STEGANOGRAPHIC FILE SYSTEM

XUAN ZHOU

(B.Sc., Fudan University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgement

First and foremost, thank God for granting me the opportunity to pursue my Ph.D. and for His companion in my research.

I would like to express my sincere gratitude to my advisors, Professor Kian-Lee Tan from the National University of Singapore (NUS) and Dr. HweeHwa Pang from the Institute for Infocomm Research (I2R), for their guidance, encouragement, and optimism. Their patience, support, and confidence have been the driving force of this thesis work. Furthermore, I would like to thank Professor Beng Chin Ooi for his kind support during my work in NUS database group.

I am also thankful to the members of my thesis evaluation committees for going through such a long document and giving me valuable feedback. They are Dr. Zhiyong Huang and Dr. Ee-Chien Chang.

I would also like to acknowledge the support and friendship I received from so many friends in NUS over the past 3 years: Cynthia Chen, Jing Dai, Xiaofeng Zhang, Wenjie Zheng, Corrisa Wong, Xiaoyan Yu, Xiaolan Li, Jinghui Qian, YiZhou li, Ming Zhang, Xiaodong Wu, Qingfeng Dou, Xia Cao, Chenyi Xia, ZhengQiang Tan, Gao Cong, Zonghong Zhang, Wee Siong Ng, Hengtao Shen, Bin Cui, Hanyu Li, Rui Zang, Yanfeng Shu, Xi Ma and many others not appearing here. Special thanks go to my former labmate Yingguang Li and my roommate Qi He for dinning and chatting with me everyday. I would also like to thank Sujoy Roy and Chu Yi Liau for giving me so many valuable suggestions in my research work.

I am also grateful to my church friends in Singapore for their love and warm encouragement: Cynthia Chen, Kim Luan Tan, Kim Tok Wong, Daniel Lau, Magdalene Chua, Calvin Chan and others.

Finally, for all the support, love, and understanding they have given me throughout the years, I wish to thank my parents.

Contents

Summary

While user access control and encryption can protect confidential data from unauthorized accesses, they leave evidence of the existence of valuable data, which may prompt an adversary to adopt unconventional tactics to circumvent the protection, such as coercing an authorized user into disclosing his access key. A steganographic file system provides a stronger protection by hiding data's existence. Access to the hidden data is possible only if the correct access key is presented. Without it, an attacker could get no information about whether the hidden data ever exists, even if he understands the system completely. Without knowing the existence of data, adversaries would not be motivated to perform attacks, and many security threats could thus be eliminated. For example, a user under compulsion could plausibly deny that he possesses the data.

However, the practicality of existing steganographic file systems is limited by several factors so that it could not be applied to commercial products that are expected to manage data reliably and efficiently. This thesis is focused on investigating the methodology of designing effective and efficient steganographic file systems for various application environments. First, we construct a new practical steganographic file system that could overcome the weakness of existing systems. Then, we extend the file system from local machines to open network platforms which face higher levels of security threats, and a number of security mechanisms are devised to counter various emerging attacks. We also create a model for steganographic file system that could be used to evaluate its effectiveness in different application environments. We have implemented the proposed systems, and conducted extensive experiments to show their effectiveness and reasonable performance. We believe our research has richly extended the technology of steganographic file systems, and has made it practical for real-world applications.

List of Figures

List of Tables

Chapter 1

Introduction

The advances of the internet and World Wide Web have brought a great innovation to data management technologies. Data is no longer stored locally and processed centrally. On the contrary, data is shared in various forms over the internet. It is distributed among remote storages and processed by remote processors. Thus, researchers begin to explore new methods to manage the huge amount of data shared over the internet, in order to use them more efficiently and safely.

Security is increasingly recognized as a key impediment of the emerging data management technologies, especially when data is shared over the internet and thus exposed to higher risks. Many research projects are in progress addressing various problems on data security, such as remote data access control, copyright protection, privacy protection and trust management. This thesis presents our research on one of the emerging areas – *Steganographic File System*, a system that can provide high confidentiality of data by hiding data's existence.

1.1 Steganographic File System

User access control and encryption are standard mechanisms for protecting data from unauthorized accesses. User access control, which is conventionally enforced by the operating system, enables a data owner to specify who can conduct what operations (i.e. browse, read or write) on which part of his data. Thereafter the operating system grants user accesses according to his specifications. The technology of access control has been well studied and has become very sophisticated. There are a large number of literature [24, 27, 13, 14] addressing its methods, models and implementations. However, data could not always be protected by the access control of operating systems, especially when it is transmitted over networks or stored in public devices such as web cache [29] and shared network storage [44]. When data leaves the protection of access control, it can be encrypted so that it is only accessible to those who are assigned decryption keys. With the prevalence of many internet applications, encryption is increasingly being used to protect data confidentiality [33]. The Encrypting File System (EFS) of MS Windows [16] is a typical example that combines the mechanisms of access control and encryption.

In practice, user access control and encryption can be inadequate when highly valuable data is concerned. Access control could be disabled if adversaries manage to compromise the operating system and access the raw storage directly. In reality, there have been many reports about large systems being cracked by outside hackers or betrayed by inside administrators. Furthermore, a centralized access control is difficult to be established on some distributed systems, e.g. P2P databases [44], DataGrid [1]. While encryption could complement user access control by restricting the access privileges to key holders, the encrypted data itself is the evidence of the existence of valuable data, which would prompt adversaries to attempt to obtain access through some unconventional tactics. For example, attackers could resort to force and compel an authorized user to unlock the encrypted data. Police and government officer could abuse their authorities and require users to disclose the decryption keys. A profligate system administrator could be bribed to release the control of the encrypting system.

To protect data against such unexpected threats, an alternative strategy to building a “super robust” protection around the data is to hide the data so that adversaries could not know that it ever *exists*. Without knowing the existence of data, an adversary would not be motivated to perform attacks, and many security threats could thus be eliminated. For instance, a user under compulsion could plausibly deny the existence of the data. Or he could disclose some less sensitive data such as his address book, but keep silent on more important ones such as the budget of his company. The strategy of data hiding inspires us to create a system that could conceal user selected data automatically so that it remains invisible to adversaries but easily accessible to authorized users.

Steganography, the art of information hiding, offers a way to achieve this desired system. It provides a better protection than cryptography alone – while cryptography scrambles data so it cannot be understood, steganography goes a step further by hiding its very existence. In 1998, Ross Anderson et.al proposed the first prototype of *steganographic file system* [9]. The system hides data files within the physical storage, and grants access to a hidden file only when the correct access key is provided. Without it, an adversary could get no information about whether the data ever exists, even if he understands the software and hardware of the system completely. Following that, a number of constructions of steganographic file system were proposed, and some were implemented into real systems. However, in order to support the steganographic property, these proposals have had to make a number of decisions that compromise the practicality of a file system, resulting in poor processing performance, low effective space utilization and risk of data corruption. We still lack a practical steganograhpic file system that could fulfill the requirements of real-world applications. In addition, the applicability of existing constructions of steganographic file system is limited to personal computers and servers with local storage. With recent technology trends like pervasive computing, peer-to-peer database, data grid, data are increasingly being migrated from local storage devices to shared storage on open networks. These open platforms potentially expose data to higher risks. Deploying a steganographic file system on shared network storage remains an unexplored area.

1.2 Objectives of Research

This thesis aims to investigate the methodology of designing practical steganographic file systems for various applications that are faced with different levels of risks. The specific objectives are classified as follows:

- *A practical steganographic file system:*

  To achieve the ability to hide data, the existing constructions of steganographic file systems have had to make a number of decisions to sacrifice a certain amount of performance, storage space or data integrity. However, they either incur huge performance overhead or waste too much storage space. (Details will be given in chapters 2 and 3.) It is unlikely that these constructions could move beyond niche applications into mass-market commercial file systems that are expected to manage large volumes of data reliably and efficiently. In our research, we attempt to construct a practical steganographic file system that could meet the key requirements of real world applications, without compromising the steganographic property.

- *A model for steganographic file system:*

  Although there have been a number of proposals of steganographic file systems, the application scope of these systems were not clearly defined. A steganographic file system used by a personal computer would be inadequate for a distributed system whose storage is located remotely and protected loosely. In different applications, steganographic file system could be challenged by different threats, which require the system to be constructed accordingly to provide adequate protection for data. Therefore, it is necessary to have a system model to formalize the objective of steganographic file system and to describe the level of risks faced by any particular application environment. Such a model could enable us to construct effective steganographic file systems and to verify whether a construction is adequate (in the senses of security) for a specific application environment. In our research, we attempt to create a model for steganographic file system to meet those demands.

