to perform the basic operations needed for the assignment. For instance, we spent too to much time trying to find out how to read and write information from/to a memory stick. This lead to a delay of a couple of weeks, which resulted in that we were not able to optimize the software as much as we wanted.

Also, we now feel we should have been more conscious on how to handle merging of code written by different group members. We should have considered that two separated codes which worked like intended when running separately, possibly could cause problems after they were merged.

9 Conclusions

It is possible to hide and recover files by editing the FAT (File Allocation Table) region without using encryption, but it will lack the security encryption provides.

The software provides two methods of hiding files. One is secure and the other one is less secure. Using the less secure version made it possible for recovery tools to restore the files, but looks less suspicious at first glance. The safe method prevents easy access to the file contents by marking the clusters as bad. The safe mode's weak spot is against people who can invest in knowledge about the file system, time and forensic tools.

The software operates extremely fast for larger amount of data than what a cryptographic tool could accomplish.

Bibliography

[1] BBC News. Health records found in Asda car park. BBC; 2010 [cited May 2011]. Available from: http://news.bbc.co.uk/2/hi/uk_news/scotland/tayside_and_central/8661839.stm.

[2] Winnett R, Swaine J. Data on 130,000 criminals lost. Telegraph; 2008 [cited May 2011]. Available from: http://www.telegraph.co.uk/news/politics/2601056/Data-on-130000-criminals-lost.html.

[3] TrueCrypt. TrueCrypt - Free open-source disk encryption software for Windows 7/Vista/XP, Mac OS X, and Linux; 2011. http://www.truecrypt.org.

[4] TrueCrypt. TrueCrypt User's Guide. TrueCrypt Foundation; 2010. Can also be found at: http://www.truecrypt.org/docs/.

[5] Microsoft. FAT: General Overview of On-Disk Format. File System Specification. Microsoft Corporation; 2000. http://msdn.microsoft.com/nb-no/windows/hardware/gg463084.

[6] Carrier B. The Sleuth Kit; 2011. http://www.sleuthkit.org/sleuthkit/.

[7] Microsoft. Startside for MSDN; 2011. http://msdn.microsoft.com/.

[8] Jeremy Davis DD Joe MacLean. Methods of Information Hiding and Detection in File Systems. In: 2010 Fifth International Workshop on Systematic Approaches to Digital Forensic Engineering; 2010. p. 66–69.

[9] Maas C. Freeware Hex Editor XVI32; 2011. http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm.

[10] Mantis. Mantis Bug Tracker; 2011. http://www.mantisbt.org/.

[11] Carrier B. File System Forensic Analysis. Upper Saddle River,N,J: Addison-Wesley; 2005.

[12] Microsoft. FSCTL_LOCK_VOLUME Control Code; 2011. http://msdn.microsoft.com/en-us/library/aa364575%28v=vs.85%29.aspx.

[13] Microsoft. FSCTL_DISMOUNT_VOLUME Control Code; 2011. http://msdn.microsoft.com/en-us/library/aa364562%28v=vs.85%29.aspx.

[14] VMWare. VMWare Player; 2011. http://www.vmware.com/products/player/.

[15] Ogley R. Magic numbers for files; 2011. http://www.astro.keele.ac.uk/oldusers/rno/Computing/File_magic.html.

[16] Compaq ILMNP Hewlett-Packard. Universal Serial Bus Specification; 2000. http://www.usb.org/developers/docs/.

A Analysis results

You can find all our results from the analysis here: `http://hovedprosjekter.hig.no/v2011/imt/in/usbdatahiding/Graphs_hide_and_find.zip`

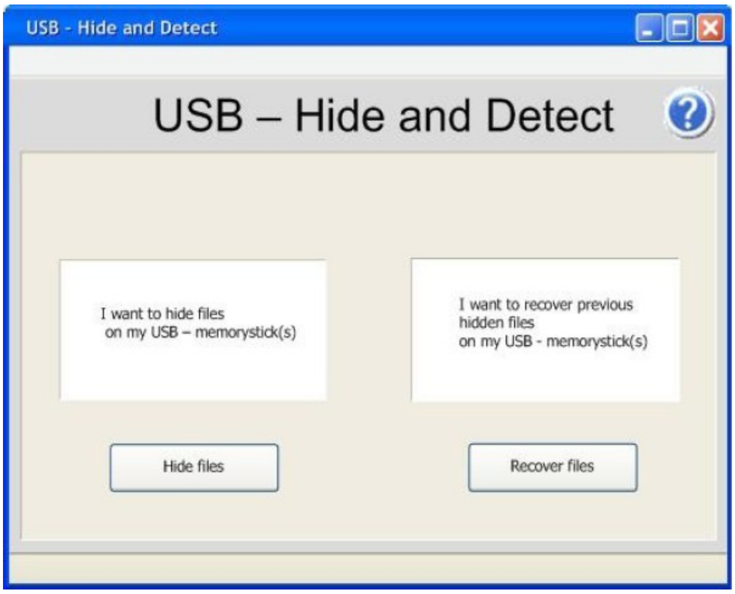
B GUI suggestions

Graphical User Interface Results

- GUI (1):
    - Positive
        - Self explanatory.
        - Easy to use.
    - Negative
        - Too many elements.
        - The two information boxes could have been merged into one box instead.
- GUI (2)
    - Positive
        - Nice presentation of information.
        - The information is sufficient enough.
        - Easy to get a quick overview.
        - Good composition with information first and then the choices below.
        - A suitable GUI for people with little experience with computers.
    - Negative
        - A lot of information. It is boring to watch if you have seen it an x amount of times.
        - Should have had some visible progress bar of some sort to indicate the progress of the hiding or recovering process.
- GUI (3)
    - Positive
        - Technically the user will not need more than two buttons which makes this GUI nice and clean.
    - Negative
        - Way too little information.
        - The user has no clue of what is going on.
- GUI (4)
    - Positive
        - Great for the more 'advanced' user by showing a more detailed overview of what the software do at any given time.
        - Suggestion to merge GUI (2) and GUI (4).
    - Negative
        - No information about what the hide and recover button do compared to GUI (2).

Graphical User Interface suggestions

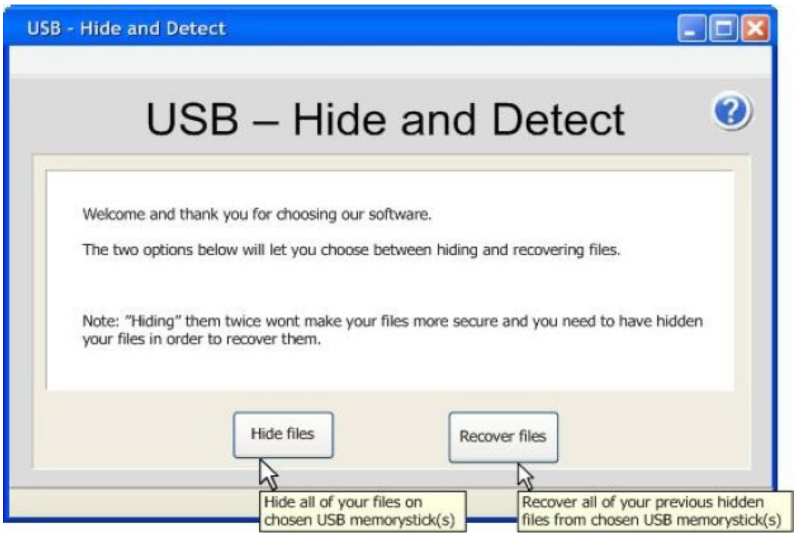
1)



I want to hide files
on my USB – memorystick(s)

I want to recover previous
hidden files
on my USB - memorystick(s)

2)



Welcome and thank you for choosing our software.

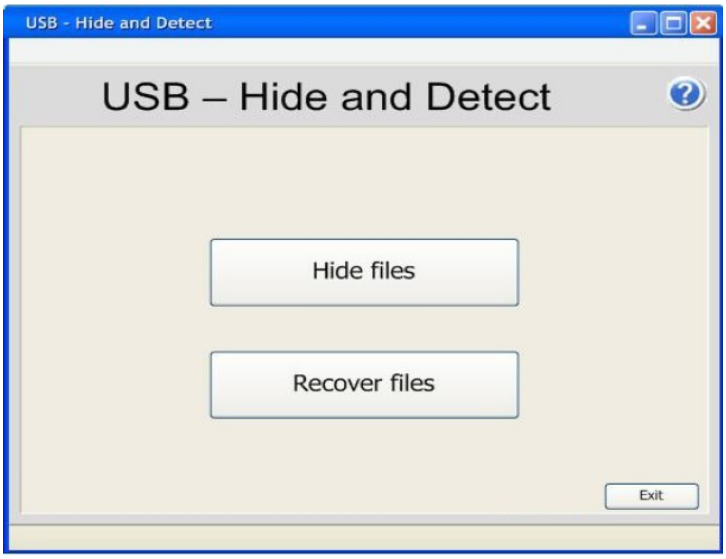
The two options below will let you choose between hiding and recovering files.

Note: "Hiding" them twice wont make your files more secure and you need to have hidden your files in order to recover them.

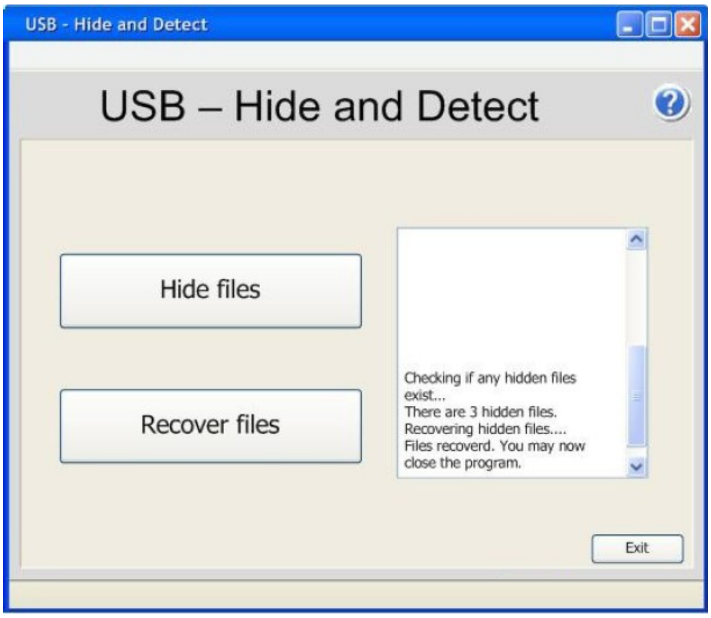
Hide all of your files on chosen USB memorystick(s)

Recover all of your previous hidden files from chosen USB memorystick(s)

3)



4)

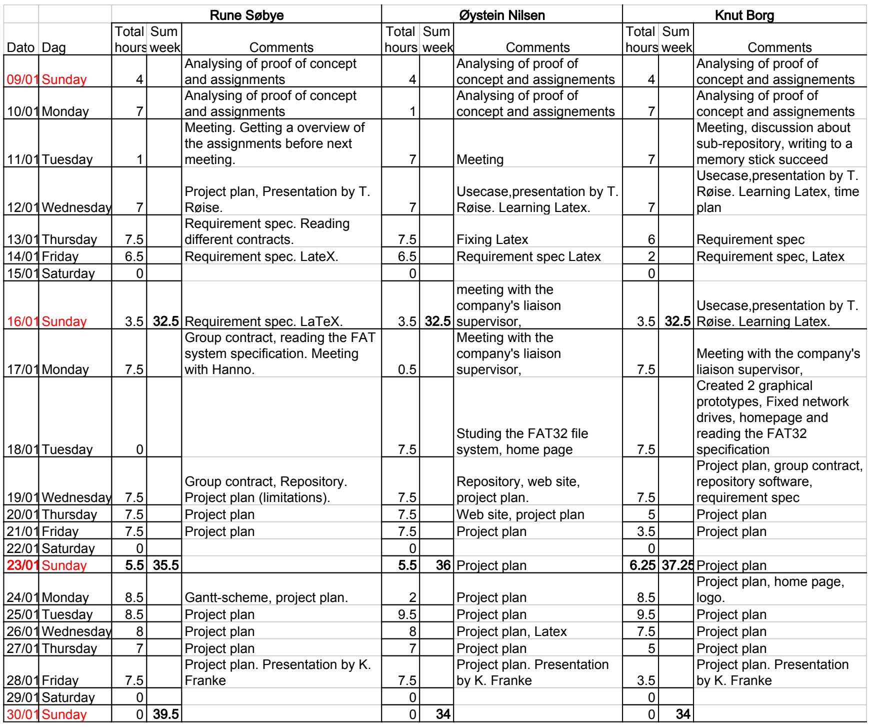


```
Checking if any hidden files
exist...
There are 3 hidden files.
Recovering hidden files....
Files recoverd. You may now
close the program.
```

