VERSITA

TATRA MOUNTAINS
Mathematical Publications
DOI: 10.2478/tmmp-2013-0036
Tatra Mt. Math. Publ. **57** (2013), 65–83

STEGANOGRAPHIC FILE SYSTEM BASED ON JPEG FILES

Matúš Jókay — Martin Košdy

ABSTRACT. A steganographic system provides a hidden communication channel in background of a public channel. The existence of the hidden channel must remain secret, i.e. the adversary cannot decide whether the public channel contains any covert information or not. The public channel that is used in construction of a steganographic system can often be embedded in a static file (medium), that is called a carrier (if the steganographic information is present). Most of the current research focuses on a single medium. The most suitable types of media, such as images or music files, contain a lot of redundancy. Small changes in the redundant parts are not easily detected. However, new methods for the detection of this information are developed along with the new algorithms for embedding the hidden information. Our work describes a new steganographic system design, where the hidden information is spread among many static images in a form of a virtual steganographic filesystem. We note that the implementation of the system must also take into account "steganographic side-channels", i.e., some information channels that are present in the operating system (in our case Linux, and Android) that leak information about the presence of the hidden channel.

1. Introduction

The steganography has advanced tremendously in the last decade. Currently, there exist numerous sophisticated steganographic techniques of hiding the secret data in the various kinds of cover media. [19] These methods can be also used in a conjunction with the existing cryptographic tools to improve their security [20].

In this paper we present the possibility of creating a virtual disk whose content is stored within several cover files. We focus on the JPEG file steganography and

2010 Mathematics Subject Classification: 68P30, 94B05, 94A08.
Keywords: steganography, file system, side-channels.
This material is based upon work supported under the grants VEGA 1/0173/13, VEGA 1/0529/13, APVV-0586-11 and NATOs Public Diplomacy Division in the framework of "Science for Peace", SPS Project 984520.

the techniques ensuring that the hidden storage is uniformly diffused over the entire set of the carrier files. This diffusion must be key-dependent so only the legitimate user with knowledge of the key is able to extract the hidden storage correctly. We present several methods of achieving this property and discuss their advantages and disadvantages in connection with concrete implementations.

The structure of the article is as follows: firstly, in section 2 we provide a short introduction to the steganography, which is followed by an overview of the existing steganographic techniques and the currently available stegosystems. In section 3 we specify the requirements and the core features of the proposed system. Later, in section 4 we provide the design and the performance analysis of our proposal. We also examine various system limitations and propose possible solutions. Finally, in section 6, we present the performance benchmarks of the implemented system and conclude the overall contribution of our work.

2. Preliminaries

2.1. Modern steganography

A recent definition says that: "Steganography is the art of hiding the presence of communication by embedding secret messages into innocent, innocuous looking cover documents, such as digital images, videos, sound files." [8]

In other words, modern steganographic techniques are used to hide the content of secret message in digital documents called *cover files*. Cover files containing secret information are called *carrier files*.

In the past, steganography was often mistaken with cryptography and the first stego techniques were considered as a part of cryptography. [1] The main difference between these two scientific disciplines is that cryptography aims to protect secret message by its transformation in a way that can not be reversed without the knowledge of the secret key. On the other hand the main goal of steganography is to hide the secret message in cover medium that should not look suspicious to the attacker.

According to [11], steganographic systems can be divided into these groups:

1. Pure steganography,
2. Secret key steganography,
3. Public key steganography.

Security of the pure steganographic systems is based on two main assumptions. The first one states that the original cover medium cannot be publicly known. If it was available to the attacker, the presence of the hidden message in cover medium could be easily discovered. The second assumption states that the embedding algorithm must also remain secret, since the embedding process does not involve any kind of a secret key. This statement is in full contradiction with the Kerckhoff's principle.

It is always good to assume that the details of the stego algorithm are already available to the attacker. Security of the system should rely on the secret key which is known only to valid participants of the communication. In spite of the fact that the steganographic system is compromised, when the presence of the secret message is detected, the content of the hidden message should remain secret until the correct key is used. In case of stegosystems, this property is often achieved by the use of key-dependent permutations.

Similarly as in public key cryptography, public key steganography systems were designed in order to avoid a problematic exchange of the secret key. Public key is used in the process of embedding so anyone can hide some secret message into the cover file, but only the legitimate receiver can extract the hidden message using private key.

2.2. Existing steganographic systems

In this section we present a brief overview of existing steganographic utilities. We are mainly focused on applications that utilize the image-based steganography and tools that allow the use of multiple carrier files as a cover (or stego) medium.

2.2.1. Systems working with a single cover file

There are dozens of stego tools allowing to hide a secret message in a single carrier file. We are mainly interested in JPEG steganography, so here we present a short list of tools that are, in our opinion, the most important from the historical point of view.

JSteg: this is probably the first JPEG stego utility, it uses simple sequential embedding [14].