- *Steganographic file systems for open platforms:*

  With the system model, we would like to extend the application of stegnographic file systems from local machine to other various platforms. Recently, some emerging storage technologies such as SAN, DataGrid, P2P data storage have been increasingly used in real applications. As the storage in these platforms are located remotely and shared among the public, deploying a steganographic file system on them would definitely expose the system to higher security threats. Adversaries can easily obtain the access to those shared storage and scour for evidence of hidden data. They could even monitor the activities of the storage device to discover useful information. Thus, previous constructions of steganographic file system would be inadequate for

> a system constructed on those open platforms. In our research, we attempt to propose a number of new system constructions that could defend against the additional threats faced by the open platforms.

In order for the designed steganographic file systems to be practical, we would like them to satisfy the following requirements. First, the system should be able to hide data files securely, so that attacker could not detect the existence of hidden file through any possible attacks and analysis. Second, the system should store data safely, such that data usability would not be easily destroyed by accidents or tampered by attacker. Third, the system should run efficiently and maintain an economical storage space utilization. Actually, to realize the data-hiding function, it would unavoidably impair some other properties of the system, such as performance and data integrity. The impairment need to be limited under a tolerable range, in order to preserve the practicality of the system. As performance is the most important measure of practicality, good performance would be a key objective when we design our steganographic file systems.

1.3 Overview of Contributions

To accomplish the above objectives, we propose a system model and a number of constructions of steganographic file system and experimentally verified their effectiveness and efficient performance.

First, we propose StegFD, a steganographic file system for local machines such as PC and server with local storage. As introduced in chapter 3, it not only overcomes the data loss problems faced by some previous constructions, but also achieves significant improvements in performance and space utilization than the existing constructions. We implemented StegFD into a Linux file system, and conducted experiment to show its practicality for real world applications. We also constructed database components such as B-trees on top of StegFD to demonstrate its potential for database applications.

Second, we create a system model to generalize the objective and design of steganographic file systems. This model divides the activity space of a file system into secure and insecure domains, and defines the objective of steganographic file system as preventing adversaries from detecting hidden data through their observations in the insecure domain. Based on the model, we also propose a set of metrics for measuring the security levels of any steganographic file system. The model and the metrics, introduced in chapter 4, are used in designing the new steganographic file systems.

Finally, to extend the application of steganographic file system, we propose three constructions of steganographic file system for open platforms such as SAN, DataGrid and out-source data storages, which are confronted with higher risks than local/exclusive systems. The first construction, introduced in chapter 5, is created to counter update analysis attack, in which attackers attempt to detect hidden file by observing the updates on the storage. The other two constructions, introduced in chapter 6, are able to counter traffic analysis attack, which is intended to disclose hidden files through monitoring and analyzing the data traffics on the storage. One of the two constructions is unconditionally secure but incurs high overhead. The other is computationally secure and is able to achieve a better performance. We have implemented/simulated the proposed systems, and have conducted intensive experiments to demonstrate their effectiveness and reasonable performance.

We believe that our work has richly extended the technology of steganographic file system, and made it more practical for real-world applications.

1.4 Thesis Organization

Hereby, we outline the organization of this thesis. The rest of this thesis are organized in 6 chapters. Chapter 2 reviews the research works that is closely related to this thesis. They include cyptographic file systems, steganography, steganographic file system and traffic analysis. They form the background knowledge of this thesis.

Chapter 3 introduces the construction of StegFD, a steganographic file system we designed for local machines. We will show through experiments that StegFD achieves significant improvement in both performance and space utilization over existing constructions and satisfies the criteria of a practical file system that is expected to manage data reliably and efficiently. We will also present StegBtree, the B-tree we constructed on top of StegFD, and conduct experiments to demonstrate the efficacy of StegFD in supporting database applications.

Chapter 4 presents a model of steganographic file system. Various examples are given to illustrate how to this model is used on different steganographic file systems designed for different applications. Based on the model, a set of security metrics are also proposed for measuring the level of protection a steganogrpahic file system could offer for hidden data.

In chapter 5, we introduce a construction of steganographic file systems for countering update analysis attacks. It works by conducting dummy updates and relocating data block periodically. Implementation and experiment results will show that it incurs only marginal performance penalties over StegFD and meets the criteria of practical file systems. It is the first step we made to extend steganographic file systems from local machines to open network platforms, such as SAN and DataGrid where the storage could be accessed by attackers repeatedly.

Chapter 6 presents two constructions of steganographic file systems for countering data traffic analysis attacks, which are also potential threats to open network platforms. The first construction is called oblivious storage. It is able to remove all unusual patterns in data traffics, and achieves unconditional security in countering traffic analysis. The second is called DataCavern, which works by reducing the accuracy of traffic analysis to a minimum level. It is computationally secure, but incurs less overhead than oblivious storage. Experiment results will be presented to show their effectiveness and reasonable performance.

Finally, Chapter 7 summarizes the thesis and discuss directions of the future research.

Some of the works in this thesis have been published in several international conferences and journals. The work in chapter 3 has been published in [53] and [54]. The work in chapter 5 has been published in [72]. The work in chapter 6 have been submitted for publication.

Chapter 2

Related Works

This chapter introduces some research works closely related to this thesis. We first give an overview of the existing cryptographic file systems such as EFS for MS Windows and CFS for Unix, and discuss their constructions and functionalities. Then we review the history and the state of art of Steganography, the technique we use to hide data in file systems. Subsequently, we present some existing proposals of steganographic file system and discuss their effectiveness and weakness. Finally, we review current works on traffic analysis, which could be used to secure the steganographic file systems built on open platforms.

2.1 Cryptographic File Systems

While most file systems rely on user access control, which is enforced by operating systems, to protect data from unauthorized accesses, the functions of user access control is limited by particular system construction and actual application environment. In practice, access control is not necessarily able to ensure the security of data. For example, for a personal computer shared among multiple users, it is possible that a user accesses the physical storage device directly when the other

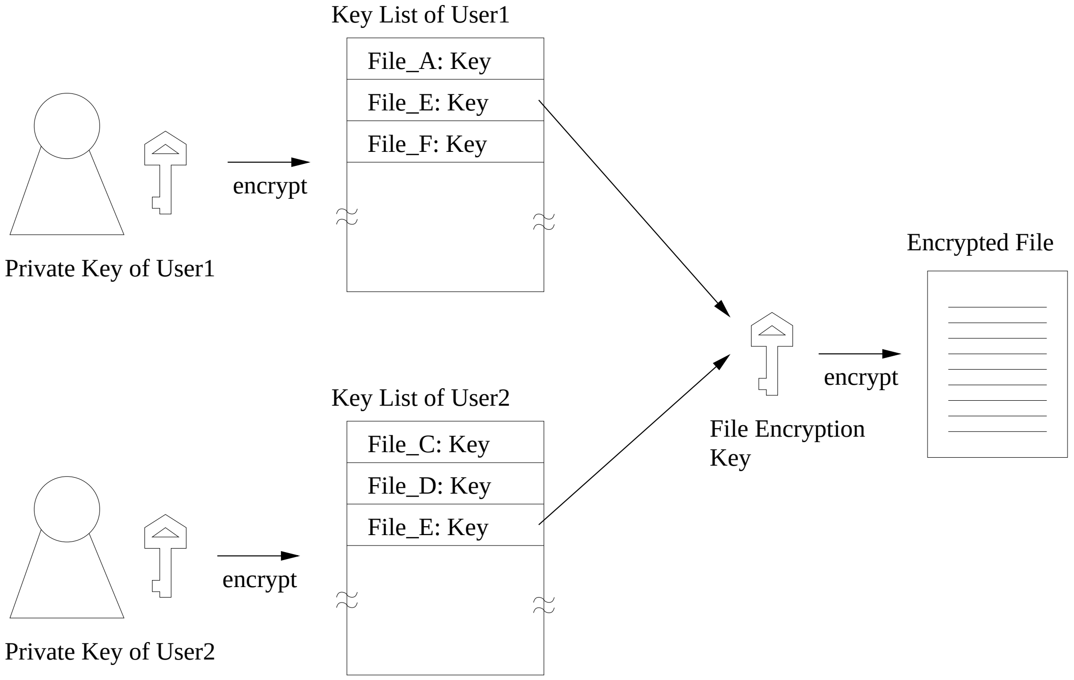


Figure 2.1: EFS of MS Windows

users are not around and steals the others' private data. Laptop and other mobile computing devices are popular today, and they are more susceptible to theft than desktop PCs. Once being stolen, its access control could be easily removed through reverse engineering [39]. In some large systems, data may reside on remote storage (e.g. SAN, out-sourced storage) that is unreachable by the servers' access control. Consequently, it is desirable to encrypt valuable data so that it remains unaccessible to adversaries when access control does not function. A number of cryptographic file systems have been proposed to provide such protection. Examples include the EFS of MS Windows [16] and the CFS of Unix [15].