C Script

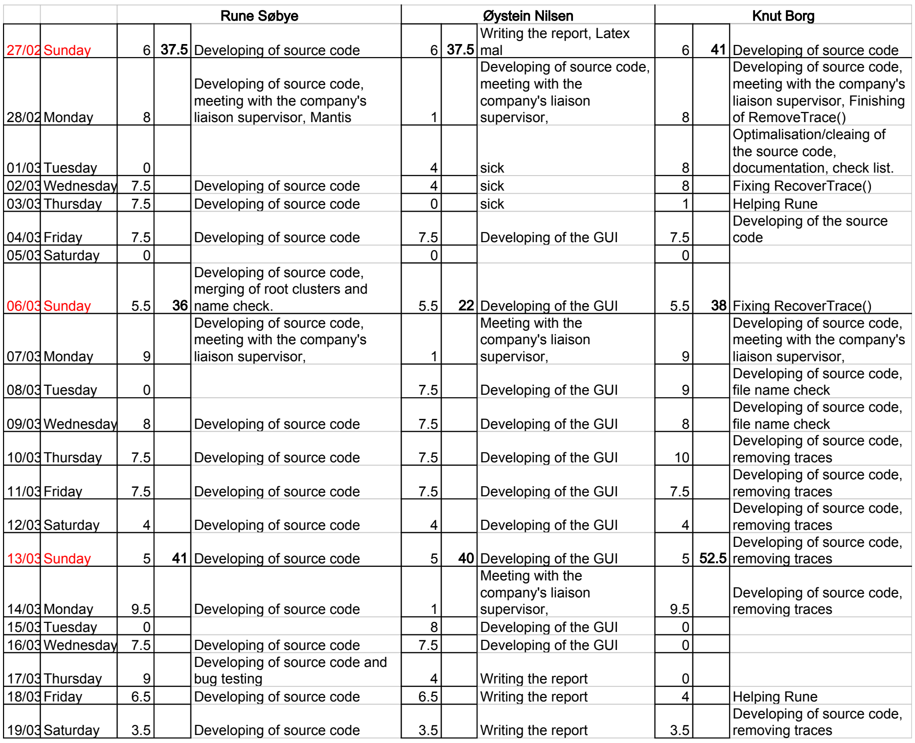
```
#run.ps1
for($i=0; $i -lt 400;$i++){
 echo $i
 if($i -gt 200){
./analyse.exe "\\.\F:" "l" "8 GB"
}
else{
./analyse.exe "\\.\F:" "s" "8 GB"

}
}
```

D Timesheets

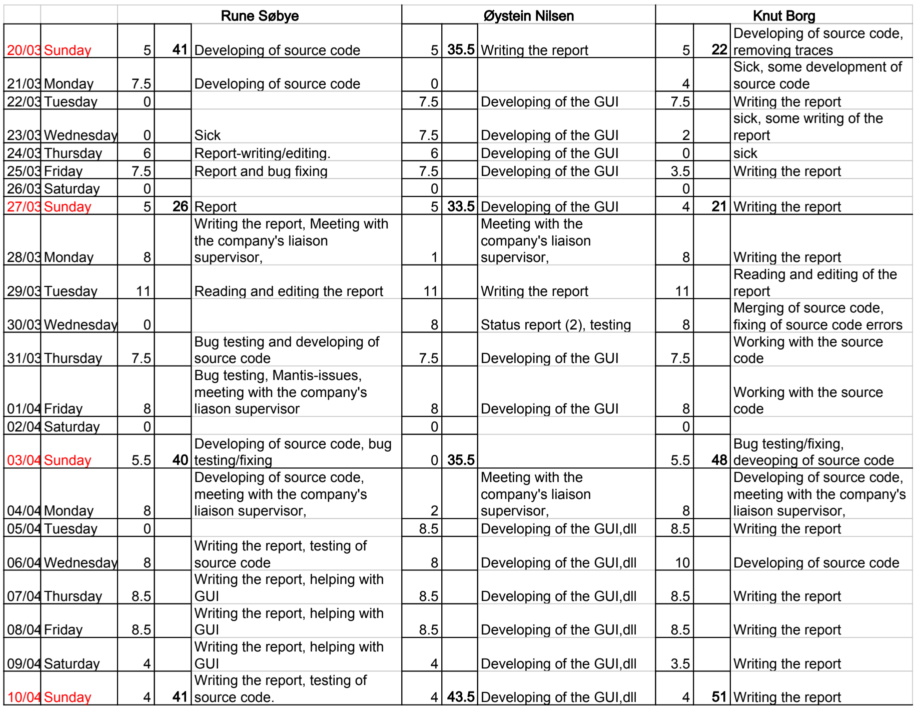


		Rune Søbye			Øystein Nilsen			Knut Borg		
Dato	Dag	Total hours	Sum week	Comments	Total hours	Sum week	Comments	Total hours	Sum week	Comments
09/01	Sunday	4		Analysing of proof of concept and assignments	4		Analysing of proof of concept and assignements	4		Analysing of proof of concept and assignements
10/01	Monday	7		Analysing of proof of concept and assignments	1		Analysing of proof of concept and assignements	7		Analysing of proof of concept and assignements
11/01	Tuesday	1		Meeting. Getting a overview of the assignments before next meeting.	7		Meeting	7		Meeting, discussion about sub-repository, writing to a memory stick succeed
12/01	Wednesday	7		Project plan, Presentation by T. Røise.	7		Usecase,presentation by T. Røise. Learning Latex.	7		Usecase,presentation by T. Røise. Learning Latex, time plan
13/01	Thursday	7.5		Requirement spec. Reading different contracts.	7.5		Fixing Latex	6		Requirement spec
14/01	Friday	6.5		Requirement spec. LateX.	6.5		Requirement spec Latex	2		Requirement spec, Latex
15/01	Saturday	0			0			0		
16/01	Sunday	3.5	**32.5**	Requirement spec. LaTeX.	3.5	**32.5**	meeting with the company's liaison supervisor,	3.5	**32.5**	Usecase,presentation by T. Røise. Learning Latex.
17/01	Monday	7.5		Group contract, reading the FAT system specification. Meeting with Hanno.	0.5		Meeting with the company's liaison supervisor,	7.5		Meeting with the company's liaison supervisor,
18/01	Tuesday	0			7.5		Studing the FAT32 file system, home page	7.5		Created 2 graphical prototypes, Fixed network drives, homepage and reading the FAT32 specification
19/01	Wednesday	7.5		Group contract, Repository. Project plan (limitations).	7.5		Repository, web site, project plan.	7.5		Project plan, group contract, repository software, requirement spec
20/01	Thursday	7.5		Project plan	7.5		Web site, project plan	5		Project plan
21/01	Friday	7.5		Project plan	7.5		Project plan	3.5		Project plan
22/01	Saturday	0			0			0		
23/01	Sunday	**5.5**	**35.5**		**5.5**	**36**	Project plan	**6.25**	**37.25**	Project plan
24/01	Monday	8.5		Gantt-scheme, project plan.	2		Project plan	8.5		Project plan, home page, logo.
25/01	Tuesday	8.5		Project plan	9.5		Project plan	9.5		Project plan
26/01	Wednesday	8		Project plan	8		Project plan, Latex	7.5		Project plan
27/01	Thursday	7		Project plan	7		Project plan	5		Project plan
28/01	Friday	7.5		Project plan. Presentation by K. Franke	7.5		Project plan. Presentation by K. Franke	3.5		Project plan. Presentation by K. Franke
29/01	Saturday	0			0			0		
30/01	Sunday	0	**39.5**		0	**34**		0	**34**	

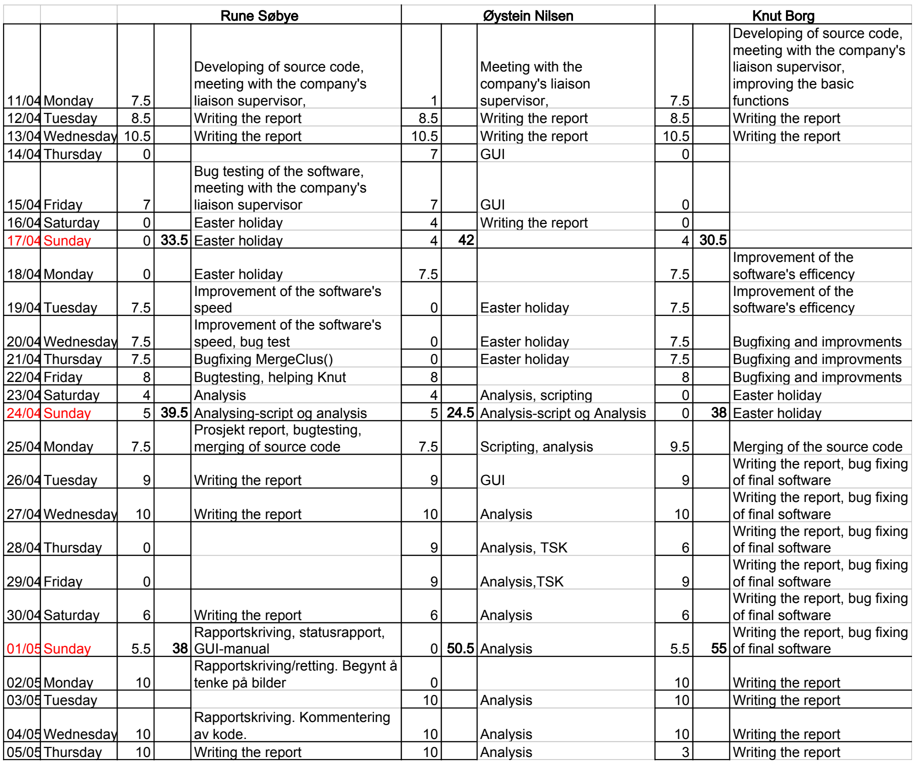
		Rune Søbye			Øystein Nilsen			Knut Borg		
31/01	Monday	7.5		Reading documetation, meeting	0.5		Meeting	7.5		FAT32 specification, meeting.
01/02	Tuesday	0			8		Developing of source code	8		Developing of source code, testing of parameteres, reading the FAT file system specification
02/02	Wednesday	7.5		Reading documetation	8		Developing of source code	7.5		Developing of source code, testing of parameteres, reading the FAT file system specification
03/02	Thursday	8		Developing of source code	8		Developing of source code	3.5		Nothing special in particular, small fixes.
04/02	Friday	6		Developing of source code	6		Developing of source code	3.5		Developing of source code
05/02	Saturday	0			0			0		
06/02	Sunday	0	**29**		0	**30.5**		0	**30**	
07/02	Monday	11		Back-on-track-meeting, developing of source code	5		Back-on-track-meeting, developing of source code	11		Back-on-track-meeting, developing of source code
08/02	Tuesday	8		Developing of source code, meeting with supervisor liasor and employer.	9		Developing of source code, meeting with supervisor liasor and employer.	9		Developing of source code, meeting with supervisor liasor and employer.
09/02	Wednesday	10		Developing of source code	10		Developing of source code	10		Developing of source code
10/02	Thursday	8		Developing of source code	8		Developing of source code	6		Developing of source code and home page
11/02	Friday	7.5		Developing of source code	7.5		Developing of source code	5.5		Developing of source code
12/02	Saturday	0			0			0		
13/02	Sunday	4	**48.5**	Beginning to write the report, developing source code	4	**43.5**	Report	4	**45.5**	Developing of source code
14/02	Monday	7.5		Developing of source code	0			7.5		Developing of source code
15/02	Tuesday	0			7.5		Developing of source code	7.5		Developing of source code
16/02	Wednesday	7.5		Developing of source code	7.5		Developing of source code	7.5		Developing of source code
17/02	Thursday	7.5		Developing of source code	7.5		Developing of source code	5.5		Developing of source code
18/02	Friday	7.5		Developing of source code	7.5		Developing of source code	4.5		Developing of source code
19/02	Saturday	0			0			0		
20/02	Sunday	0	**30**		0	**30**		0	**32.5**	
21/02	Monday	7.5		Developing of source code	0			7.5		Developing of source code
22/02	Tuesday	0			7.5		Status report (1)	7.5		Koding
23/02	Wednesday	7.5		Developing of source code	7.5		Writing the report	7.5		Koding
24/02	Thursday	9		Developing of source code	9		Writing the report	7		Koding
25/02	Friday	7.5		Developing of source code and meeting with supervisor	7.5		Report and meeting with supervisor	5.5		Developing of source code and meeting with supervisor
26/02	Saturday	0			0			0		