OutGuess: uses PRNG to permute DCT coefficients and RC4 to encrypt the hidden data [17].

Steghide: uses compression and encryption of embedded data. It also stores checksums to verify the data integrity [15].

JP Hide & JP Seek: uses the Blowfish algorithm for least significant bit randomization and encryption [16].

F5: implements matrix encoding to improve the efficiency of embedding and employs permutative straddling to uniformly spread out the changes over the whole steganogram [7].

2.2.2. Systems working with multiple carrier files

A majority of currently available stego tools work only with a single cover file, but there are also several tools that can handle multiple carriers. *OpenPuff* [13] is a really interesting tool that supports various carrier formats (images, audio/video files, SWFs, PDFs), is free and portable. It implements several layers that handle encryption, scrambling, whitening and encoding of the hidden data. The only disadvantage is that it does not provide a virtual disk functionality and it allows to hide only one secret file. Of course, this issue can be overcome by placing multiple secret files in a single archive file.

2.2.3. Steganographic filesystems

The first steganographic file system was proposed by Ross Anderson, Roger Needham and Adi Shamir in [2]. The main purpose of this system is to give the user a high level of protection against being compelled to disclose its contents. In this system, a file can be delivered only to user who knows its name and password. Moreover, an attacker can gain no information about whether the file is present, even if he/she has a full access to the hardware. This property is achieved by randomizing the entire partition and writing the encrypted files to pseudo-random locations using a key derived from the file name and directory password. Due to the fact that the encrypted files strongly resemble the randomized sections of partition, there is no easy way to distinguish between unoccupied space and the actually encrypted file. One of the main drawbacks of this approach is that the files can overwrite each other so they need to be stored in multiple locations in order to minimize the probability of data loss. This leads to poor read/write performance and reduced storage capacity.

Authors of [3] presented an implementation of steganographic file system for Linux called *StegFS*. They extended the standard Linux file system (ext2fs) with a plausible-deniability encryption function. The design of this hidden file system was inspired by [2] but, as the authors claim, it is more practical and efficient. Instead of using entire partition, they place the hidden files into unused blocks of a partition that also contains normal files. They also use a separate block allocation table in order to avoid collisions that lead to overwriting of the hidden data.

3. Components of the proposed steganographic system

In this section we discuss various aspects of steganographic systems that need to be considered. Then in next section we propose the design of a new steganographic system.

3.1. Pseudorandom permutations

The main purpose of a key-dependent permutation in our system is to achieve that only the legitimate user with knowledge of the key can reveal the hidden storage.

The permutation layer should also assure that the hidden data are not embedded in carrier files sequentially in the order as they are stored in hidden storage, but they should be uniformly distributed over the entire set of carrier files. This property should also help to defend the system against various steganalytic techniques. [6]

Required properties of a permutation generator are:

1. Key dependency,
2. Fast computation,
3. Low memory footprint.

3.1.1. Affine cipher

The affine cipher is probably the simplest and the most efficient key-dependent pseudo-random permutation. It works on exactly the same principle as the *linear congruential generator (LCG)*, which is commonly used by most compilers to implement the *rand()* function in their runtime libraries. Calculation of individual elements is really fast and except the two coefficients it does not require any additional memory.

According to [9], the individual elements of the permutation based on affine cipher are calculated using following equation

$$p(x) = ax + b \mod m, \quad \gcd(a, m) = 1. \tag{1}$$

Every change in coefficients a and b leads to a different permutation, so we can think of these parameters as two parts of the key. The only thing we need to do is to derive them from the master key. This can be done by splitting the binary representation of the master key into two parts and calculating their congruent values modulo m. To ensure that the function is bijective, a must be coprime to m. The number of integers coprime to m in range $< 1, m >$ is given by the Euler's totient function $\phi(m)$. In order to maximize this number and also the key space, we decided to choose m as a prime, so the number of possible values of a is always equal to $m-1$. In case that the input capacity is not prime, the closest smaller prime is chosen to initialize this permutation. To achieve better diffusion properties of this permutation it is also recommended to use only values within the range $< m/2, m-1 >$ for both parameters.

The main drawback of this solution is quite small key space since there are only $(m-1)m$ distinct values that can be used as a key and if we use only the values from the second half of the range, the number of possible keys will be even smaller ($(m-1)m/4$).

3.1.2. Feistel network

Feistel network provides an efficient solution for constructing an invertible pseudorandom permutation from a pseudorandom function which does not need to be invertible. In this work we are using cryptographically strong hash function whose output values are precomputed and stored in memory in order to achieve better performance. This approach is a bit slower than the affine cipher based permutation, but it offers a significantly larger key space.