The Encrypting File System (EFS) of MS Windows enables users to protect data in PCs and Laptops through encryption, in case attackers could bypass the operating system to directly read the hard disk. In EFS, files and directories could be selectively encrypted, and only the cipher text is permanently stored in the

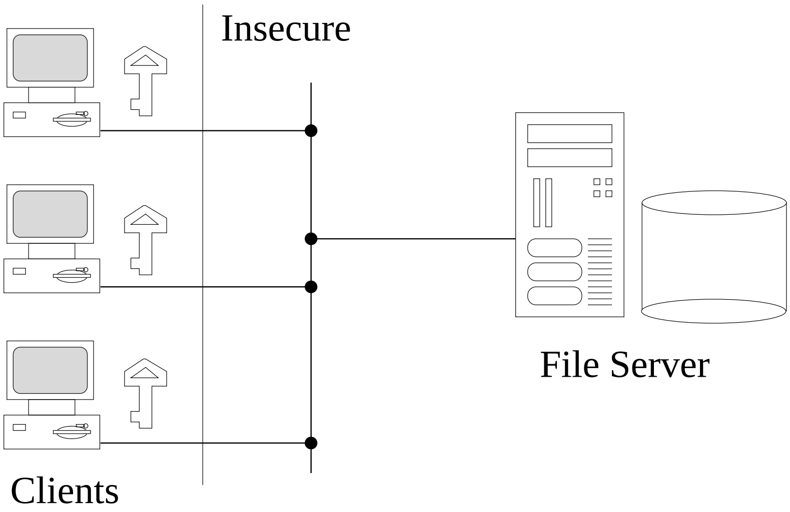


Figure 2.2: CFS of Unix

secondary storage. To facilitate key management, both symmetrical cryptography and asymmetrical cryptography are used. As shown in figure 2.1, files/directories are always encrypted through symmetrical algorithms such as DES (data encryption standard). The file encryption keys, in turn, are encrypted by the public key of each authorized user through an asymmetrical algorithm and kept in the user's key list in the storage. When accessing an encrypted file/directory, a user provides his private key, which could be stored in his smart-card or other private device, to decrypt the corresponding file encryption key from his key list. Then the file could be accessed after it is decrypted by the file encrypting key. Without the private key of authorized users, adversaries are not able to read the file even though they can access the disk directly. The procedure is automatically performed by the file system and is transparent to end users.

In contrast to EFS, the Cryptographic File System (CFS) of Unix is not only used for securing data in PC or laptop, but used for protecting data in a Network File System (NFS) [64]. As the storage of a file server is usually much more capacious and stable than those of client PCs, people would prefer to store their files on server side. A NFS enables users to store files on server side while accessing them just as they are on the client side. However, if users do not trust a remote file server to protect the confidentiality of their data, they would choose to encrypt the files before uploading them to the server. This demand could be met by CFS. As shown in figure 2.2, CFS stores encrypted files on the remote file server, and keeps the encryption keys in the client PCs. When a user requests to access a file, CFS first downloads the file from the server to the client PC, and decrypts it using the encryption key. Once a file is updated by user, CFS encrypts it before updating it on the file server. Files keep being encrypted when they are in the server or being transferred over the network, and thus are resistant to any authorized access from outside the client PC. Besides CFS, there are a number of cryptographic file systems designed for remote file servers, such as TCFS (transparent cryptographic FS) [47, 20], CryptFS [71], SFS (Self-certifying FS) [48]. As their functions are similar to that of CFS, we ignore their detailed constructions in this thesis.

Cryptographic files systems provide a layer of protection for data when access control is unavailable. However, this protection could still be inadequate, as encrypted files alerts adversaries the existence of valuable data, and prompts them to adopt unconventional tactics, such as coercing an authorized user into disclosing the encryption keys. The threats could be overcome by steganography, which intends to provide an extra layer of protection than cryptography by hiding the existence of data.

2.2 Steganography

Derived from a Greek word meaning “covered writing”, *steganography* is about the art of concealing secret message within innocuous looking carriers. Its practice can date back many centuries. In the history [37] by Herodotus (a Greek historian in the 5th century B.C.), to notify Greece the invasion from Xerxes, Demeratus wrote the message on a wood tablet and covered it with wax on which another-innocuous-message was written. Then the tablet passed inspection by sentries without question. An instance of another technique, during the same period, is to shave off the messenger's hairs and tattoo the message on his head. When his hair grows out, the message would be concealed until his head is shaved again. During World War II, the technology of stegonagraphy had a remarkable development in the research of military intelligence, where the emerged techniques include invisible ink [42, 51], microdot [52, 38] and unencrypted cypher [40]. The use of unencrypted cypher is illustrated by the following message, which was actually composed by German spy in WWII.

> *Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by products, ejecting suets and vegetable oils.*

Taking the second letter in each word, it becomes:

> *Pershing sails from NY June 1.*

Steganography is different from cryptography. The latter intends to prevent enemies from interpreting or modifying the secret, while the former aims to prevent enemies from detecting the presence of the secret.

Contemporary steganographic technologies have been focused on digital data, as information are increasingly exchanged in digital forms with the advances of information technology. Many digital steganographic techniques emerged to hide secrets into files of image [41], audio [65] and video [35], which usually contain plenty of room for extra data that will not noticeably affect the end result if someone should choose to view or listen to them. For example, secret information could

Original Image Hidden Image

Figure 2.3: Steganography for Image

be hidden by modifying the insignificant bits of a image without changing its appearance to human eyes. As illustrated in figure 2.3, removing all but the last 2 bits of each pixel of the left image and making the resulting image 85 times brighter results in the image on the right[1]. As an example of application, a copyrighted software could be hidden in images, which are then posted on a Web site or a news group to enable intended recipients to download without leaving evidence to web masters. A positive application of steganography is to help protect copyrights of digital products. Namely, copyright information or serial numbers could be hidden in the digital products through steganographic techniques, so that the producer can later prove his ownership or trace the distribution and reproduction of his products. This is also known as digital watermarking [50, 5, 7]. In contrast to steganography, which purely aims to conceal the embedded information, digital watermarking is more focused on preventing the embedded information from being erased by active attackers.

---

[1] adapted from http://en.wikipedia.org/wiki/Steganography

Steganography and digital watermarking have received great interest from the research community in recent years. The main driving force is the concern over copyright protection of the increasing amount of data published in digital forms. Other applications that drive interest in this area include covert or anonymous communications performed by military and the law enforcement to limit illegal data sharing over the internet. A number of theories [63, 11] and mathematical models [17, 73] have been created for steganography, and many techniques [66, 23, 69] have been proposed in order to hide data more imperceptibly, robustly and efficiently. A good survey on these techniques could be found in [55].

The art of detecting messages hidden using steganography is called *steganalysis* [56, 57], which is comparable to cryptanalysis applied to cryptography. The goal is to identify suspected packages, determine whether or not they have a secret embedded into them, and, if possible, recover the secret. After steganography is applied, some unusual pattern could standout in the hiding data and expose the possibility of hidden information. For example, if the insignificant bits of an image have been used to embed extra information, these bits would become statistically inconsistent with those of a normal image [26]. Then, some statistical analysis could be conducted on the image to disclose the existence of hidden information. On the one hand, steganalysis techniques keep emerging for discovering new statistical artifacts left by information hiding process. On the other hand, steganographic techniques are also improving, and increasing the difficulty of attacks. It seems that their competition would last for a long time, just like that between cryptography and cryptanalysis [10].