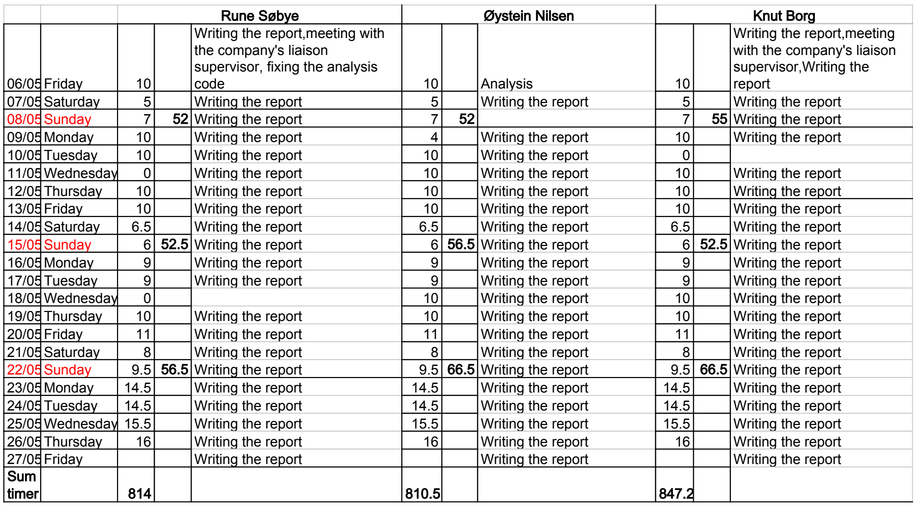
		Rune Søbye			Øystein Nilsen			Knut Borg		
27/02	Sunday	6	37.5	Developing of source code	6	37.5	Writing the report, Latex mal	6	41	Developing of source code
28/02	Monday	8		Developing of source code, meeting with the company's liaison supervisor, Mantis	1		Developing of source code, meeting with the company's liaison supervisor,	8		Developing of source code, meeting with the company's liaison supervisor, Finishing of RemoveTrace()
01/03	Tuesday	0			4		sick	8		Optimalisation/cleaing of the source code, documentation, check list.
02/03	Wednesday	7.5		Developing of source code	4		sick	8		Fixing RecoverTrace()
03/03	Thursday	7.5		Developing of source code	0		sick	1		Helping Rune
04/03	Friday	7.5		Developing of source code	7.5		Developing of the GUI	7.5		Developing of the source code
05/03	Saturday	0			0			0		
06/03	Sunday	5.5	36	Developing of source code, merging of root clusters and name check.	5.5	22	Developing of the GUI	5.5	38	Fixing RecoverTrace()
07/03	Monday	9		Developing of source code, meeting with the company's liaison supervisor,	1		Meeting with the company's liaison supervisor,	9		Developing of source code, meeting with the company's liaison supervisor,
08/03	Tuesday	0			7.5		Developing of the GUI	9		Developing of source code, file name check
09/03	Wednesday	8		Developing of source code	7.5		Developing of the GUI	8		Developing of source code, file name check
10/03	Thursday	7.5		Developing of source code	7.5		Developing of the GUI	10		Developing of source code, removing traces
11/03	Friday	7.5		Developing of source code	7.5		Developing of the GUI	7.5		Developing of source code, removing traces
12/03	Saturday	4		Developing of source code	4		Developing of the GUI	4		Developing of source code, removing traces
13/03	Sunday	5	41	Developing of source code	5	40	Developing of the GUI	5	52.5	Developing of source code, removing traces
14/03	Monday	9.5		Developing of source code	1		Meeting with the company's liaison supervisor,	9.5		Developing of source code, removing traces
15/03	Tuesday	0			8		Developing of the GUI	0		
16/03	Wednesday	7.5		Developing of source code	7.5		Developing of the GUI	0		
17/03	Thursday	9		Developing of source code and bug testing	4		Writing the report	0		
18/03	Friday	6.5		Developing of source code	6.5		Writing the report	4		Helping Rune
19/03	Saturday	3.5		Developing of source code	3.5		Writing the report	3.5		Developing of source code, removing traces



		Rune Søbye			Øystein Nilsen			Knut Borg		
20/03	Sunday	5	41	Developing of source code	5	35.5	Writing the report	5	22	Developing of source code, removing traces
21/03	Monday	7.5		Developing of source code	0			4		Sick, some development of source code
22/03	Tuesday	0			7.5		Developing of the GUI	7.5		Writing the report
23/03	Wednesday	0		Sick	7.5		Developing of the GUI	2		sick, some writing of the report
24/03	Thursday	6		Report-writing/editing.	6		Developing of the GUI	0		sick
25/03	Friday	7.5		Report and bug fixing	7.5		Developing of the GUI	3.5		Writing the report
26/03	Saturday	0			0			0		
27/03	Sunday	5	26	Report	5	33.5	Developing of the GUI	4	21	Writing the report
28/03	Monday	8		Writing the report, Meeting with the company's liaison supervisor,	1		Meeting with the company's liaison supervisor,	8		Writing the report
29/03	Tuesday	11		Reading and editing the report	11		Writing the report	11		Reading and editing of the report
30/03	Wednesday	0			8		Status report (2), testing	8		Merging of source code, fixing of source code errors
31/03	Thursday	7.5		Bug testing and developing of source code	7.5		Developing of the GUI	7.5		Working with the source code
01/04	Friday	8		Bug testing, Mantis-issues, meeting with the company's liason supervisor	8		Developing of the GUI	8		Working with the source code
02/04	Saturday	0			0			0		
03/04	Sunday	5.5	40	Developing of source code, bug testing/fixing	0	35.5		5.5	48	Bug testing/fixing, deveoping of source code
04/04	Monday	8		Developing of source code, meeting with the company's liaison supervisor,	2		Meeting with the company's liaison supervisor,	8		Developing of source code, meeting with the company's liaison supervisor,
05/04	Tuesday	0			8.5		Developing of the GUI,dll	8.5		Writing the report
06/04	Wednesday	8		Writing the report, testing of source code	8		Developing of the GUI,dll	10		Developing of source code
07/04	Thursday	8.5		Writing the report, helping with GUI	8.5		Developing of the GUI,dll	8.5		Writing the report
08/04	Friday	8.5		Writing the report, helping with GUI	8.5		Developing of the GUI,dll	8.5		Writing the report
09/04	Saturday	4		Writing the report, helping with GUI	4		Developing of the GUI,dll	3.5		Writing the report
10/04	Sunday	4	41	Writing the report, testing of source code.	4	43.5	Developing of the GUI,dll	4	51	Writing the report

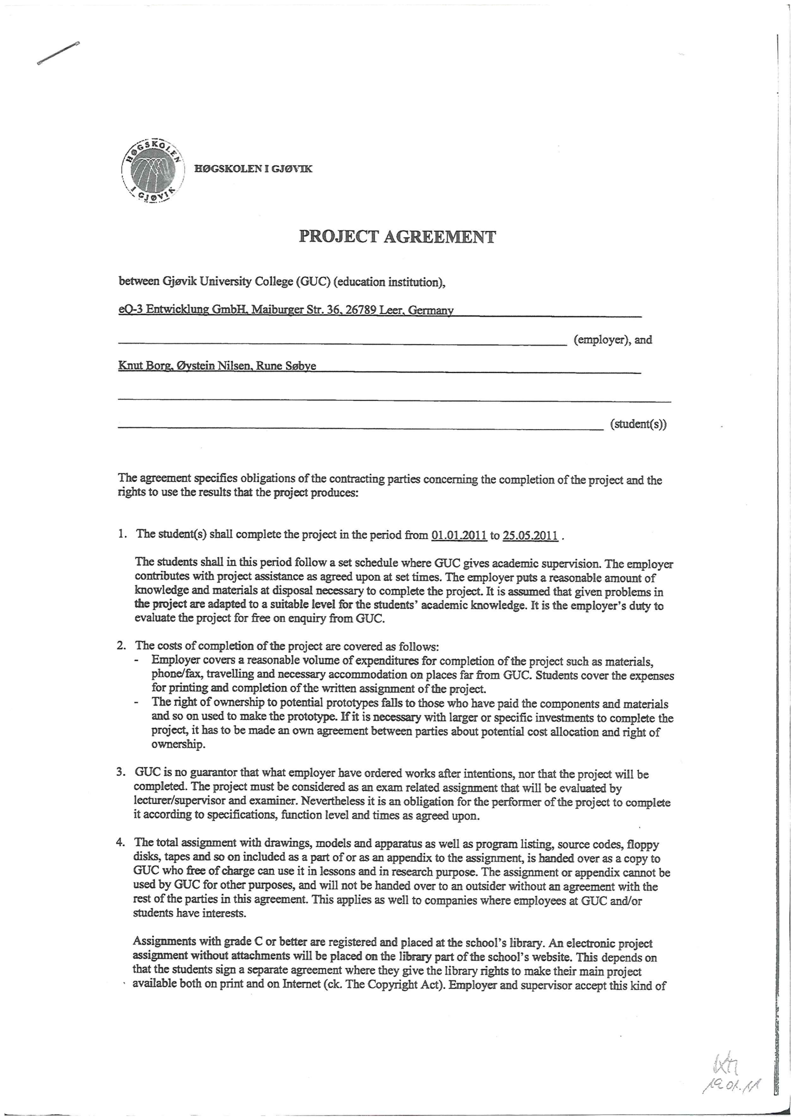


		Rune Søbye			Øystein Nilsen			Knut Borg		
11/04	Monday	7.5		Developing of source code, meeting with the company's liaison supervisor,	1		Meeting with the company's liaison supervisor,	7.5		Developing of source code, meeting with the company's liaison supervisor, improving the basic functions
12/04	Tuesday	8.5		Writing the report	8.5		Writing the report	8.5		Writing the report
13/04	Wednesday	10.5		Writing the report	10.5		Writing the report	10.5		Writing the report
14/04	Thursday	0			7		GUI	0		
15/04	Friday	7		Bug testing of the software, meeting with the company's liaison supervisor	7		GUI	0		
16/04	Saturday	0		Easter holiday	4		Writing the report	0		
17/04	Sunday	0	**33.5**	Easter holiday	4	**42**		4	**30.5**	
18/04	Monday	0		Easter holiday	7.5			7.5		Improvement of the software's efficency
19/04	Tuesday	7.5		Improvement of the software's speed	0		Easter holiday	7.5		Improvement of the software's efficency
20/04	Wednesday	7.5		Improvement of the software's speed, bug test	0		Easter holiday	7.5		Bugfixing and improvments
21/04	Thursday	7.5		Bugfixing MergeClus()	0		Easter holiday	7.5		Bugfixing and improvments
22/04	Friday	8		Bugtesting, helping Knut	8			8		Bugfixing and improvments
23/04	Saturday	4		Analysis	4		Analysis, scripting	0		Easter holiday
24/04	Sunday	5	**39.5**	Analysing-script og analysis	5	**24.5**	Analysis-script og Analysis	0	**38**	Easter holiday
25/04	Monday	7.5		Prosjekt report, bugtesting, merging of source code	7.5		Scripting, analysis	9.5		Merging of the source code
26/04	Tuesday	9		Writing the report	9		GUI	9		Writing the report, bug fixing of final software
27/04	Wednesday	10		Writing the report	10		Analysis	10		Writing the report, bug fixing of final software
28/04	Thursday	0			9		Analysis, TSK	6		Writing the report, bug fixing of final software
29/04	Friday	0			9		Analysis,TSK	9		Writing the report, bug fixing of final software
30/04	Saturday	6		Writing the report	6		Analysis	6		Writing the report, bug fixing of final software
01/05	Sunday	5.5	**38**	Rapportskriving, statusrapport, GUI-manual	0	**50.5**	Analysis	5.5	**55**	Writing the report, bug fixing of final software
02/05	Monday	10		Rapportskriving/retting. Begynt å tenke på bilder	0			10		Writing the report
03/05	Tuesday				10		Analysis	10		Writing the report
04/05	Wednesday	10		Rapportskriving. Kommentering av kode.	10		Analysis	10		Writing the report
05/05	Thursday	10		Writing the report	10		Analysis	3		Writing the report