In this work we present two similar approaches on how to split a number into two parts that are necessary for Feistel construction. [5] The first one, which is depicted in Figure 1, enciphers a number

$$x = aN + b \in \mathbb{Z}_{MN}, (a \in \mathbb{Z}_M, b \in \mathbb{Z}_N). \tag{2}$$

The second version, depicted in Figure 2, uses a slightly different representation utilizing two different group operations in order to achieve better performance. It enciphers a number in following form

$$x = a.2^n + b \in \mathbb{Z}_{M2^n}, (a \in \mathbb{Z}_M, b \in \mathbb{Z}_{2^n}). \tag{3}$$

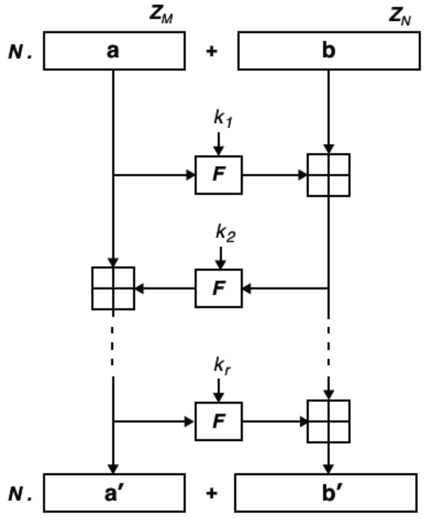

FIGURE 1. Numeric Feistel.

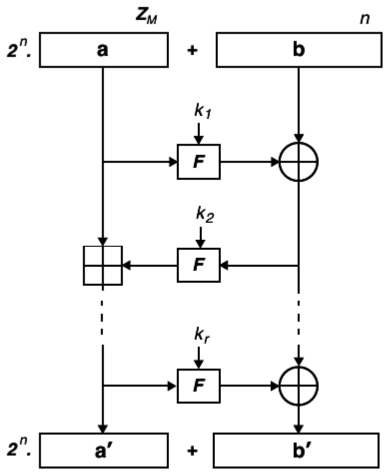

FIGURE 2. Mixed Feistel.

Both numeric division and multiplication by 2^n can be implemented using single bitwise shift and the modular addition on the right side of Feistel network can be replaced with operation xor which does not need to be followed by modulo operation.

Neither these two constructions allow to use an arbitrary value as a size of the permutation. Again, if the capacity of the storage can not be written in a form which is specific for chosen construction, the closest smaller value satisfying associated equation is chosen.

In order to achieve the smallest possible memory footprint and to minimize the time needed for initialization of such a permutation, we decided to use the balanced version of the Feistel network. As a result of this, the numeric version is initialized with $M = N = \sqrt{c}$, where c is the required capacity. The final capacity offered by this permutation is therefore equal to $\lfloor\sqrt{c}\rfloor^2$, in other words, it is the closest smaller (or equal) square to the required capacity. The approach used in the second presented Feistel construction utilizes computation in different groups so in most cases it simply can not be balanced. To achieve the smallest difference in sizes of these groups we decided to initialize this permutation using following equations:

$$n = \log_2(c)/2, \qquad M = \lfloor c.2^{-n} \rfloor$$

where c is the required capacity.

The final capacity *cap* offered by this permutation is therefore equal to

$$cap = 2^{\lfloor \log_2(c)/2 \rfloor}.\lfloor c.2^{-\lfloor \log_2(c)/2 \rfloor} \rfloor. \tag{4}$$

In other words, it is the closest smaller (or equal) multiple of $\lfloor \log_2(c)/2 \rfloor$th power of 2 .

According to [5], block cipher $\mathrm{Feistel}^r_{\#}[M,N] : \mathcal{K} \times \mathbb{Z}_{MN} \to \mathbb{Z}_{MN}$ has key space

$$\mathcal{K} = \big(\mathrm{fnc}\big(\mathbb{Z}_N, \mathbb{Z}_M\big)\big)^r. \tag{5}$$

where $\mathrm{fnc}(\mathbb{Z}_N, \mathbb{Z}_M)$ is the set of all functions from $\mathbb{Z}_N$ to $\mathbb{Z}_M$ and r is the number of rounds.

Applied to our system, where $M = N = \lfloor \sqrt{c} \rfloor$ and the round function is a cryptographically strong hash function, the upper bound of the key space is $(\lfloor \sqrt{c} \rfloor !)^r$.

3.2. Virtual Disk

Almost every operating system provides a mechanism for adding support for new hardware, that is often achieved by loadable kernel modules (LKM). Windows and most current Unix-like systems support these modules, although they may use different names for them. In Windows they are called *kernel-mode drivers*, in OS X they are known as *kernel extensions (kext)*.

This way we could implement a loadable kernel module that would offer a virtual disk functionality. However, the presence of such a module may cause suspicion that there is also some hidden storage present in the files on hard drive. Another disadvantage of this solution is that kernel modules can be loaded only by users with root privileges. Proper implementation of these modules is also a quite difficult task.

To avoid all these issues we examined alternative techniques of providing virtual drive functionality, that are presented in following paragraphs.