In the perspective of information theory, digital steganographic techniques usually utilize the noise contained by a communication channel to hide extra information, such as the least significant bits of an image and audio. The resulting embedding capacity is determined to be restricted under a small limit. Thus, it would be impractical to use them for securing large volumes of data, e.g. dozens of data files. While there have been a number of steganographic systems [2, 3] available on the internet that could be used to secure data files, e.g. DriveCrypt [10] is capable of hiding a entire disk volume in music files, the resulting overhead in storage space is unacceptable for a ideal steganographic file system that needs to hold large volumes of data with high space usage efficiency.

2.3 Steganographic File System

In 1998, Ross Anderson et al. proposed the prototype of steganographic file system which hide data files directly in disk volumes instead of cover data like image and audio. The file system allows a user to associate a password with a file or directory object, such that requests for the object will be granted only if accompanied by the correct password. An attacker who does not have the matching object name and password, and lacks the computational power to guess them, cannot deduce from the snapshot of the raw disk whether the named object exists. Even though it may not be convincing to claim a empty storage device, it is always feasible to disclose some less sensitive files and keep silent on the others, as attacker cannot determine how many data have been hidden in. Such a system could achieve much better space utilization and performance than the classical steganographic methods that use image or music to hide data.

In their paper [9], two constructions of such file system are proposed. The first construction is shown in figure 2.4. It initializes the file system with a number of randomly generated cover files. When a new object is deposited, it is embedded as the exclusive-or of a subset of the cover files, where the subset is a function of

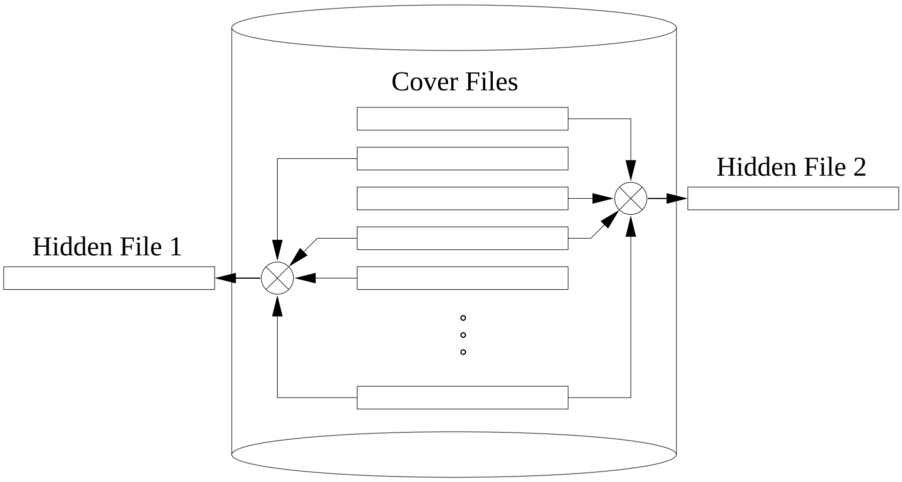


Figure 2.4: Construction of StegCover

the associated password. Without the password, it is computationally infeasible to obtain a correct set of cover files that could construct a hidden object, given a sufficient large number of cover files. Based on their deduction in linear algebra, for a system containing n cover files, more than $\frac{n}{2}$ files could be hidden securely and safely. Compared to the classical steganography techniques, this scheme entails a lower space overhead. However, the performance penalty is very high as every file read or write translates into I/O operations on multiple cover files. (The overhead would be $O(n)$ times of that in regular file systems.)

In contrast, the second construction in [9] encrypts the blocks of a hidden file and writes them to absolute disk addresses given by some pseudo-random process, which is shown in figure 2.5. To reconstruct a hidden file, a user provides the password as the seed to a pseudo-random number generator (PRNG), which in turn generates a sequence of addresses pointing to the data blocks that compose the file. An implementation based on the second scheme was reported in [49]. The problem with this scheme is that different files could map to the same disk addresses, thus causing one to overwrite the other. While the risk can be controlled

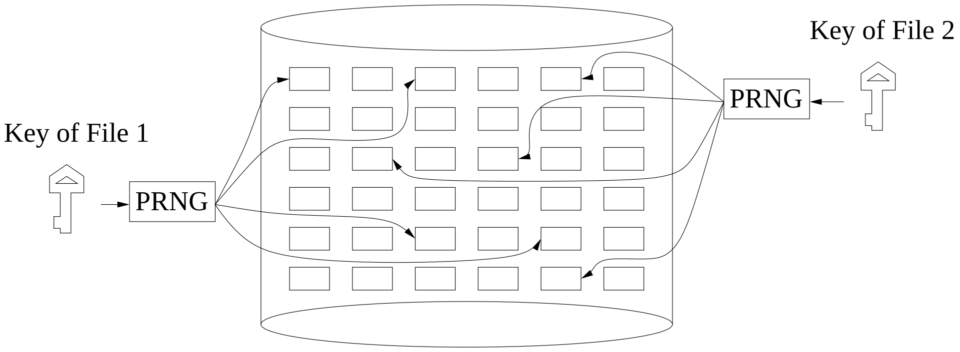


Figure 2.5: Construction of StegRand

by replicating the hidden files and by limiting the number of hidden files, it cannot be eliminated completely, and the resulting storage space utilization has also to be limited to a very low level.

In [34], Hand and Roscoe extended the scheme to on a peer-to-peer platform. In order to provide better resilience against address collision, it utilizes the information dispersal algorithm (IDA) [59] instead of simple replication. Using IDA, a file owner chooses two numbers $m \geq n$ and encodes the hidden file into m cipher-files such that any n of them suffice to reconstruct the hidden file. However, this is achieved at the expense of higher storage and read/write overheads, and there is still the possibility of data loss when more than (m - n) cipher-files get corrupted.

Due to the large performance/space overhead and the risks of data corruption, it is unlikely that these constructions of steganographic file system could move beyond niche applications into mass-market commercial file systems that are expected to manage large volume of data reliably and efficiently.

2.4 Traffic Analysis and Related Techniques

As what will be discussed in chapter 5 and chapter 6, when constructing a steganographic file system on a shared network storage, we need to prevent an attacker, who is monitoring the storage, from detecting the existence of hidden files by analyzing the patterns of the update or data traffic activities. This is the traffic analysis problem [60]. Traffic analysis has been studied extensively in the context of privacy-providing systems – a user would like to reserve his private information while using the system, and an attacker attempts to disclose the information through monitoring and analyzing the data traffics over networks. A typical example is the MIX networks [32, 8], which is intended to enable user to anonymously send message to a recipient. To achieve that, the message is sent through a set of randomly selected nodes in a route, so that the observer cannot determine where is the source or the destination of the message. However, attackers could still be able to reconstruct the route by analyzing the timings and patterns of the network traffics [67]. Then, a number of counter measures were proposed, such as time padding, inserting dummy messages [12], etc.

Some other related techniques that could be adopted to counter traffic analysis include Secure Multi Party Computation (SMPC) [31], Private Information Retrieval (PIR)[22], oblivious RAM [30], oblivious transfers [58] and etc.. While they use different mechanisms to accomplish the peculiar objectives of individual systems, all serve to prevent secret information from being released to adversaries through the data traffic or access patterns. Intuitively, the traffic analysis on steganographic file system would apply steganalysis techniques to data traffics to discover unusual patterns that indicate the existence of hidden data. Thus, the counter measures should be able to remove all the statistically observable artifacts incurred by hidden data from data traffics. Two privacy protection mechanisms that could offer such ability are oblivious RAM and private information retrieval (PIR).

PIR enables users to privately retrieve their information from a secondary storage system, such as a database. With such a mechanism, user data are stored into multiple databases that are not aware of each other, so that a user can retrieve data without revealing them. However, all the existing schemes of PIR [28, 18] only concentrate on reducing the communication complexity, but ignore the I/O overheads. Specifically, most of them need to scan the entire storage volume for every query, and are too expensive for a steganographic file system that is expected to manage data efficiently.