		Rune Søbye			**Øystein Nilsen**			**Knut Borg**		
06/05	Friday	10		Writing the report,meeting with the company's liaison supervisor, fixing the analysis code	10		Analysis	10		Writing the report,meeting with the company's liaison supervisor,Writing the report
07/05	Saturday	5		Writing the report	5		Writing the report	5		Writing the report
08/05	Sunday	7	**52**	Writing the report	7	**52**		7	**55**	Writing the report
09/05	Monday	10		Writing the report	4		Writing the report	10		Writing the report
10/05	Tuesday	10		Writing the report	10		Writing the report	0		
11/05	Wednesday	0		Writing the report	10		Writing the report	10		Writing the report
12/05	Thursday	10		Writing the report	10		Writing the report	10		Writing the report
13/05	Friday	10		Writing the report	10		Writing the report	10		Writing the report
14/05	Saturday	6.5		Writing the report	6.5		Writing the report	6.5		Writing the report
15/05	Sunday	6	**52.5**	Writing the report	6	**56.5**	Writing the report	6	**52.5**	Writing the report
16/05	Monday	9		Writing the report	9		Writing the report	9		Writing the report
17/05	Tuesday	9		Writing the report	9		Writing the report	9		Writing the report
18/05	Wednesday	0			10		Writing the report	10		Writing the report
19/05	Thursday	10		Writing the report	10		Writing the report	10		Writing the report
20/05	Friday	11		Writing the report	11		Writing the report	11		Writing the report
21/05	Saturday	8		Writing the report	8		Writing the report	8		Writing the report
22/05	Sunday	9.5	**56.5**	Writing the report	9.5	**66.5**	Writing the report	9.5	**66.5**	Writing the report
23/05	Monday	14.5		Writing the report	14.5		Writing the report	14.5		Writing the report
24/05	Tuesday	14.5		Writing the report	14.5		Writing the report	14.5		Writing the report
25/05	Wednesday	15.5		Writing the report	15.5		Writing the report	15.5		Writing the report
26/05	Thursday	16		Writing the report	16		Writing the report	16		Writing the report
27/05	Friday			Writing the report			Writing the report			Writing the report
Sum timer		**814**			**810.5**			**847.2**		

E Project agreement



HØGSKOLEN I GJØVIK

PROJECT AGREEMENT

between Gjøvik University College (GUC) (education institution),

eO-3 Entwicklung GmbH, Maiburger Str. 36, 26789 Leer, Germany

____________________ (employer), and

Knut Borg, Øystein Nilsen, Rune Søbye

____

____________________ (student(s))

The agreement specifies obligations of the contracting parties concerning the completion of the project and the rights to use the results that the project produces:

1. The student(s) shall complete the project in the period from 01.01.2011 to 25.05.2011 .

   The students shall in this period follow a set schedule where GUC gives academic supervision. The employer contributes with project assistance as agreed upon at set times. The employer puts a reasonable amount of knowledge and materials at disposal necessary to complete the project. It is assumed that given problems in the project are adapted to a suitable level for the students' academic knowledge. It is the employer's duty to evaluate the project for free on enquiry from GUC.

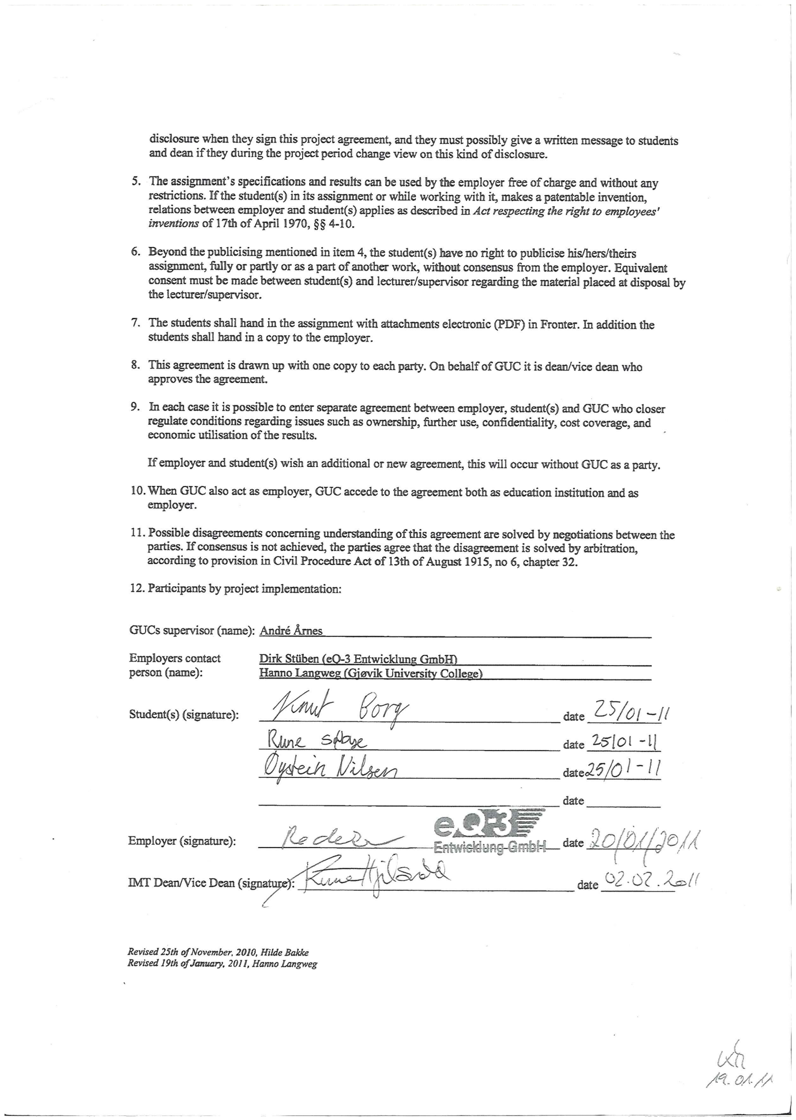
2. The costs of completion of the project are covered as follows:
   - Employer covers a reasonable volume of expenditures for completion of the project such as materials, phone/fax, travelling and necessary accommodation on places far from GUC. Students cover the expenses for printing and completion of the written assignment of the project.
   - The right of ownership to potential prototypes falls to those who have paid the components and materials and so on used to make the prototype. If it is necessary with larger or specific investments to complete the project, it has to be made an own agreement between parties about potential cost allocation and right of ownership.

3. GUC is no guarantor that what employer have ordered works after intentions, nor that the project will be completed. The project must be considered as an exam related assignment that will be evaluated by lecturer/supervisor and examiner. Nevertheless it is an obligation for the performer of the project to complete it according to specifications, function level and times as agreed upon.

4. The total assignment with drawings, models and apparatus as well as program listing, source codes, floppy disks, tapes and so on included as a part of or as an appendix to the assignment, is handed over as a copy to GUC who free of charge can use it in lessons and in research purpose. The assignment or appendix cannot be used by GUC for other purposes, and will not be handed over to an outsider without an agreement with the rest of the parties in this agreement. This applies as well to companies where employees at GUC and/or students have interests.

   Assignments with grade C or better are registered and placed at the school's library. An electronic project assignment without attachments will be placed on the library part of the school's website. This depends on that the students sign a separate agreement where they give the library rights to make their main project available both on print and on Internet (ck. The Copyright Act). Employer and supervisor accept this kind of

19.01.11



disclosure when they sign this project agreement, and they must possibly give a written message to students and dean if they during the project period change view on this kind of disclosure.

5. The assignment's specifications and results can be used by the employer free of charge and without any restrictions. If the student(s) in its assignment or while working with it, makes a patentable invention, relations between employer and student(s) applies as described in *Act respecting the right to employees' inventions* of 17th of April 1970, §§ 4-10.

6. Beyond the publicising mentioned in item 4, the student(s) have no right to publicise his/hers/theirs assignment, fully or partly or as a part of another work, without consensus from the employer. Equivalent consent must be made between student(s) and lecturer/supervisor regarding the material placed at disposal by the lecturer/supervisor.

7. The students shall hand in the assignment with attachments electronic (PDF) in Fronter. In addition the students shall hand in a copy to the employer.

8. This agreement is drawn up with one copy to each party. On behalf of GUC it is dean/vice dean who approves the agreement.

9. In each case it is possible to enter separate agreement between employer, student(s) and GUC who closer regulate conditions regarding issues such as ownership, further use, confidentiality, cost coverage, and economic utilisation of the results.

   If employer and student(s) wish an additional or new agreement, this will occur without GUC as a party.

10. When GUC also act as employer, GUC accede to the agreement both as education institution and as employer.

11. Possible disagreements concerning understanding of this agreement are solved by negotiations between the parties. If consensus is not achieved, the parties agree that the disagreement is solved by arbitration, according to provision in Civil Procedure Act of 13th of August 1915, no 6, chapter 32.

12. Participants by project implementation:

GUCs supervisor (name): André Årnes

Employers contact person (name): Dirk Stüben (eQ-3 Entwicklung GmbH)
Hanno Langweg (Gjøvik University College)

Student(s) (signature): Knut Borg date 25/01-11
Rune Støge date 25/01-11
Øystein Nilsen date 25/01-11
date

Employer (signature): Redeker eQ3 Entwicklung GmbH date 20/01/2011

IMT Dean/Vice Dean (signature): Rune Hjelsvold date 02.02.2011

Revised 25th of November, 2010, Hilde Bakke
Revised 19th of January, 2011, Hanno Langweg

19.01.11

F Status reports

Status report 1

Inexpensive Data Hiding on USB Memory Sticks

Knut Borg, Øystein Nilsen, Rune Søbye.

What has been done:

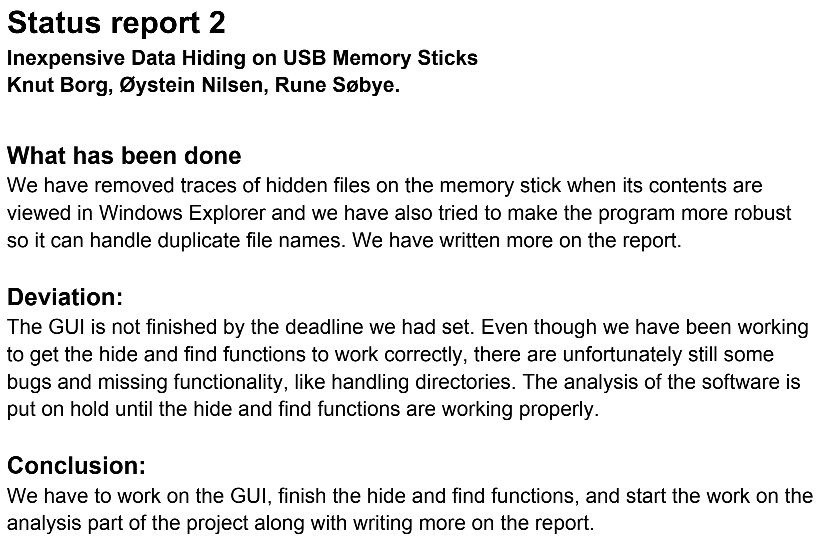
The project plan was finished within the deadline. The webpage was up early. Started with the coding part of the assignment after delivery of the project plan. After a difficult start we managed to write a program that is able to hide and recover files, but it is in no way robust, and has not been tested in all possible scenarios yet. We have however tested to see if you can find a hidden file on Linux (Ubuntu), and the result is that you can't.

Deviation:

There is still no solution about the practicalities of the quality assurance part of the project. The bug tracking system (Mantis) was set up a bit late. The thesis itself also needs more work. The status report is a bit late, due to not checking the project plan. We have done the coding in C instead of C++, since the proof of concept code that we're working from was in C. C is not very different from C++ however, and we have some experience in C from other courses. Since we don't have anything to analyze yet, the analysis part of the project is put on hold.

Conclusion:

Some parts of the assignment need more work, like writing on the thesis. The coding of the program is not behind schedule. Need to check our project plan more often to avoid breaking deadlines.



Status report 2

Inexpensive Data Hiding on USB Memory Sticks
Knut Borg, Øystein Nilsen, Rune Søbye.

What has been done

We have removed traces of hidden files on the memory stick when its contents are viewed in Windows Explorer and we have also tried to make the program more robust so it can handle duplicate file names. We have written more on the report.

Deviation:

The GUI is not finished by the deadline we had set. Even though we have been working to get the hide and find functions to work correctly, there are unfortunately still some bugs and missing functionality, like handling directories. The analysis of the software is put on hold until the hide and find functions are working properly.

Conclusion:

We have to work on the GUI, finish the hide and find functions, and start the work on the analysis part of the project along with writing more on the report.

G Project plan

Project Plan

Knut Borg, Øystein Nilsen, Rune Søbye
IMT
HIG, Gjøvik

January 28, 2011

Contents

1 Project plan

1.1 About

This project plan contains information about how the group will work with the bachelor project towards the 25th of May. This includes, amongst others, effect and result goals, development method and a Gantt chart for what we will work with and when we are planning to do it.

1.2 Introduction

The use of USB (Universal Serial Bus) memory sticks has in the recent years become more popular for regular users. The main reason for this is increased storage capacity, while the price has decreased. Also, with increased popularity, there are a lot of new products which utilize the functionality of memory sticks, for instance digital photo frames.

One downside with memory sticks is that their contents may be easily captured by others than the owner of the memory stick. This may for instance be in cases where the memory stick is stolen, when the owner lends it to another person or if the memory stick is unattended. In this project, we are going to make a simple protection mechanism which should be as lightweight as possible. The main idea is to hide the contents on a memory stick, making the owner less vulnerable of getting his contents copied, read, deleted or edited by another person.