In Unix-like systems, there is a possibility of creating a *loop device*, which is a pseudo-device that makes a file accessible as a block device. In case that this file contains an entire file system, it can be mounted as if it was a disk device. It is often used for mounting CD or DVD images.

This way, we could extract the entire content of hidden storage to temporary file and attach it as a loop device, but we wanted to avoid creating temporary files on hard drive in order to achieve better performance and also to prevent other possible side channel attacks.

There is also a possibility to create a virtual file system provided by FUSE, which is an acronym for *Filesystem in Userspace*. It is a loadable kernel module that lets non-privileged users create their own file systems. This is achieved by running the file system implementation code in user space while the FUSE module provides a bridge to the actual kernel interfaces [10].

This module allows us to create a virtual file system containing single disk image file that can be later attached as a loop device. Content of this file is provided by our application using FUSE callback methods, so it can be stored in system memory. Moreover, it can be dynamically extracted from carrier files during each read attempt and analogically embedded into carrier files on every single write attempt. This technique is also used by TrueCrypt to create a virtual encrypted volume so we decided to utilize the same approach.

3.3. Key generation

One of the main requirements for the permutation is key-dependency. Every change of a bit in the key should lead to generation of a different permutation.

Another thing to consider is which parameters should be used for generation of the master key. This key is later used to derive subkeys for specific permutation generators.

Possible parameters that **could be involved** in master key generation:

- User-entered password,
- Names of carrier files in alphabetical order,
- Relative paths to carrier files in alphabetical order,
- Names of other files (not carrier files) in selected directory,
- Carrier-specific data (EXIF data in JPEG files, ID3 tags in mp3 files, ...),
- Capacity offered by individual carriers,
- Checksums or hash values of individual carriers computed without bits which are used for data embedding.

Parameters that **should not affect** the master key generation:

- Absolute paths to carrier files,
- Time attributes of carrier files,
- Checksums or hash values of individual carriers.

If we used absolute paths to carrier files, it would be almost impossible (or at least not practical) to mount the system from other location as that which was used at the time of hidden storage creation.

In this context, the main reason why we decided not to use time attributes as input for key generation is that it would cause similar issues when the user wants to transfer his hidden storage to another location or send it to another legitimate user.

There is no doubt that checksums or hash values of individual carrier files should not be used in master key generation, just because it would make the system unusable.

The list of parameters we decided to use in master key generation:

- User-entered password (optional parameter),
- Relative paths to carrier files in alphabetical order,
- Capacity offered by the individual carriers.

In order to avoid the possible steganalytic attacks that may exploit some statistical properties in a case that the same permutation was used within the several carrier files with the same capacity, we decided to initialize every *local* permutation with a different subkey. Since these subkeys are derived from the master key, every parameter that affects the master key also affects the individual subkeys.

In order to achieve required properties of the master key, we decided to use hash function in the key derivation process. We have chosen the Keccak [4] sponge function which is current winner of the last NIST hash function competition so it has been selected as a new standard known as SHA-3.

3.4. Side channels

Even if we used a visually and statistically undetectable steganographic algorithm (currently there is no known algorithm for which the statistical undetectability is proven), we should examine possible leaks of side channel information that could help an adversary reveal the hidden storage.

3.4.1. Time attributes

In the UNIX-like environment, each file has three time stamps associated with it: its access time (*atime*), its modification time (*mtime*), and its attribute modification time (*ctime*). [12] Attributes *mtime* and *atime* can be modified by system call touch or system function *utime*. The problem is only with the attribute *ctime* which is changed by operating system when the *inode* of the file is modified. This one cannot be modified by any system call without unmounting file system or changing system time.

In case that the timestamps of individual carriers were modified during embedding process, it would be really easy for an adversary to identify those files.

3.4.2. Memory buffers

Every buffer stored in memory containing some sensitive data should be overwritten by random data before its deallocation. It does not matter whether the buffer stores encrypted or unencrypted data, keys in raw form or the hashed version of the password. All these kinds of information can be potentially exploited by an attacker in case they are not properly destroyed when the application exits.

4. Design and performance analysis

In this section, we present two slightly different architectures. They share the majority of the code located in individual layers, but those layers are stacked in a different order. As a result of this, these two architectures vary in important properties such as memory footprint, I/O vs. mount/unmount performance and so on. The order of layers in the first version (Figure 3) reflects the process of adding new features during the work on this paper. The purpose of designing second architecture (Figure 4) was to achieve better performance and lower memory footprint so we will call it *optimized version*.