Oblivious RAM is a tamper-resistant cryptographic processor that serves to protect code privacy and prohibit software copyright violation. Even an attacker who can look into the memory and monitor the memory accesses (reads or writes) cannot gain any useful information about what is being computed and how it is being computed. In [30], the oblivious RAM's processing overhead is reduced to O((log t)3) where t is the number of computation steps of the RAM. One of our proposed counter-measures against traffic analysis, oblivious storage (see chapter 6), is inspired by the oblivious RAM.

As the existing techniques on traffic analysis were not specially proposed for steganographic file systems, they usually incur unnecessary cost that would compromise the practicality of a file system. In this thesis, we will propose a number of techniques to deal with the traffic analysis on steganographic file systems.

2.5 Summary

In this chapter, we introduced cyptographic file system, steganographic techniques, existing work on steganographic file system and the related works on traffic analysis. They form the background knowledge of the technique of steganographic file system. Some schemes and methods used in this thesis are actually adapted from them.

Chapter 3

StegFD: A Local Steganographic File System

This chapter introduces StegFD, a local steganographic file systems designed to overcome the weakness of the previous systems like StegCover and StegRand.

3.1 Introduction

There have been a number of proposals for steganographic file systems in recent years [9, 49]. To support the steganographic property, these proposals have had to make a number of design decisions that compromise the practicality of the file systems, resulting in large increases in I/O operations, low effective storage space utilizations, and even risk of data loss as the file system itself could write over hidden files. With such compromises, it is unlikely that the proposed schemes could move beyond niche applications into mass-market commercial file systems that are expected to manage large volumes of data reliably and efficiently.

In this chapter, we introduce StegFD, a scheme to implement a steganographic file system on a local machine, e.g. a personal computer, a server with local storage. StegFD enables users to selectively hide their directories and files. It grants access to a hidden directory/file only if the correct access key is supplied. Without it an adversary would not be able to deduce their existence, even he understands the hardware and software of the file system completely, and is able to scour through its data structures and the content on the raw disks. To ensure its practicality, StegFD is designed to meet three key requirements – it should not lose data or corrupt files, it should offer plausible deniability to owners of protected directories/files, and it should minimize any processing and space overheads. StegFD excludes hidden directories and files from the central directory of the file system. Instead, the metadata of a hidden directory/file object is stored in a header within the object itself. The entire object, including header and data, is encrypted to make it indistinguishable from unused blocks to an observer. Only an authorized user with the correct access key can compute the location of the header, and access the directory/file through the header. We have implemented StegFD on the Linux operating system, and extensive experiments confirm that StegFD indeed produces an order of magnitude improvements in performance and/or space utilization over the existing schemes. We have also extended this StegFD to address how B-trees can be supported in a steganographic file system. We introduce two schemes for implementing steganographic B-trees, and also report a performance study to evaluate the proposed B-tree schemes.

The remainder of this chapter is organized as follows: Section 3 introduces our StegFD file system, together with a discussion on some potential limitations of StegFD and ways to work around them. Section 4 presents our StegFD implementation on the Linux operating system, and profiles StegFD's performance characteristics. In Section 5, we present extensions to StegFD to support B-trees. Finally, Section 6 summarizes this chapter and discusses its further extensions.

3.2 StegFD: Steganographic File Driver

In this section, we present StegFD, a practical scheme for implementing a general-purpose steganographic file system. The scheme is designed to satisfy three key objectives: (a) StegFD should not lose data or corrupt files. (b) StegFD should hide the existence of protected directories and files from users who do not possess the corresponding access keys, even if the users are thoroughly familiar with the implementation of the file system. (c) StegFD should minimize any processing and space overheads.

To hide the existence of a directory/file, it should be excluded from the central directory of the file system. Instead, StegFD maintains the hidden directory/file object's structure, eg. its inode table, in a header within the object itself. Similarly, all records pertaining to the object, for example usage statistics, should also be isolated within the object instead of being written to common log files. The entire object, including header and data, is encrypted to make it indistinguishable from unused blocks in the file system to an unauthorized observer. Only a user with the access key is able to locate the file header and, from there, the hidden directory/file. To simplify the description, we will henceforth focus on hidden files, with the understanding that the discussion applies equally to hidden directories.

3.2.1 File System Construction

Figure 3.1 gives an overview of the StegFD file system. The storage space is partitioned into standard-size blocks, and a bitmap tracks whether each block is free or has been allocated – a 0 bit indicates that the corresponding block is free, while a 1 bit signifies a used block. All the plain files are accessed through the central directory, which is modeled after the inode table in Unix. Hidden files are not reg-

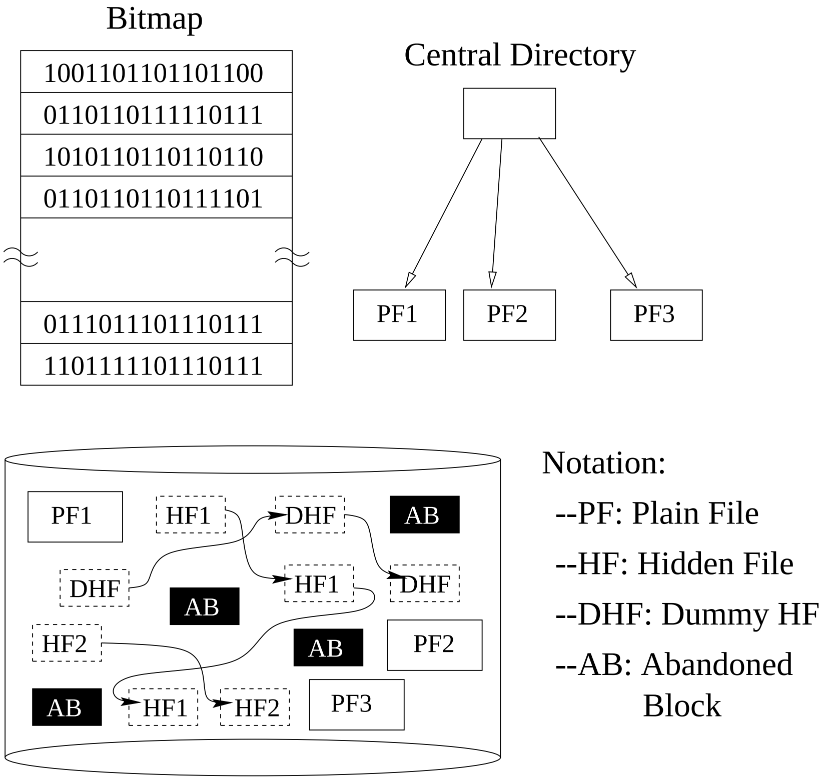


Figure 3.1: Overview of the StegFD File System

istered with the central directory, though the blocks occupied by them are marked off in the bitmap to prevent the space from being re-allocated.

When the file system is created, randomly generated patterns are written into all the blocks so that used blocks do not stand out from the free blocks. Furthermore, some randomly selected blocks are abandoned by turning on their corresponding bits in the bitmap. These abandoned blocks are intended to foil any attempt to locate hidden data by looking for blocks that are marked in the bitmap as having been assigned, yet are not listed in the central directory. The higher the number of abandoned blocks, the harder it is to succeed with such a brute-force examination for hidden data. However, this has to be balanced with space utilization considerations. In practice, the number of abandoned blocks may be determined by an administrator, or set randomly by StegFD.

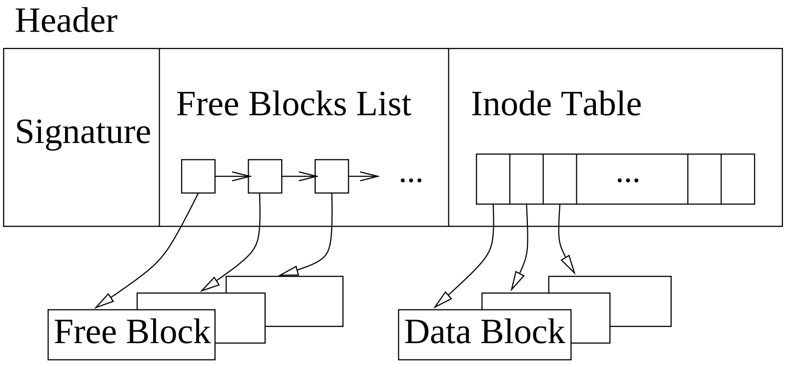


Figure 3.2: Structure of Hidden File

StegFD additionally maintains one or more dummy hidden files that it updates periodically. This serves to prevent an observer from deducing that blocks allocated between successive snapshots of the bitmap that do not belong to any plain files must hold hidden data. The number of dummy hidden files can also be set manually or automatically. Note that dummy files do not eliminate the need for abandoned blocks – whereas dummy files are maintained by StegFD and could be vulnerable to an attacker with administrator privileges, abandoned blocks offer extra protection because they cannot be traced.