1.3 Goals and framework

1.3.1 Project goals

The goal of this project is to hide data stored on a USB memory stick without the need of modifying the device's hardware or firmware and without using time-expensive encryption. The software must also be user friendly, making it possible to use for all kinds of users.

1.3.2 Effect goals

Effect goals are the goals we believe might be the result of our finished product:

- A well made product will lead to good publicity to the employer, because the product reflects how the company handles quality assurance. Also, a good product itself reflects the quality of the company's products as a whole.
- The customers' data will be better protected against people who do not invest a huge amount of time and effort to investigate the memory stick for hidden data.

1.3.3 Result goals

Result goals represents what should be accomplished at the end of the project:

- Make an intuitive user interface that everyone can use.
- Make a product that is of such quality that the company chooses to share it with their customers.
- Hide files in a quick and efficient manner.

1.3.4 Framework

The project has to be finished by 25th of May. We need some memory sticks and forensic tools for testing and analysis. The documents are written in LaTeX and source code in Microsoft (MS) Visual Studio 2008. We will program in C++ and use our own computers for development and testing of the software. The costs will be small, and probably not exceed 1000 NOK.

1.4 Scope

1.4.1 Task description

The main objective of the software is to protect contents on a memory stick. The idea is to hide contents to protect them from being altered, viewed or deleted by others than the actual owner of the memory stick. In order to do so, we need to figure out a way to hide and recover files on the memory stick. Everyone should be able to use the software regardless of their knowledge of computers, and because of that we have to make an user friendly GUI (Graphical User Interface). It is also important that the software does not consume a large amount of computer resources, while still succeeding its tasks within a reasonable amount of time.

In addition to creating the software, we are also going to make an analysis of the hidden contents. This means that we must study the hidden files in detail and then compare them to how they appear when they are not hidden. By doing so we might be able to conclude in which degree it is obvious that files are hidden, in which manner the hidden files are being represented and how strong the protection of the files is. Also, a comparison with a software using cryptographic methods when hiding files should be done. In this way we might be able to draw a conclusion on how much faster the process of hiding and recovering files is when no expensive cryptographic methods are involved.

1.4.2 Requirements

The main requirements for the software are as follows:

- It must be reliable. It should work every time, as well as being able to handle unexpected situations.
- It has to be able to hide files and recover files hidden by the software on the FAT32 file system.
- The software must be able to be used without any further preparations than installing it and being in possession of a memory stick.
- It must be user friendly. It should be easy and intuitive for all kinds of users to hide files and later recover them.
- When an illegitimate user tries to access the memory stick, he should not be able to discover hidden contents or suspect that files are being hidden. This is only the case if the user is not in possession of any advanced forensic tools or skills regarding the FAT32 file system.
- The memory stick must still be possible to use when files are being hidden.
- It is important that the user does not need to wait for a long time in order to hide or recover files, so the process of doing so must be inexpensive. Being inexpensive means there will be no use of cryptographic algorithms either by software or hardware implementations, making the process of hiding or recovering files able to finish in a short amount of time which equals no longer than 5 to 6 seconds.
- It should be impossible to detect the hidden files when using computers running either Windows, Mac OS X or various Linux distributions.

1.4.3 Limitations

There is not enough time to consider every option and eventuality, so we need to set some limitations:

- The software will only be developed for MS Windows XP, Vista and 7.
- The possibility of hiding only a range of files might be excluded due to complexity. Our primary goal is to hide or recover all files on a memory stick in a single operation. If we are able to complete that goal in a reasonable amount of time, we will also try to implement the option of hiding or recovering a range of files, decided by the user.
- We are limiting the software to only hide and recover files on the FAT32 file system.

1.5 Project organization

1.5.1 Responsibilities and roles

The group consists of Knut Borg, Rune Søbye and Øystein Nilsen, where Knut Borg is the group leader. The employer is the German company eQ-3 Entwickung GmbH. Our contact persons are Hanno Langweg from GUC (Gjøvik University College) and Dirk Stüben from the company itself. The academic advisor is André Årnes from GUC.

1.5.2 Procedures and rules

All members are required to work at least 30 hours per week. Due to the members taking different classes, there will be an unique schedule for each member. Even though the students' availability may vary, they need to show up on time and stay as long as the schedule requires them to do. All of the group members are prepared to work more than 30 hours a week to finish a goal with the group's expectations of quality. A meeting protocol must be written for every meeting with the employer and the supervisor, so we are able to keep track of feedback and what the group is planning to do. All of the group members are required to notice the others if they feel that they are not able to complete a certain goal within a given time frame. By doing so, the group is able to adjust the workload between the members accordingly, and stand a better chance of completing a given goal.

Check attachment A for more details.

1.6 Planning, monitoring and reporting

1.6.1 The main classification of the project

The project consists of two main parts. The first one is to develop the software with user friendly GUI, required to hide or recover files from a memory stick, whereas the second part is to analyze the file structure on a memory stick where files have been hidden and to compare our method of hiding and recover files with the use of cryptographic tools.

1.6.2 Choice of system development model

We want to use a system development model where we can change the system requirements during the development process, and change the source code if unexpected problems occur. The evolutionary model fits our project well because it will give us the flexibility to switch between specification, development and validation. Also, it makes it possible for for the employer to continuously review our work and then give us feedback. We will deliver several prototypes to the employer and each of the prototypes is an improvement of the previous.

The very first prototype will contain the functionality required to write data to a memory stick. It should also be possible to change file names, contents on the memory stick, etc. We decide what the prototypes should contain at the beginning of the working process, in collaboration with supervisor and/or Hanno Langweg. By the evolutionary model we will then improve our software through several iterations until we have a final version.

1.6.3 Plan for status meetings and decision points

Status meetings with the supervisor will be held at GUC or through Elluminate/Skype, once or twice a month. Minor consultations with the supervisor may take place when required, through a phone call or email.

Meetings with the employer will take place with their contact person at GUC every Monday. We will present our recent prototype and the contact person will provide feedback.

Any important decision point will be handled in accordance to the supervisor/employer.

1.7 Organization of quality assurance

1.7.1 Documentation of source code

All non-trivial lines of code must be commented. This must be done continuously as the code is written. With doing so, we are avoiding problems with remembering what non-commented code is supposed to mean. Also, a well commented code is important for others reading it. All documentation of code, and the code itself, has to be written in English. All code, including the comments, should be written within column 80 and need to be short and concise.

1.7.2 Documentation of the software

The software needs a documentation, serving as a manual for the user. It should contain explanation of the software in general and the various features in particular. The manual should be be handed in as an appendix to the project report and as an own help function in the software. In addition to documentation of the software itself, we also need to document any major software related decision made throughout the work. This may for instance explain why we decided to use one GUI rather than another GUI, and why we chose a specific method to solve a particular problem.

1.7.3 Documentation of the working process as whole

As for the software part, every major decision needs to be documented. This may for example be done by describing the various options we had, and then explain why the chosen were preferred. Timesheets, status reports and a worklog are written to document the progress.

1.7.4 Configuration management

To keep track over the work we are doing, like the bachelor thesis it self and the coding of the software, we are going to use Subversion. Subversion is an open source version control system [source: http://subversion.apache.org/], used to keep track of changes and backup of files.

For bug-tracking we will use Mantis, which is an open source web based bug tracking system [source: http://www.mantisbt.org/]. It will be used to get an overview over all bugs and to keep track of changes.

Continuous integration will be used during the development of the source code. This means that we will use our subversion repository with TortoiseSVN, continuously committing new changes and integrate them with the existing software.

1.7.5 Risk analysis

In the table below, we have identified some aspects we consider to be a potential problem or threat to our work. For each of the aspects, a rating between 1 and 10 for probability and consequence is given, as well as some measures on how to possibly deal with that problem. The risk is the probability multiplied by the consequence.

Problem	Probability	Consequence	Risk	Measure(s)
Not able to solve parts of the task.	3	8	24	Redefine the scope. Start with the easiest parts, in order to at least get them done.
Missing internal or external deadlines.	2	8	16	Plan well ahead. Work extra hours prior to the deadlines.
Major collaboration problems or disagreements within the group.	2	6	12	Write group rules. Communicate in a clear and concise manner. Discuss the problem within the group or together with the supervisor.
Not able to find the necessary documentation for the project.	3	4	12	Ask supervisor, teachers, etc. Search actively for documentation early in the process.
One or more members dropping out from the group.	1	9	9	Redefine the scope of the project.
Loss of written code or other written materials.	1	7	7	Backup. Subversion. Save work frequently.
Long period of illness or injury for one or more members.	1	6	6	Adjust the workload between the rest of the members. The ill member may be able to work from home.
Lack of communication between the group and the employer/supervisor.	2	3	6	Ask for regular meetings. Communicate in a clear and concise manner.

1.7.6 Comments about the various aspects

Not able to solve parts of the task
It should not be a problem to solve the minimum requirements, which is to hide and recover all files. Some additional features, such as hiding a range of files or authentication, might be a problem due to complexity or lack of time. The consequence of missing the minimum requirements will be a negative impact on our grade. Missing any additional parts may have a small impact on the final grade.

Missing internal or external deadlines
Not very probable because we plan well ahead and the members are prepared to work extra hours prior to deadlines. There are two different types of deadlines, internal and external. Internal deadlines are the group's own deadlines for when objectives should be completed etc. External deadlines are the GUC's own deadlines, for instance handing in the bachelor assignment. The consequence depends on which deadline, and the type of deadline, we missed. The internal ones might be rescheduled, whereas the external ones has to be complied.

Major collaboration problems or disagreements within the group
Problems and disagreements have always been solved in a friendly manner without the need of bringing in external guidance. The consequence is not that big, since discussions within the group or with the supervisor might eliminate or reduce the problem. However, the progress might get on hold if a major disagreement occur.

Not able to find the necessary documentation for the project
The FAT32 file system and GUI-programming are common, so finding the necessary documentation should not be a problem. The consequence of not finding any documentation is that the we might not be able to keep up with the workload originally planned, which might interfere with deadlines.

One or more members dropping out from the group
We consider this to be rather unlikely because the the group have cooperated in almost all courses at GUC so far and thereby know each other well. The consequence however, is serious since the remaining group members probably need to redefine the scope of the project.

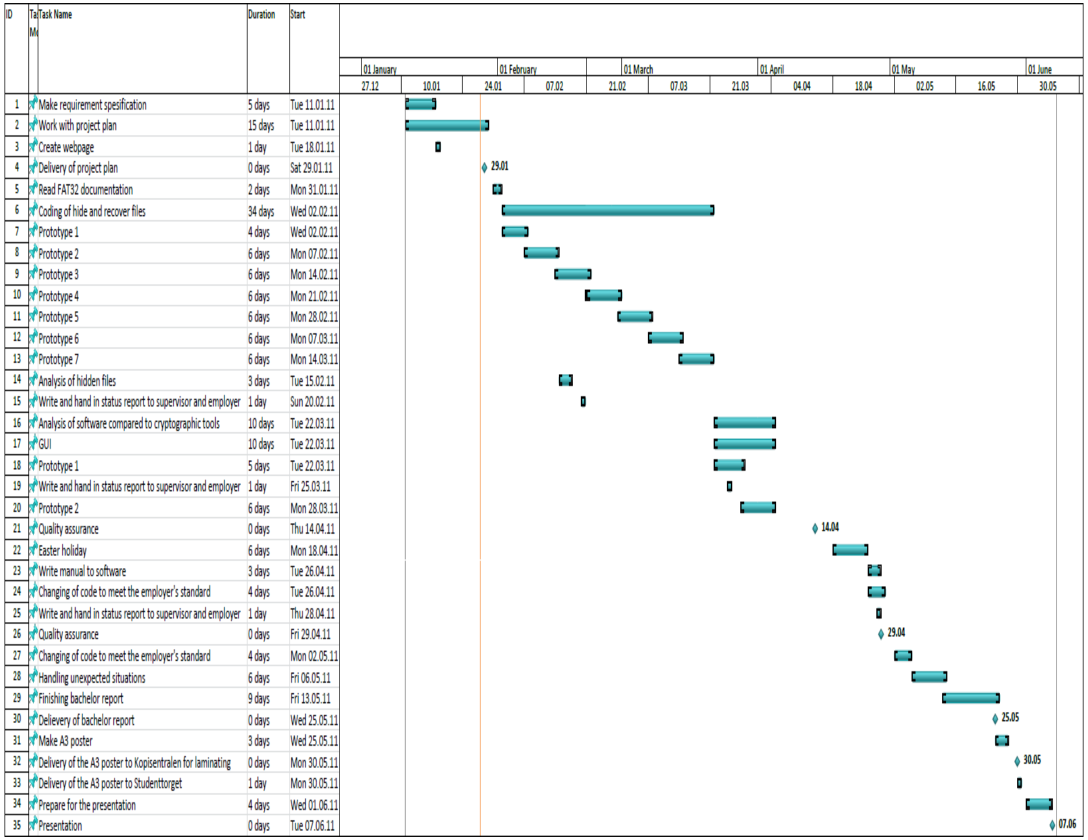
Loss of written code or other written materials
Very unlikely, because we are doing backup of our work. The most likely case will be to lose only a short time of work, for example due to the computer crashing. If we lose all or a majority of our work late in the working process, the consequence would be catastrophic. However, that probability is close to zero.