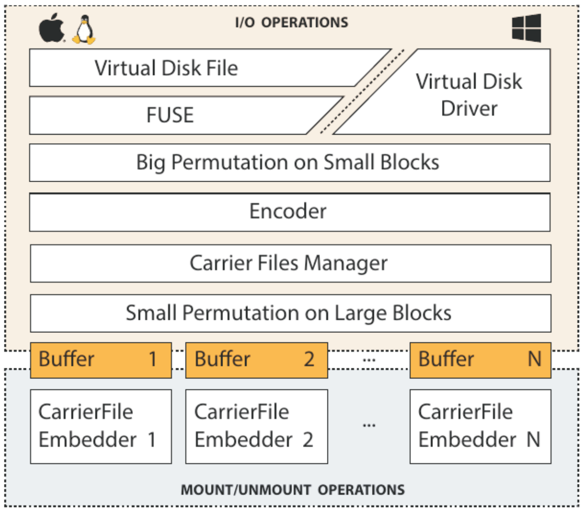

FIGURE 3. The first version of steganographic system design.

In the diagrams we divided layers and blocks into two parts. The first part contains layers and blocks involved in mounting and unmounting. Each of these operations is executed only once, when the user wants to reveal his/her hidden file system and when he/she wants to finish working with his/her hidden data. The second part contains blocks and layers responsible for I/O operations, which are executed during each read/write attempt.

As it is shown in Figure 3, this version keeps the main part of computation on the I/O side. This solution has potentially negative effect on I/O performance. On the other hand, it provides the way how the memory footprint and mounting/unmounting times can be minimized.

While designing this version, we wanted to create a system that would be able to handle gigabytes of data contained in the hidden storage. Such a system

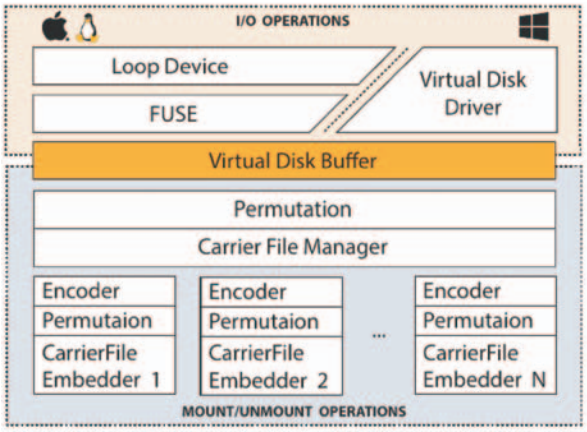

FIGURE 4. The optimized version of steganographic system design.

must be able to dynamically load and store data contained in carrier files during I/O operations in order not to store whole storage in RAM. Dynamic caching like this could work if the hidden data was written in carrier files sequentially in natural order. But the another requirement for our system says that the data should be permuted in order to achieve diffusion, which is the exact contradiction of previous requirement. In case we used a good permutation to diffuse hidden data between carrier files, almost every I/O attempt would lead to reading and writing every single carrier file. If the read and write operations need the JPEG file to be decompressed and compressed again, this would lead to unacceptable performance penalty.

On the other hand, in the real world situation, if the user wanted to create a hidden storage with capacity in gigabytes, the overall size of the carriers required to store its content would be in hundreds of gigabytes. This scenario is very unlikely to happen so in the most cases the capacity of RAM should be sufficient to store entire hidden storage in memory.

Another advantage of this solution is in its versatility. Layers that handle I/O operations (like encoder and permutation) can share the same interface so they can be stacked in various combinations and read/write operations could be represented as streams. The bad thing is that every added layer comes with significant performance penalty.

After several optimizations of the first version we decided to make some changes in the architecture, that will lead to better performance. As it is shown in Figure 4, we moved the whole computation part into the mount/unmount phase.

We relocated the coding layer into individual carrier file embedders where we also placed an another key-dependent permutation. Such modification makes the architecture of the bottom three layers similar to existing single-file embedding algorithms like F5 [7].

It also makes it easier to implement another embedding algorithms that may use completely different coding techniques or data diffusion methods since the use of local permutation and encoder is optional.

In the next parts of this section we will describe individual parts of the system. We will focus only on the optimized version of architecture in the next parts of this work.

4.1. Carrier File Manager

Carrier files manager handles following tasks:

- Directory content enumeration,
- Creating and handling CarrierFile instances,
- Master key generation,
- Deriving subkeys for individual CarrierFile instances,

4.2. Carrier file

Steganographic embedding is specific for every single carrier file format so we decided to place the embedding algorithm in objects that share the same interface. In other words, Carrier file is an object responsible for embedding and extracting data from single carrier file. Each supported file format has its own derived Carrier file object, so it is easy to extend the application by supporting various file formats and different stego embedding algorithms. In this work we have implemented two objects for handling BMP and JPEG files. Both use the permuted LSB method for embedding that is used in connection with Hamming encoder. This solution allows to use different encoder for every single file loaded in carrier files manager, but it is also possible to use single shared encoder. Permutation object needs to be created for each Carrier file instance separately due to the fact it has to be initialized with a particular subkey.