In the example in Figure 3.1, the file system contains two hidden user files, a dummy hidden file and three plain files, each of which comprises one or more disk blocks. There are also abandoned blocks scattered across the disk.

The structure of a hidden file is shown in Figure 3.2. Each hidden file is accessed through its own header, which contains three data structures: (a) A link to an inode table that indexes all the data blocks in the file. (b) A signature that uniquely identifies the file. (c) A linked list of pointers to free blocks held by the file.

All the components of the file, including header and data, are encrypted with an access key to make them indistinguishable from the abandoned blocks and dummy hidden files to unauthorized observers.

Since the hidden file is not recorded in the central directory, StegFD must be able to locate the file header using only the (physical) file name and access key. During file creation, StegFD supplies a hash value computed from the file name and access key as seed to a pseudorandom block number generator, and checks each successive generated block number against the bitmap until the file system finds a free block to store the header. Once the header is allocated, subsequent blocks for the file can be assigned randomly from any free space by consulting the bitmap, and linked into the file's inode table. To prevent overwriting due to different users issuing the same file name and access key, the physical file name is derived by concatenating the user id with the complete path name of the file.

To retrieve the hidden file, StegFD once again inputs the hash value computed from the file name and access key as seed to the pseudorandom block number generator, and looks for the first block number that is marked as assigned in the bitmap and contains a matching file signature. The initial block numbers given by the generator may not hold the correct file header because they were unavailable when the file was created. Thus the signature, created by hashing the file name with the access key, is crucial for confirming that the correct file header has been located. To avoid false matches, the file signature has to be a long string. A one-way hash function is used to generate the signature so that an attacker cannot infer the access key from the file name and the signature. Examples of such hash functions include SHA [6] and MD5 [61].

Another characteristic of a hidden file is that it may hold on to free blocks. Here the intention is to deter any intruder who starts to monitor the file system right after it is created, and hence is able to eliminate the abandoned blocks from consideration, then continues to take snapshots frequently enough to track block allocations in between updates to the dummy hidden files. Such an intruder would probably be able to isolate some of the blocks that are assigned to hidden files. By maintaining an internal pool of free blocks within a hidden file, StegFD prevents the intruder from distinguishing blocks that contain useful data from the free blocks. When a hidden file is created, StegFD straightaway allocates several blocks to the file. These blocks, tracked through a linked list of pointers in the file header, are selected randomly from the free space in the file system so as to increase the difficulty in identifying the blocks belonging to the file and the order between them. As the file is extended, blocks are taken off the linked list randomly for storing data or inodes until the number of free blocks falls below a preset lower bound, at which time the internal pool is topped up. Conversely, when the file is truncated, the freed blocks are added to the internal pool until it exceeds an upper bound, wherein some of the free blocks are returned to the file system.

3.2.2 Directory Support for File Sharing

While StegFD incorporates several features to safeguard files that are hidden by a user, it is most effective in a multi-user environment. This is because when many blocks are allocated for hidden files, an attacker may be able to estimate the amount of useful data in these files, but there is no way to ascertain just how much of that belong to any particular user. Hence a user acting under coercion is likely to have a lot of leeway in denying the existence of valuable data that is accessible by him.

One of the natural requirements of a multi-user system is the sharing of hidden files among users. As a user may want to share only selected files, StegFD secures each hidden file with a randomly generated file access key (FAK) rather than the user's access key, so that the file name and FAK pair can be shared among multiple users.

Figure 3.3 depicts the directory structure that StegFD implements to help users

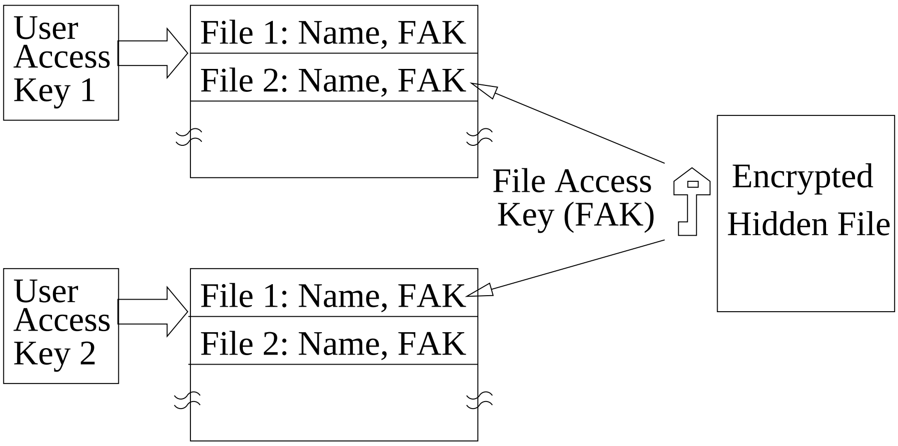


Figure 3.3: Directory Structure of StegFD

track their hidden files. StegFD allows a user to own several user access keys (UAK). For each UAK, StegFD maintains a directory of file name and FAK pairs for all the hidden files that are accessed with that UAK. The entire directory is encrypted with the UAK and stored as a hidden file on the file system. The UAKs could be managed independently, for example stored in separate smart cards for maximum security. Alternatively, to make the file system more user-friendly, UAKs belonging to a user could be organized into a linear access hierarchy such that when the user signs on at a given access level, all the hidden files associated with UAKs at that access level or lower are visible. Thus, under compulsion, the user could selectively disclose only a subset of his UAKs. Without knowing how many UAKs the user owns, the attacker would not be able to deduce that the user is holding back some UAKs.

To share a hidden file with another user, the owner has to release its file name and FAK pair to the recipient. Since neither the owner nor StegFD has the UAK of the recipient, the sharing cannot be effected automatically. Instead, the file information is encrypted with the recipient's public key, and the resulting ciphertext is sent to the recipient, for example via email. Using a StegFD utility, the recipient

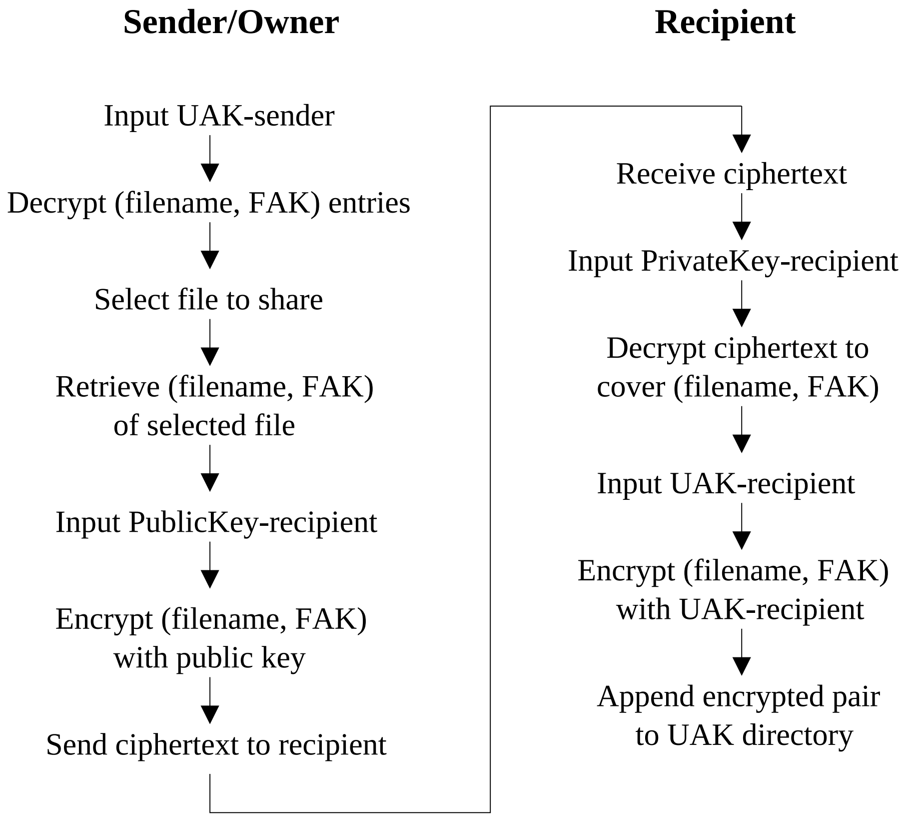


Figure 3.4: File Sharing in StegFD

then decrypts the ciphertext with his private key and associates the hidden file with his own UAK, at which time the file information is added to the UAK's directory and the ciphertext is destroyed. The practice of transmitting the file information is a relatively weak point in StegFD, as the ciphertext could alert an attacker to the existence of the hidden file. However, as each hidden file has its own FAK, a compromised ciphertext does not expose other hidden files in StegFD. The file sharing mechanism is summarized in Figure 3.4.