Long period of illness or injury for one or more members
We are considering this one to be rather unlikely. We have worked together as a group in almost every course at GUC, and so far, illness have not been a problem at all. If a group member gets ill and have to stay home for a longer period of time, then we will have to discuss the possibility of that given member to work from home. For worst case scenarios, see “One or more members dropping out from the group”.

Lack of communication between the group and the employer/supervisor
Lack of communication means that we do not communicate with the employer or supervisor as often as we should due to availability. The communication with the employer should not be a problem since one of their contact persons is working at GUC. The supervisor is not present at GUC, but well planned physical or electronical meetings should solve that problem.

1.8 Plan for implementation

1.8.1 Gantt chart



ID	Ta Mo	Task Name	Duration	Start
1		Make requirement spesification	5 days	Tue 11.01.11
2		Work with project plan	15 days	Tue 11.01.11
3		Create webpage	1 day	Tue 18.01.11
4		Delivery of project plan	0 days	Sat 29.01.11
5		Read FAT32 documentation	2 days	Mon 31.01.11
6		Coding of hide and recover files	34 days	Wed 02.02.11
7		Prototype 1	4 days	Wed 02.02.11
8		Prototype 2	6 days	Mon 07.02.11
9		Prototype 3	6 days	Mon 14.02.11
10		Prototype 4	6 days	Mon 21.02.11
11		Prototype 5	6 days	Mon 28.02.11
12		Prototype 6	6 days	Mon 07.03.11
13		Prototype 7	6 days	Mon 14.03.11
14		Analysis of hidden files	3 days	Tue 15.02.11
15		Write and hand in status report to supervisor and employer	1 day	Sun 20.02.11
16		Analysis of software compared to cryptographic tools	10 days	Tue 22.03.11
17		GUI	10 days	Tue 22.03.11
18		Prototype 1	5 days	Tue 22.03.11
19		Write and hand in status report to supervisor and employer	1 day	Fri 25.03.11
20		Prototype 2	6 days	Mon 28.03.11
21		Quality assurance	0 days	Thu 14.04.11
22		Easter holiday	6 days	Mon 18.04.11
23		Write manual to software	3 days	Tue 26.04.11
24		Changing of code to meet the employer's standard	4 days	Tue 26.04.11
25		Write and hand in status report to supervisor and employer	1 day	Thu 28.04.11
26		Quality assurance	0 days	Fri 29.04.11
27		Changing of code to meet the employer's standard	4 days	Mon 02.05.11
28		Handling unexpected situations	6 days	Fri 06.05.11
29		Finishing bachelor report	9 days	Fri 13.05.11
30		Delievery of bachelor report	0 days	Wed 25.05.11
31		Make A3 poster	3 days	Wed 25.05.11
32		Delivery of the A3 poster to Kopisentralen for laminating	0 days	Mon 30.05.11
33		Delivery of the A3 poster to Studenttorget	1 day	Mon 30.05.11
34		Prepare for the presentation	4 days	Wed 01.06.11
35		Presentation	0 days	Tue 07.06.11

12

1.8.2 Comments to the Gantt chart

Coding of hide and recover files
The coding of hide and recover files are divided into one separate period. We are planning to make 7 prototypes during that period. Each of them should include new features and be an improvement of the previous prototype. The various prototypes will be presented to our employers contact person at GUC and sent to our supervisor.

Analysis of software compared to cryptographic tools
We will analyze how our method of hiding files compares to encrypting them considering time used.

Quality assurance
We will send our source code to the employer who will test our software. They will then check if the software is within the company's guidelines.

Changeing of code to meet the employer's standard
After we receive the source code back from the first quality assurance, we will probably have to change our source code because no one creates an error free software at the first attempt.

Handling unexpected situations
If any unexpected events happens, we will use two weeks as a buffer to handle those.

1.9 Milestones

29.01: Delivery of project plan.

14.04: Quality assurance.

25.05: Delivery of bachelor report.

30.05: Delivery of the A3 poster to Kopisentralen for laminating.

07.06: Presentation.

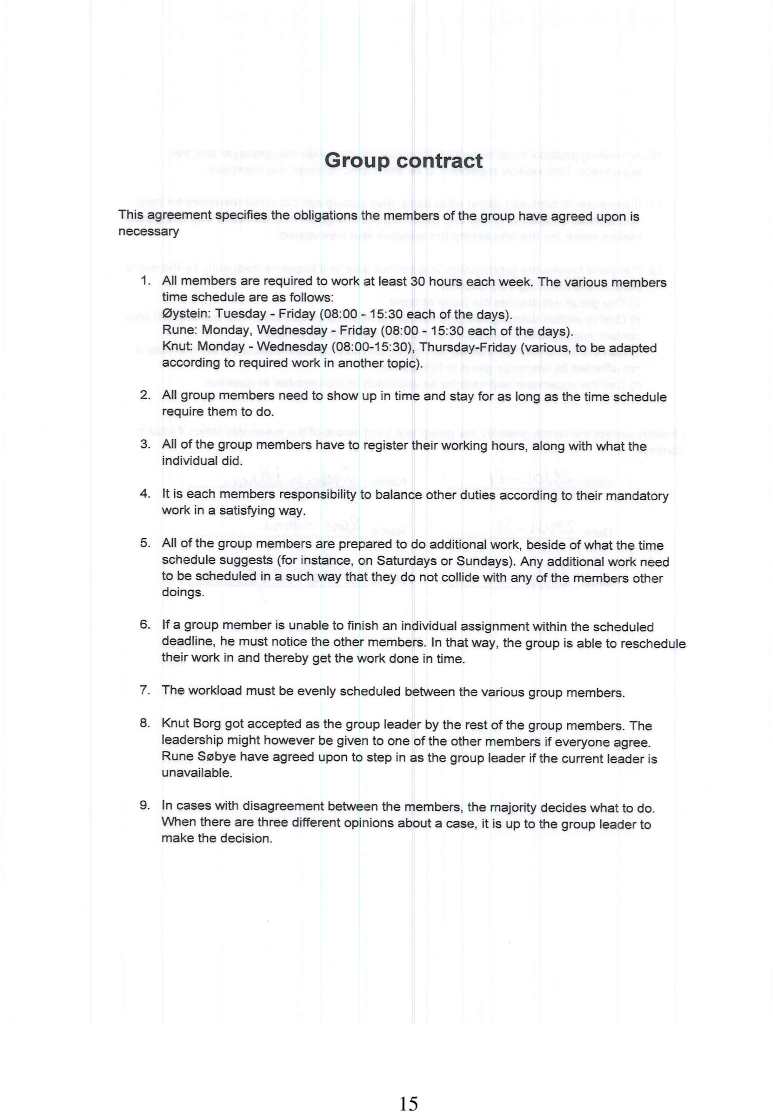
1.10 Hand-ins

20.02: Status report 1

25.03: Status report 2

28.04: Status report 3

A Group contract



Group contract

This agreement specifies the obligations the members of the group have agreed upon is necessary

1. All members are required to work at least 30 hours each week. The various members time schedule are as follows:
   Øystein: Tuesday - Friday (08:00 - 15:30 each of the days).
   Rune: Monday, Wednesday - Friday (08:00 - 15:30 each of the days).
   Knut: Monday - Wednesday (08:00-15:30), Thursday-Friday (various, to be adapted according to required work in another topic).

2. All group members need to show up in time and stay for as long as the time schedule require them to do.

3. All of the group members have to register their working hours, along with what the individual did.

4. It is each members responsibility to balance other duties according to their mandatory work in a satisfying way.

5. All of the group members are prepared to do additional work, beside of what the time schedule suggests (for instance, on Saturdays or Sundays). Any additional work need to be scheduled in a such way that they do not collide with any of the members other doings.

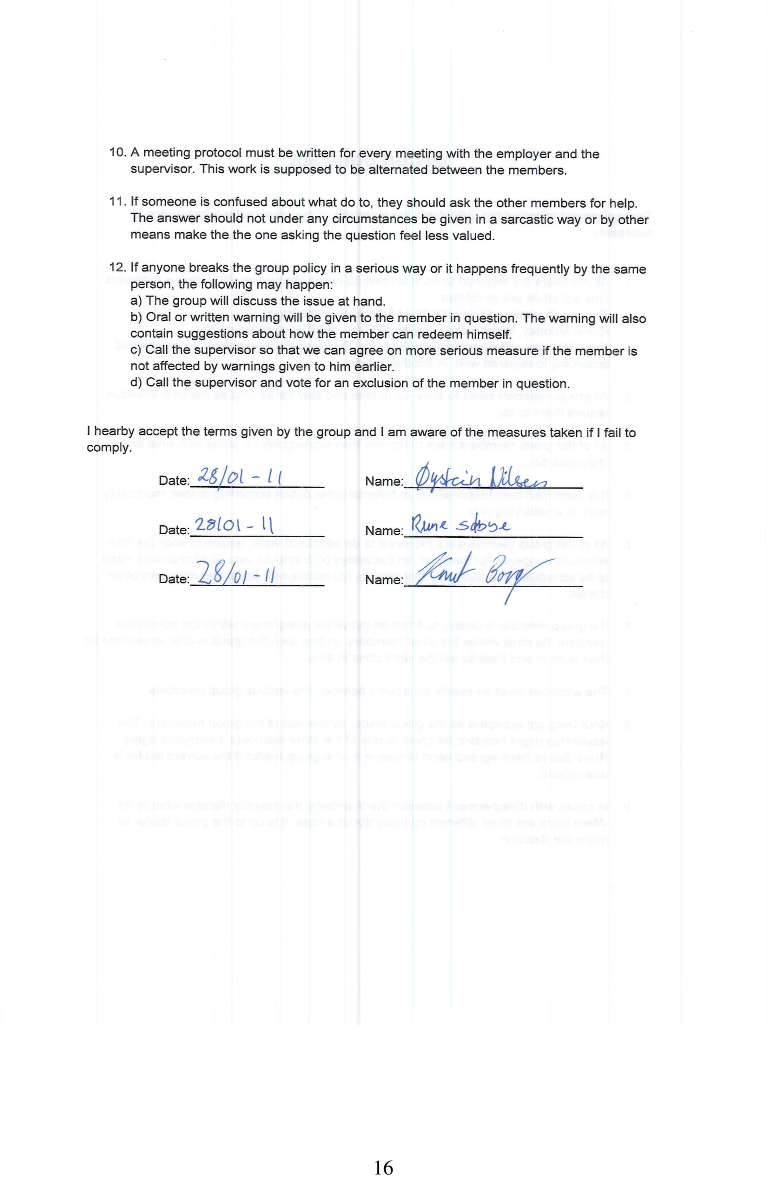
6. If a group member is unable to finish an individual assignment within the scheduled deadline, he must notice the other members. In that way, the group is able to reschedule their work in and thereby get the work done in time.

7. The workload must be evenly scheduled between the various group members.

8. Knut Borg got accepted as the group leader by the rest of the group members. The leadership might however be given to one of the other members if everyone agree. Rune Søbye have agreed upon to step in as the group leader if the current leader is unavailable.

9. In cases with disagreement between the members, the majority decides what to do. When there are three different opinions about a case, it is up to the group leader to make the decision.

15



10. A meeting protocol must be written for every meeting with the employer and the supervisor. This work is supposed to be alternated between the members.

11. If someone is confused about what do to, they should ask the other members for help. The answer should not under any circumstances be given in a sarcastic way or by other means make the the one asking the question feel less valued.

12. If anyone breaks the group policy in a serious way or it happens frequently by the same person, the following may happen:
a) The group will discuss the issue at hand.
b) Oral or written warning will be given to the member in question. The warning will also contain suggestions about how the member can redeem himself.
c) Call the supervisor so that we can agree on more serious measure if the member is not affected by warnings given to him earlier.
d) Call the supervisor and vote for an exclusion of the member in question.