4.3. Encoder

This part of system allows to use coding of data during embedding process in order to minimize the number of changes in carrier file. Encoder provides the interface for coder initialization, accessing the block size information and for encoding and decoding data during the process of embedding or extraction. The only encoder implemented in this work is the Hamming encoder. [18] This class also affects the capacity of the medium, because the encoder works with blocks of size which is specific for each code.

4.4. Permutation

As it is shown in Figure 4, this architecture incorporates permutations at two different places. There is one *local* permutation for each carrier file embedder and one *global* permutation that permutes bytes of main storage buffer.

Global permutation operates on positions of bytes in main data buffer. This module ensures that the bytes of hidden storage that are stored in a sequence one after another will be stored in different carrier files.

The main purpose of permutations that belong to carrier file embedders is to avoid extraction of data stored in individual carriers without knowledge of the key. It should also help with defending individual carriers against steganalysis.

The size of permutation depends on the algorithm used for its generation, so it does not have to be the same number as the input capacity, but if they are not equal, the size of permutation is always smaller than input capacity. This is another module which affects the overall medium capacity.

4.5. Virtual disk driver

Virtual disk driver is used to create a buffer used to store the content of the hidden storage in system memory. It is the last platform independent part of the system. On the UNIX-like systems as Mac OS X or Linux, this class is directly connected to FUSE I/O callbacks.

4.6. FUSE

FUSE is used to provide a virtual file system containing one single file - virtual disk image. This file is later attached as a virtual disk, which can be partitioned and formatted with any file system that is supported by operating system.

5. Performance testing

The benchmarking was performed on a MacBook Pro with 2.4 GHz Intel Core 2 Duo processor. The machine was equipped with 8GB (1066 MHz DDR3) RAM and 500GB SATA2 7200RPM hard drive with 16MB cache. All pictures used in this test were captured by digital camera with the same quality settings and resolution (2592×1936).

5.1. Permutations

We presented two different permutation generators. Both of them were implemented in two slightly modified versions, so in fact, we had four different implementations to compare. Permutation generation consist of two phases.

The first one, initialization, is performed only once and its performance is depicted in the Figures 5 and 7. Another operation we are interested in is the calculation of the individual permutation elements. During the extraction, this operation is performed for each permutation element once, during the embedding it needs to be performed twice. Its performance is depicted in the Figure 6. From the users' point of view, the most important is the overall computation time which involves both, the initialization and the generation of the whole permutation. It is shown in the Figures 6 and 8. The slowest implementation is a 64-bit version of the Affine cipher. Unfortunately, it was caused by the non optimal compiler implementation of the modular multiplication algorithm. Other implementations utilize only the computations with the constant time complexity.

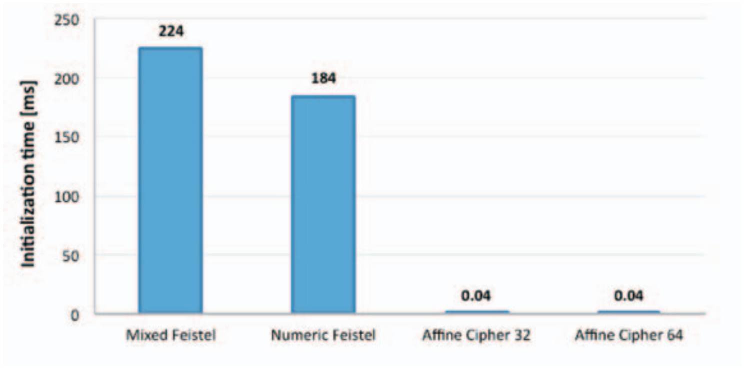

Figure 5. Initialization time of the permutation with size 100M.

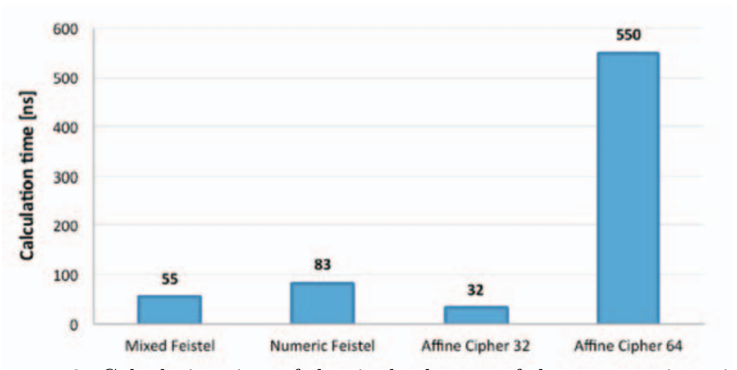

Figure 6. Calculation time of the single element of the permutation with size 100M.

5.2. Overall performance

The most time consuming operations in the process of (un)mounting are the reading and writing the carrier files followed by their (de)compression in case of JPEG files. As a result of this, the selection of the permutation does not have a significant impact on the overall system performance. As it can be seen in the Figure 9, the overall loading/saving time increases linearly with the system capacity. This benchmark was created using the Mixed Feistel permutation and the Hamming code (7,4). Obviously, working with bitmaps takes a significantly shorter amount of time due to the compression and decompression that needs to be performed in the case that the JPEG files are used.