Finally, when the owner of a hidden file decides to revoke the sharing arrangement, StegFD first makes a new copy with a fresh FAK and possibly a different file name, then removes the original file to invalidate the old FAK. The outdated FAK will be deleted from the directories of other users the next time they log in with their UAKs.

3.2.3 File System Backup and Recovery

Since the hidden files in StegFD are shielded from even the system administrator, the usual method of backing up a file by copying its content no longer works for them. Yet a brute force approach of saving the image of the entire file system would be too time-consuming, in view of the ever-growing capacity of modern storage devices.

StegFD saves the image of only those blocks that are allocated in the bitmap but do not belong to any plain file in the central directory. Plain files are still backed up by copying their content. This limits the overhead of StegFD to the space that is occupied by abandoned blocks, dummy hidden files, and free blocks held within the user hidden files.

To recover a damaged file system, StegFD first restores the image of the abandoned and hidden blocks to their original addresses. This is necessary because the hidden files contain their own inode tables that cannot be adjusted by the recovery process to reflect new block assignments. The plain files are reconstructed last, possibly at new block addresses.

Many existing file systems provide data recovery tools to fix accidental errors. For example, if the file header is lost or corrupted, a regular file system can always track the lost chains and recover the lost file. StegFD can also support recovery by introducing some redundancy: The header of a hidden file can be replicated and placed in pseudo-random locations derived from its FAK. Thus, if the file header is corrupted, the replica can be retrieved to recover the hidden file. Additionally, a signature can be inserted in each data block, so that if necessary a hidden file can be recovered by scanning the disk volume for blocks with matching signatures.

3.2.4 Potential Limitations of StegFD

While StegFD offers an extra feature over a "vanilla" file system in hiding the existence of protected files, this is achieved at the expense of introducing a number of limitations:

- All the hidden files must be restored together; it is not possible to roll-back hidden files selectively. A work-around is to restore all the hidden files to a temporary volume, from where the user can copy the required files over to the permanent StegFD volume.
- The file system is unable to defragment hidden files to improve their retrieval efficiency, without cooperation from the users who possess the file access keys. This is a common problem among secure file system products. A solution is to employ a key recovery mechanism (e.g. [70]) that allows a user to deposit a copy of his UAK with several managers through a secret sharing scheme. To reconstruct the UAK subsequently, concurrence of some minimum number of those managers is needed, thus ensuring the security of the UAK.
- The file system cannot remove hidden files belonging to expired user accounts without cooperation from the users who possess the file access keys. Again, this limitation is common for secure file system products, and can be addressed by a key recovery mechanism.

3.3 System Implementation and Performance Evaluation

This section begins with a description of an implementation of StegFD, then proceeds to present results from some of the more interesting experiments.

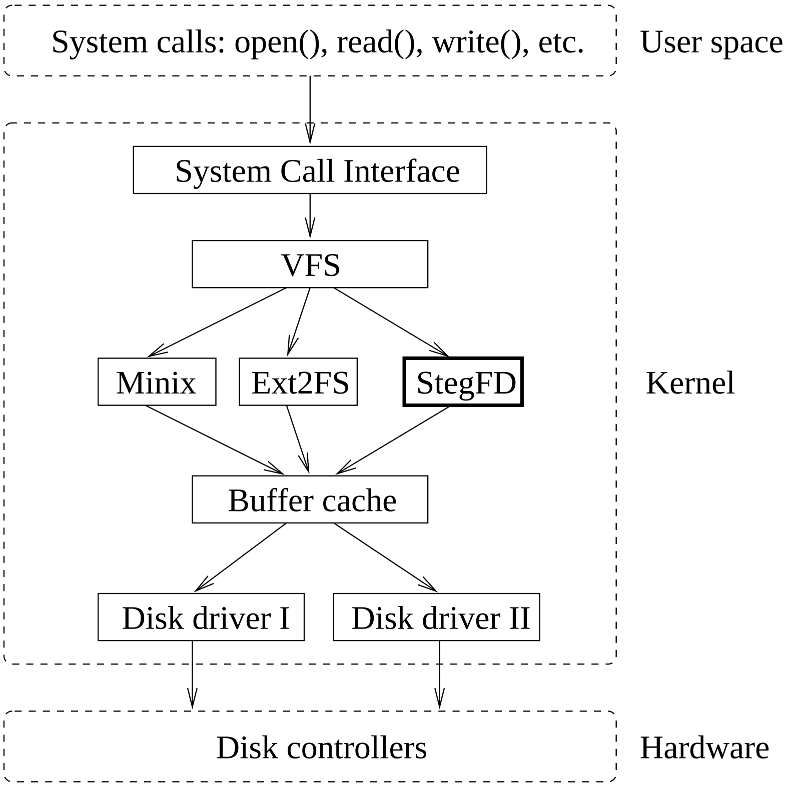


Figure 3.5: StegFD Implementation

3.3.1 System Implementation

We have implemented StegFD on the Linux kernel 2.4; the code is available for public download at the StegFD web site (http://xena1.ddns.comp.nus.edu.sg/SecureDBMS/). We have used SHA256 [6] as the pseudorandom number generator for locating the hidden object (the seed is recursively hashed to generate the pseudorandom numbers), and the block cipher for encrypting data blocks is based on AES [4]. Figure 3.5, adapted from [49], shows the system architecture. It is implemented as a file system driver between the virtual file system (VFS) and the buffer cache in the Linux kernel, alongside other file system drivers like Ext2fs [19] and Minix [68]. StegFD implements all the standard file system APIs, such as open() and read(), so it is able to support existing applications that operate only on plain files. In

Parameter	**Value**
Model of the CPU	Intel Pentium 4
Clock speed of the CPU	1.6 GHz
Type of the hard disk	IBM ATA/IDE
Capacity of the hard disk	60 GB

Table 3.1: Physical Resource Parameters

Parameter	**Default**
Size of each disk block	1 KBytes
Size of each file	(1, 2] MBytes
Capacity of the disk volume	25 GBytes
Number of files in the file system	2000
File access pattern	Interleaved
Number of concurrent users	1

Table 3.2: Workload Parameters

addition, StegFD introduces several steganographic file system APIs for creating hidden directories/files, converting between hidden and plain directories/files, revealing hidden directories/files, and sharing hidden directories/files. Details of the API can also be found at the StegFD web site.

3.3.2 Experiment Set-Up

To evaluate the performance of StegFD, we ran a series of experiments with various workloads on an Intel PC. The key parameters of the hardware are listed in Table 3.1, while Table 3.2 summarizes the workload parameters. Note in particular that we expect many file servers to use a block size of 1 KBytes – the allocation unit is 1 KBytes in NTFS, and 512 Bytes or 1 KBytes in Unix – hence we set that as the default. However, we will also experiment with larger block sizes to study how StegFD would perform with other file systems (the allocation units in FAT16 and FAT32 are 32 KBytes and 8 KBytes, respectively).

For comparison purposes, we shall benchmark against the native file system in Linux and the two schemes proposed in [9] – *StegCover* that hides each file among 16 cover files as recommended by the authors, and *StegRand* that writes a hidden file to absolute disk addresses given by a pseudorandom process and replicates the file to reduce data loss from overwritten blocks (see the Section 3 of Chapter 2). As for the native Linux file system, its performance provides an upper bound to what any file protection scheme can achieve at best; we shall examine two separate cases – *CleanDisk* and *FragDisk*. With CleanDisk, files are loaded onto a freshly formatted disk volume and occupy contiguous blocks; this is intended to highlight the best possible performance limit. In contrast, FragDisk reflects a well-used disk volume where files are fragmented, and is simulated by breaking each file into fragments of 8 blocks.