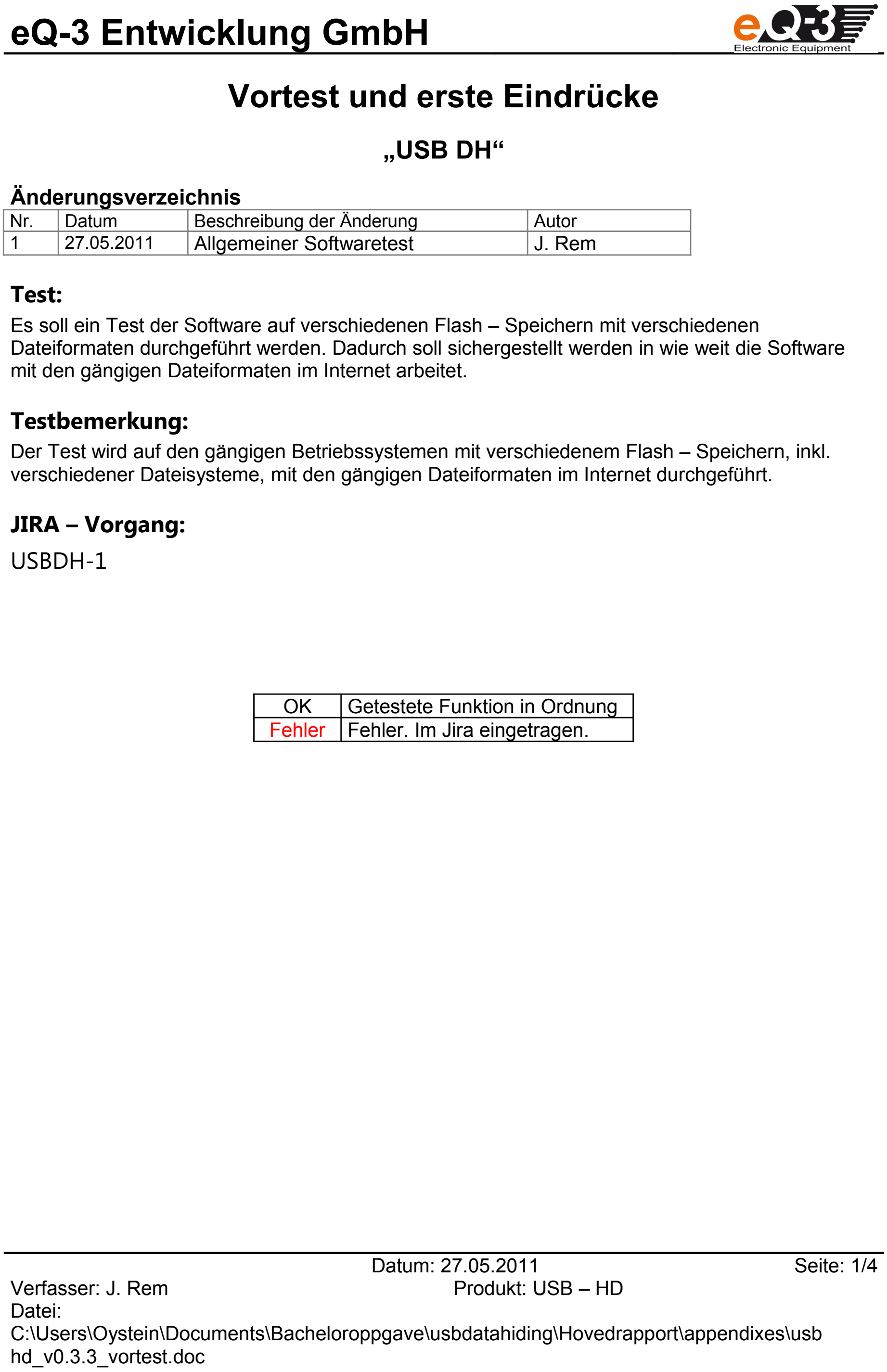
I hearby accept the terms given by the group and I am aware of the measures taken if I fail to comply.

Date: 28/01 - 11 Name: Øystein Nilsen

Date: 28101 - 11 Name: Rune Søbye

Date: 28/01 - 11 Name: Knut Borg

H External test results



eQ-3 Entwicklung GmbH

Vortest und erste Eindrücke

„USB DH“

Änderungsverzeichnis

Nr.	Datum	Beschreibung der Änderung	Autor
1	27.05.2011	Allgemeiner Softwaretest	J. Rem

Test:

Es soll ein Test der Software auf verschiedenen Flash – Speichern mit verschiedenen Dateiformaten durchgeführt werden. Dadurch soll sichergestellt werden in wie weit die Software mit den gängigen Dateiformaten im Internet arbeitet.

Testbemerkung:

Der Test wird auf den gängigen Betriebssystemen mit verschiedenem Flash – Speichern, inkl. verschiedener Dateisysteme, mit den gängigen Dateiformaten im Internet durchgeführt.

JIRA – Vorgang:

USBDH-1

OK	Getestete Funktion in Ordnung
Fehler	Fehler. Im Jira eingetragen.

Datum: 27.05.2011 Seite: 1/4
Verfasser: J. Rem Produkt: USB – HD
Datei:
C:\Users\Oystein\Documents\Bacheloroppgave\usbdatahiding\Hovedrapport\appendixes\usbhd_v0.3.3_vortest.doc

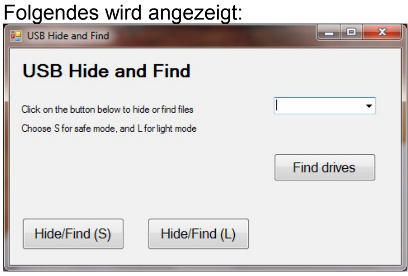
Prüfer: Juri Rem **Checkliste: USB – DH** Version: 0.3.3

Windows 7 Professional SP1 deutsch, 64 Bit:

Modell:	CnMemory Micro X Pro	CnMemory Airy
Flash – Speicher:	USB DISK V2.0	USB DISK V2.0
Dateisystem:	FAT32	FAT32
Größe:	987MB	7,51GB

Einleitung:
Das Programm wird gestartet in dem man die Datei B_GUIx64.exe für 64Bit Betriebssystem (B_GUIx86.exe für 32Bit) startet.

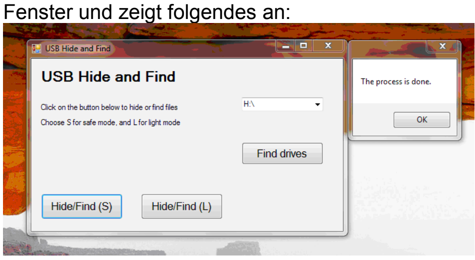
Folgendes wird angezeigt:



Click on the button below to hide or find files
Choose S for safe mode, and L for light mode

Bild1

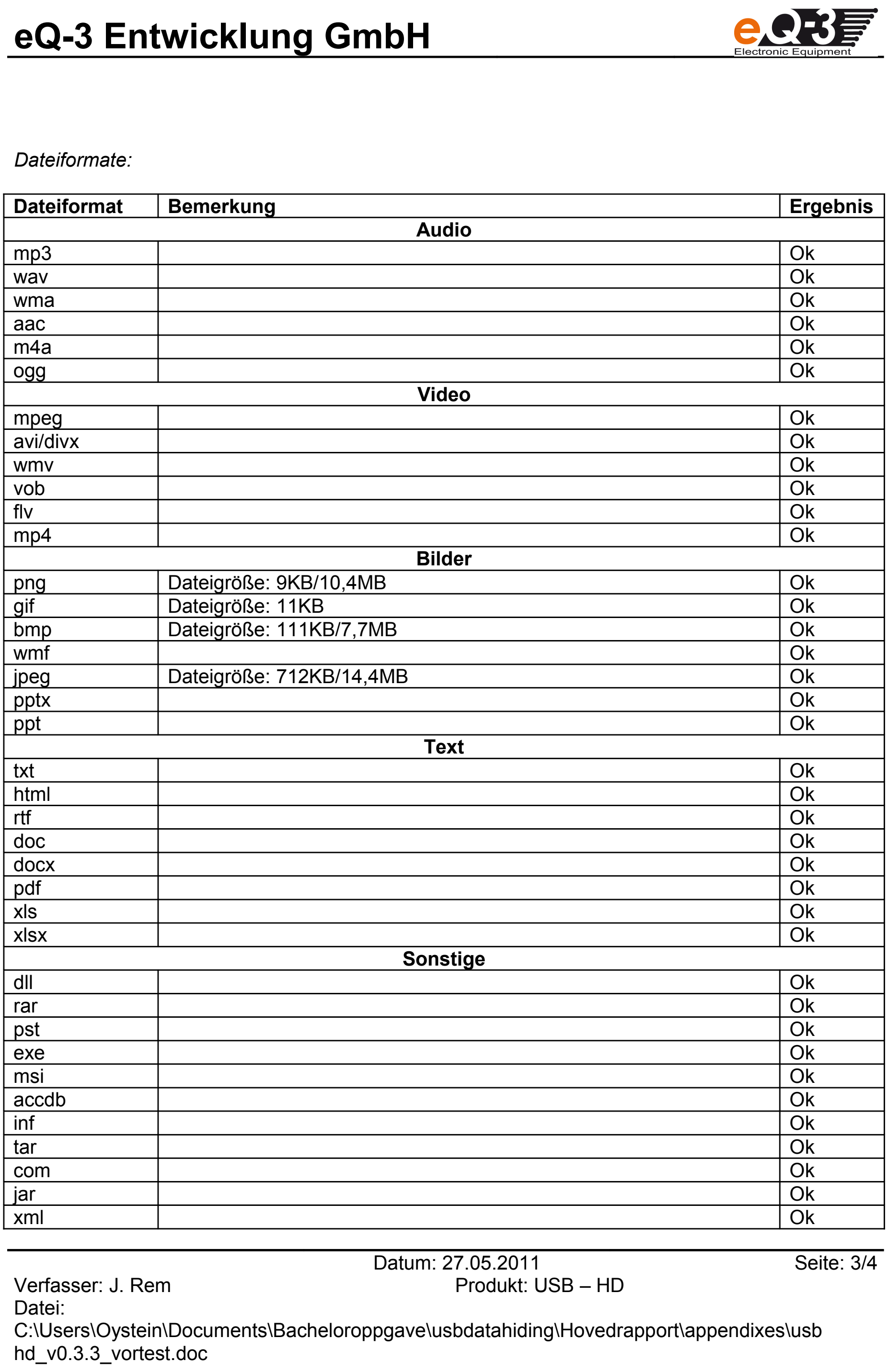
Das Programm wertet sofort nach dem Start die Wechseldatenträger aus und erkannt automatisch die Laufwerkbuchstaben. Diese sind anschließend in der Auswahlliste aufgelistet.
Wie in Bild1 zu sehen gibt es zwei Auswahlmodi. In beiden Modi sind die Daten auf dem Wechseldatenträger nicht mehr zu sehen. Jedoch kann Win7 im Modus (S) den Wechseldatenträger auswerten und den belegten Speicher anzeigen.
Dies ist im Modus (L) nicht der Fall.
Der Prozess startet und der Wechseldatenträger wird beschrieben. Während des Vorgangs kann der Wechseldatenträger nicht genutzt werden. Nach erfolgreicher Ausführung erscheint ein neues Fenster und zeigt folgendes an:



Click on the button below to hide or find files
Choose S for safe mode, and L for light mode
H:\
The process is done.