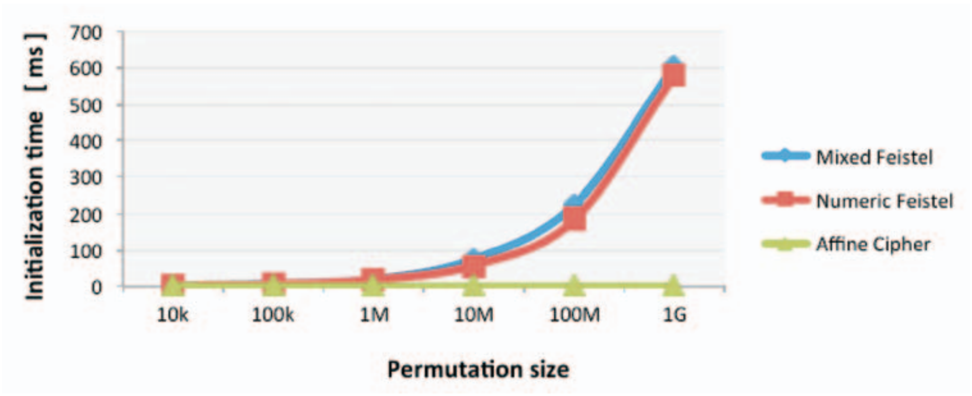

FIGURE 7. Comparison of the permutation initialization time.

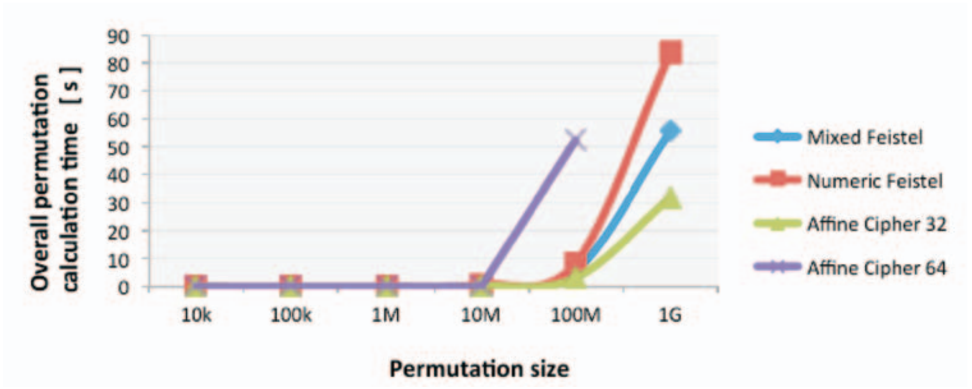

FIGURE 8. Comparison of the overall permutation calculation time.

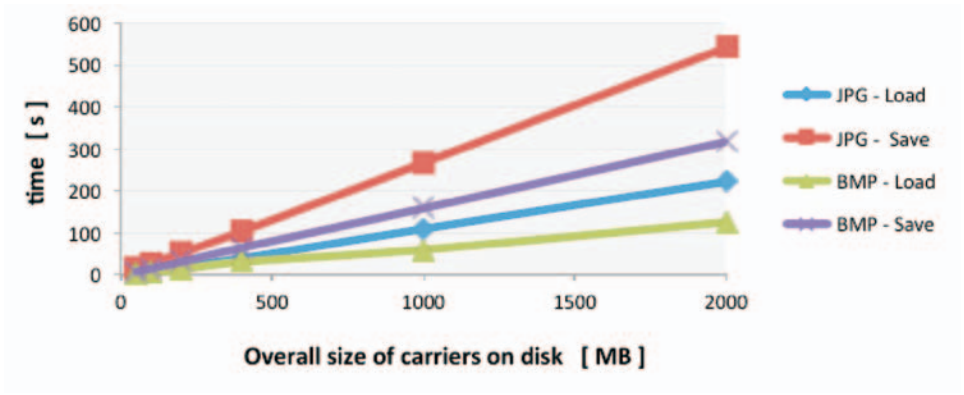

FIGURE 9. Loading and saving of the hidden storage.

5.3. Capacity

As it is shown in Table 5.3, there is not much difference in a capacity offered by our implementations of the JPEG and BMP embedders. Obviously, the selection of the Hamming code used during embedding has a significant impact on the resulting storage capacity. All capacities are in megabytes.

TABLE 1. Capacity of the system composed by cover files with the overall size 1GB.

Hamming code	JPEG	BMP
(7, 4)	47.3	46.6
(15, 11)	31.5	31
(31, 26)	19.7	19.4
(63, 57)	11.8	11.6
(127, 120)	6.9	6.8
(255, 247)	3.9	3.9

6. Conclusion

In this work we were dealing with the steganographic techniques for embedding the secret data in the image files. The primary goal was to design and implement the steganographic layer based on the JPEG files, that can be used directly as the virtual disk or in a connection with existing disk encryption solutions as a hidden storage medium. We carried out an analysis of existing solutions allowing to embed secret messages in JPEG files and existing steganographic file systems. We were also dealing with various techniques of spreading hidden data within the multiple carrier files.

Thanks to the modular architecture of our system, it can be easily extended with new embedding algorithms, coding techniques, permutation generators or other new functionality. Using this approach we prepared the future investigation of various techniques involved in the steganographic file system. Steganalysis of the proposed system is also one of the possible further activities.

REFERENCES

[1] JÓKAY, M.— MORAVČÍK, T,: *Image-based JPEG steganography*, Tatra Mt. Math. Publ. **45** (2010), 65–74.

[2] ANDERSON, R.—NEEDHAM, R.—SHAMIR, A.: *The steganographic file system*, in: Information Hiding, Second International Workshop, IH' 98, Portland, Oregon, USA, April 15-17 (Aucsmith, D. ed.), LNCS Vol. 1525, Springer-Verlag, 1998, pp. 73–82.

[3] MCDONALD, A. D.—KUHN, M. G.: *StegFs: A steganographic file system for Linux*, (A. Pfitzmann ed.), in: LNCS Vol. 1768, Springer-Verlag Berlin, 2000, pp. 463–477.

[4] BERTONI, G.—DAEMEN, J.—PEETERS, M.—VAN ASSCHE, G.: *The Keccak reference*, NIST SHA-3, 2011 (submission).

[5] TUNG HOANG, V.—ROGAWAY, P.: *On generalized Feistel networks*, in: Proceedings of the 30-th annual conference on Advances in cryptology (CRYPTO'10), (Tal Rabin, ed.) Springer-Verlag, Berlin, Heidelberg, 2010, pp. 613–630.

[6] BATEMAN, P.: *Image Steganography and Steganalysis*, Diploma thesis, Faculty of Engineering and Physical Sciences, University of Surrey, Guildford, 2008.

[7] WESTFELD, A.: *F5—a Steganographic Algorithm: High Capacity Despite Better Steganalysis*, in: 4-th International Workshop on Information Hiding, 2001.

[8] NAG, A.—SINGH, J. P.—KHAN, S.—GHOSH, S.—BISWAS, S.—SARKAR, D.—SARKAR, P. P: *A Weighted Location Based LSB Image Steganography Technique*, in: Advances in Computing and Communications, ACC 2011 Conference Communications in Computer and Information Science, Vol. 191, Springer-Verlag, Berlin, 2011, pp. 620–627.

[9] GROŠEK, O.—VOJVODA, M.—ZAJAC, P.: *Classical Ciphers*, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology, Bratislava, 2007.

[10] CHANG, K.: *Knowledge file system* in: A Principled Approach to Personal Information Management, 2010, IEEE International Conference on Data Mining Workshops, 2010.

[11] KATZENBEISSER, S.: *Information Hiding Techniques for Steganography and Digital Watermarking*, Artech House Publishers, London, 1999.

[12] FREE SOFTWARE FOUNDATION, INC.: *The GNU C Library Manual: File Times*, 2013, `http://www.gnu.org/software/libc/manual/html_node/File-Times.html`.

[13] OLIBONI, C.: *OpenPuff*, `http://embeddedsw.net/OpenPuff_Steganography_Home.html`.

[14] PAUL, H.—UPHAM, D.: *JSteg*, `http://zooid.org/~paul/crypto/jsteg`.

[15] HETZL, S.: *StegHide*, `http://steghide.sourceforge.net`.

[16] LATHAM, A.: *JP Hide & JP Seek*, `http://linux01.gwdg.de/~alatham/stego.html`.

[17] PROVOS, N.: *OutGuess*, `http://www.outguess.org`.

[18] ZHANG, W.—ZHANG, X.—WANG, S.: *Maximizing steganographic embedding efficiency by combining Hamming Codes and Wet Paper Codes*, in: Information Hiding, 10th International Workshop, 2008, pp. 60–71.

[19] CHANDRAMOULI, R.—KHARAZZI, M.—MEMON, N.: *Image steganography and steganalysis: Concepts and practice*, in: International Workshop on Digital Watermarking, 2003, (T. Kalker et al. eds.) LNCS Vol. 2939, Springer-Verlag, Berlin, 2004, pp. 35–49.

[20] ZAJAC, P.: *Remarks on the NFS complexity*, Tatra Mt. Math. Publ. **41** (2008), 79–91.

Received September 6, 2013

Department of Applied Informatics and Information Technology
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology
Ilkovičova 3
SK–812-19 Bratislava
SLOVAKIA

E-mail: matus.jokay@stuba.sk
xkosdy@stuba.sk