Bild2

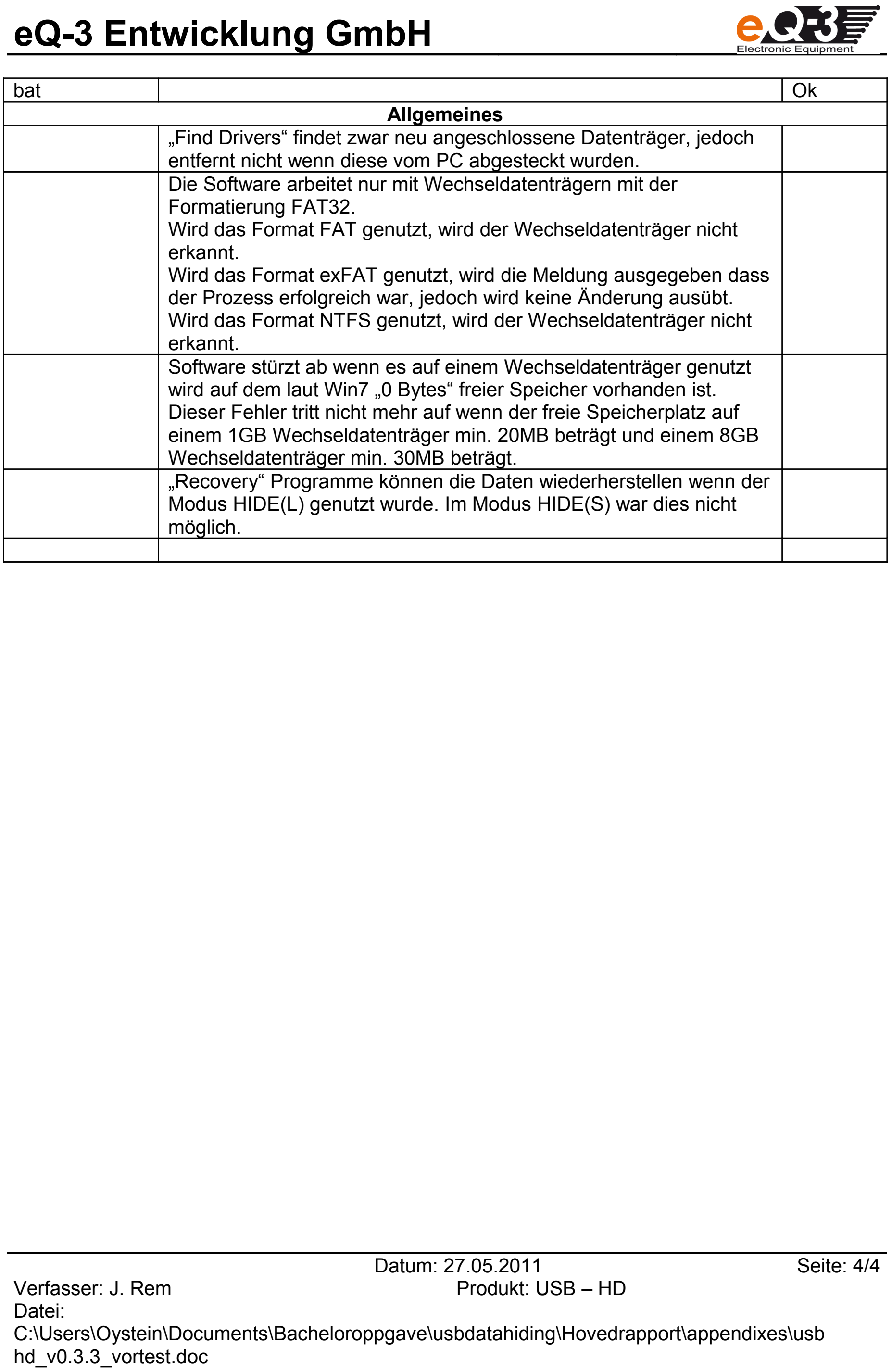
Somit ist der Prozess erfolgreich abgeschlossen und die Daten auf dem Wechseldatenträger sind für den Anwender nicht mehr sichtbar. Die Daten können wiederhergestellt werden indem man die jeweilige Funktion ein zweites Mal ausführt und somit einen „FIND“ Befehl ausführt. Nach erfolgreichem Abschluss erscheint ebenfalls Bild2 und die Daten sind wieder sichtbar.



eQ-3 Entwicklung GmbH

Dateiformate:

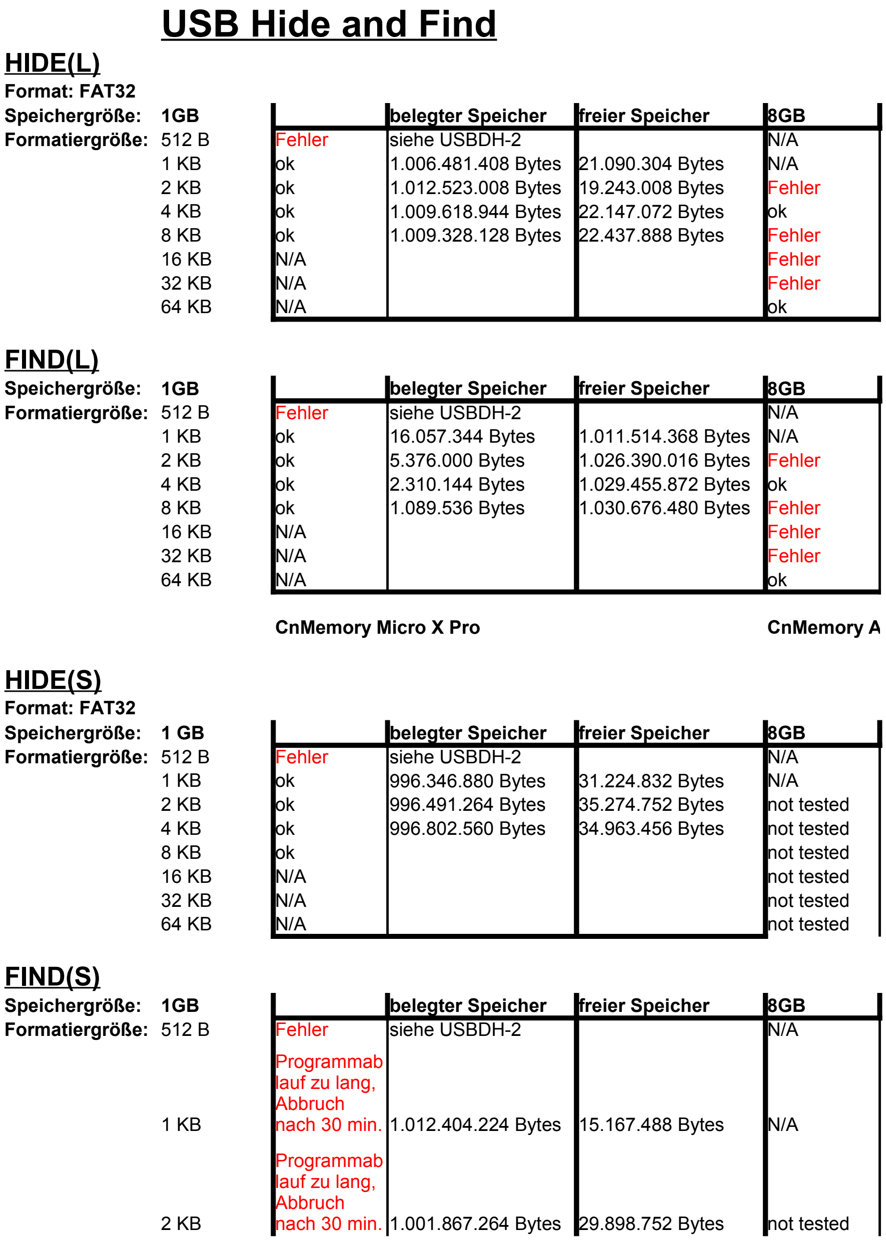
Dateiformat	Bemerkung	Ergebnis
	Audio	
mp3		Ok
wav		Ok
wma		Ok
aac		Ok
m4a		Ok
ogg		Ok
	Video	
mpeg		Ok
avi/divx		Ok
wmv		Ok
vob		Ok
flv		Ok
mp4		Ok
	Bilder	
png	Dateigröße: 9KB/10,4MB	Ok
gif	Dateigröße: 11KB	Ok
bmp	Dateigröße: 111KB/7,7MB	Ok
wmf		Ok
jpeg	Dateigröße: 712KB/14,4MB	Ok
pptx		Ok
ppt		Ok
	Text	
txt		Ok
html		Ok
rtf		Ok
doc		Ok
docx		Ok
pdf		Ok
xls		Ok
xlsx		Ok
	Sonstige	
dll		Ok
rar		Ok
pst		Ok
exe		Ok
msi		Ok
accdb		Ok
inf		Ok
tar		Ok
com		Ok
jar		Ok
xml		Ok



eQ-3 Entwicklung GmbH

bat		Ok
	Allgemeines	
	„Find Drivers“ findet zwar neu angeschlossene Datenträger, jedoch entfernt nicht wenn diese vom PC abgesteckt wurden.	
	Die Software arbeitet nur mit Wechseldatenträgern mit der Formatierung FAT32. Wird das Format FAT genutzt, wird der Wechseldatenträger nicht erkannt. Wird das Format exFAT genutzt, wird die Meldung ausgegeben dass der Prozess erfolgreich war, jedoch wird keine Änderung ausübt. Wird das Format NTFS genutzt, wird der Wechseldatenträger nicht erkannt.	
	Software stürzt ab wenn es auf einem Wechseldatenträger genutzt wird auf dem laut Win7 „0 Bytes“ freier Speicher vorhanden ist. Dieser Fehler tritt nicht mehr auf wenn der freie Speicherplatz auf einem 1GB Wechseldatenträger min. 20MB beträgt und einem 8GB Wechseldatenträger min. 30MB beträgt.	
	„Recovery“ Programme können die Daten wiederherstellen wenn der Modus HIDE(L) genutzt wurde. Im Modus HIDE(S) war dies nicht möglich.	

Datum: 27.05.2011 Seite: 4/4
Verfasser: J. Rem Produkt: USB – HD
Datei:
C:\Users\Oystein\Documents\Bacheloroppgave\usbdatahiding\Hovedrapport\appendixes\usbhd_v0.3.3_vortest.doc



USB Hide and Find

HIDE(L)

Format: FAT32

Speichergröße: **1GB**

Formatiergröße:		**belegter Speicher**	**freier Speicher**	**8GB**
512 B	Fehler	siehe USBDH-2		N/A
1 KB	ok	1.006.481.408 Bytes	21.090.304 Bytes	N/A
2 KB	ok	1.012.523.008 Bytes	19.243.008 Bytes	Fehler
4 KB	ok	1.009.618.944 Bytes	22.147.072 Bytes	ok
8 KB	ok	1.009.328.128 Bytes	22.437.888 Bytes	Fehler
16 KB	N/A			Fehler
32 KB	N/A			Fehler
64 KB	N/A			ok

FIND(L)

Speichergröße: **1GB**

Formatiergröße:		**belegter Speicher**	**freier Speicher**	**8GB**
512 B	Fehler	siehe USBDH-2		N/A
1 KB	ok	16.057.344 Bytes	1.011.514.368 Bytes	N/A
2 KB	ok	5.376.000 Bytes	1.026.390.016 Bytes	Fehler
4 KB	ok	2.310.144 Bytes	1.029.455.872 Bytes	ok
8 KB	ok	1.089.536 Bytes	1.030.676.480 Bytes	Fehler
16 KB	N/A			Fehler
32 KB	N/A			Fehler
64 KB	N/A			ok

CnMemory Micro X Pro **CnMemory A**

HIDE(S)

Format: FAT32

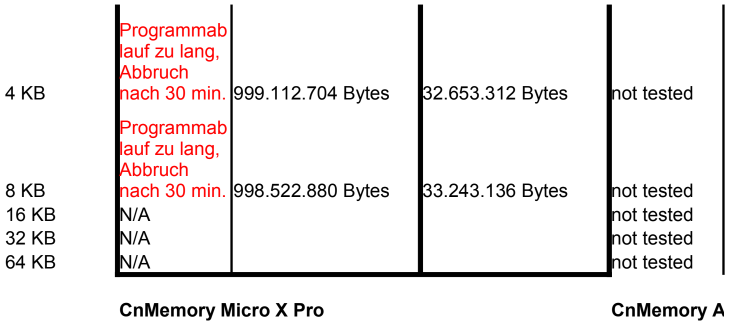
Speichergröße: **1 GB**

Formatiergröße:		**belegter Speicher**	**freier Speicher**	**8GB**
512 B	Fehler	siehe USBDH-2		N/A
1 KB	ok	996.346.880 Bytes	31.224.832 Bytes	N/A
2 KB	ok	996.491.264 Bytes	35.274.752 Bytes	not tested
4 KB	ok	996.802.560 Bytes	34.963.456 Bytes	not tested
8 KB	ok			not tested
16 KB	N/A			not tested
32 KB	N/A			not tested
64 KB	N/A			not tested

FIND(S)

Speichergröße: **1GB**

Formatiergröße:		**belegter Speicher**	**freier Speicher**	**8GB**
512 B	Fehler	siehe USBDH-2		N/A
1 KB	Programmablauf zu lang, Abbruch nach 30 min.	1.012.404.224 Bytes	15.167.488 Bytes	N/A
2 KB	Programmablauf zu lang, Abbruch nach 30 min.	1.001.867.264 Bytes	29.898.752 Bytes	not tested



4 KB	Programmablauf zu lang, Abbruch nach 30 min.	999.112.704 Bytes	32.653.312 Bytes	not tested
8 KB	Programmablauf zu lang, Abbruch nach 30 min.	998.522.880 Bytes	33.243.136 Bytes	not tested
16 KB	N/A			not tested
32 KB	N/A			not tested
64 KB	N/A			not tested
	CnMemory Micro X Pro			CnMemory A

belegter Speicher	freier Speicher
siehe USBDH-6	
8.053.784.576 Bytes	32.800.765 Bytes
siehe USBDH-5	
siehe USBDH-5	
Daten werden vom Wechseldatenträger gelöscht	
8.067.743.744 Bytes	31.391.744 Bytes

belegter Speicher	freier Speicher
siehe USBDH-6	
18.055.168 Bytes	8.068.530.176 Bytes
siehe USBDH-5	
siehe USBDH-5	
Daten werden vom Wechseldatenträger gelöscht	
1.114.112 Bytes	8.098.021.376 Bytes

iry

belegter Speicher	freier Speicher

belegter Speicher	freier Speicher

iry