The primary performance metrics for the experiments are: (a) the effective space utilization, i.e., the aggregate size of the unique data files divided by the capacity of the disk volume; (b) the file access time, defined as the time taken to read or write a file, averaged over 1000 observations (the normalized file access time is the file access time divided by the file size); (c) the CPU consumption, defined as the CPU's non-idle time; and (d) the CPU utilization, defined as the CPU consumption divided by the total elapsed time.

3.3.3 Effective Space Utilization

We begin our investigation with an experiment to profile the space utilization of the steganographic file systems. Here the size of the disk volume is set to 25 GBytes, while the file sizes vary uniformly between 1 and 2 MBytes.

Let us first examine the StegCover scheme. Since the cover files must be big enough to accommodate the largest data file, the most efficient space utilization is achieved by setting the cover files to 2 MBytes. With file sizes in the range of (1, 2] MBytes, each set of cover files can be 50% to 100% utilized, thus giving an average space utilization of 75%. While we can probably improved upon the original StegCover scheme by packing several files into each set of cover files, and by letting large files span multiple sets of cover files, that would introduce indexing complexities and performance penalties, and is beyond the scope of our work.

Turning our attention to StegRand, we note that its resilience against data corruption can be improved by file replication. Its effective space utilization is the space utilization when the first data block is irrecoverably corrupted – that is when StegRand has just passed the limit where it can safely recover all its hidden files, and beyond which more files will be corrupted and lost permanently. As reported in [9], with a replication factor of 4, the space utilization can only reach 7% for a disk with 1,000,000 blocks. Experiments on our disk volume comprising 25,000,000 blocks show that the average space utilization cannot exceed 4% even with a replication factor of 16. It is reasonable that larger storage space produces lower space utilizations since block corruptions occur more frequently in a disk volume made up of more blocks than one with fewer blocks.

Finally, we consider the StegFD scheme. Here, the only storage overheads are incurred by the abandoned blocks, the dummy hidden files, the inode structures, and the free blocks held within the hidden files. Since there is no danger of data blocks being overwritten, all of the remaining space can be used for useful data. Assuming that the percentage of abandoned blocks in the disk volume is 1%, the dummy hidden files occupy another 1% of disk space, and each hidden file contains a maximum of 10 free blocks, StegFD is able to consistently achieve more than 80% space utilization.

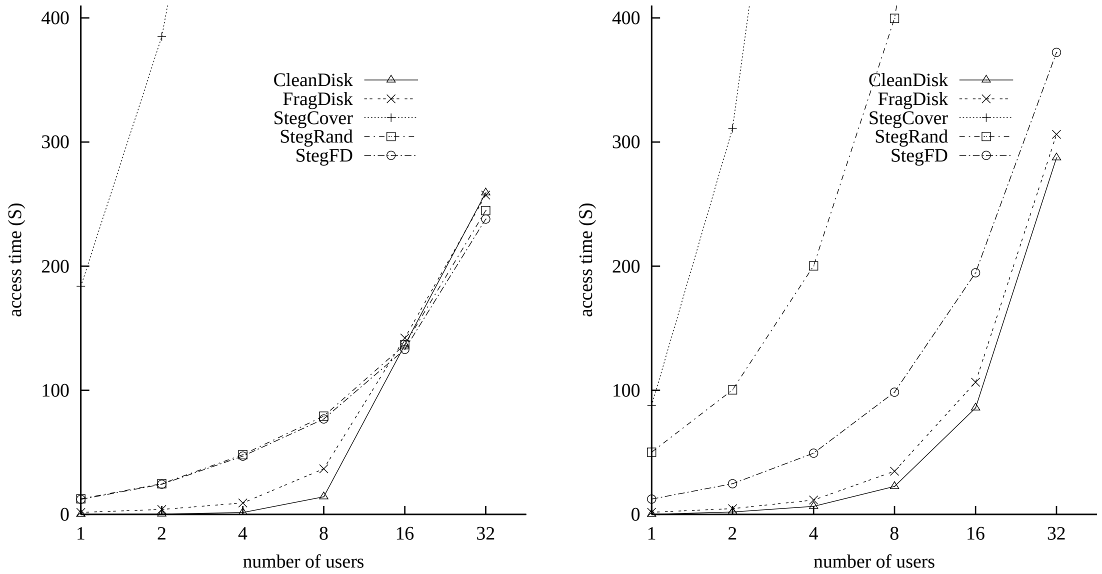
To summarize, we have arrived at a couple of observations. First, the StegCover scheme cannot achieve full space utilization without extending it to perform file packing and spanning. Second, StegRand works reliably only when the disk volume is very sparsely populated; file servers that are typically formatted with a 1 KByte block size can achieve only 4% space utilization for a 25 GByte volume, and less for larger disks, before data corruption sets in. Third, the proposed StegFD is capable of achieving higher space utilizations than StegCover, and is at least 20 times more space-efficient than StegRand.

3.3.4 Performance Analysis

Having demonstrated StegFD's superior space utilization, we now focus on its performance characteristics. This experiment is intended to study how well it works, relative to the native file system and the other steganographic schemes, on file servers where I/O operations from several users or applications are interleaved. For StegCover, the number of cover files is 16, while a replication factor of 4 is used for StegRand, both according to the authors' recommendation in [9]. The disk volume size and the block size are set to 25 GBytes and 1 KBytes, respectively, while the file sizes vary uniformly between 1 and 2 MBytes.

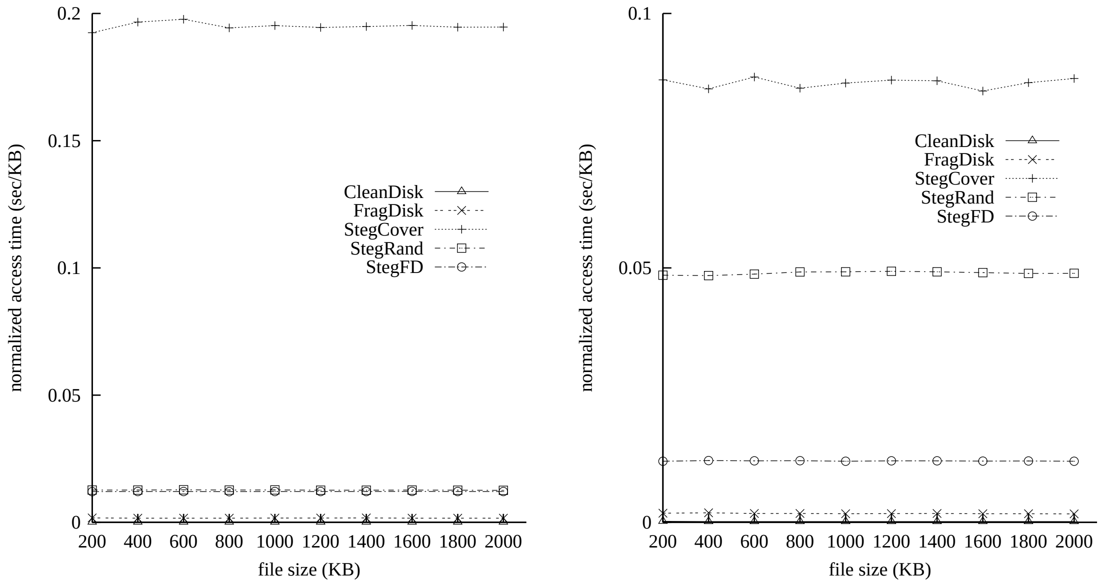
Figures 3.6(a) and (b) give the read and write access times, respectively, for the various file systems. Since StegCover spreads each hidden file among multiple cover files, every file operation translates to several disk I/Os, hence its read and write access times are very much worse than the rest. As for StegRand, its read performance is no better than StegFD's due to the need to hunt for an intact replica when the primary copy of a file is found to be corrupted, whereas the write access times are much worse because all the replicas must be updated.

As for StegFD, its access times are slower than those of CleanDisk and FragDisk under very light load conditions as they produce sequential I/Os on contiguous data



(a) Read (b) Write

Figure 3.6: Sensitivity to Concurrency



(a) Read (b) Write

Figure 3.7: Sensitivity to File Size

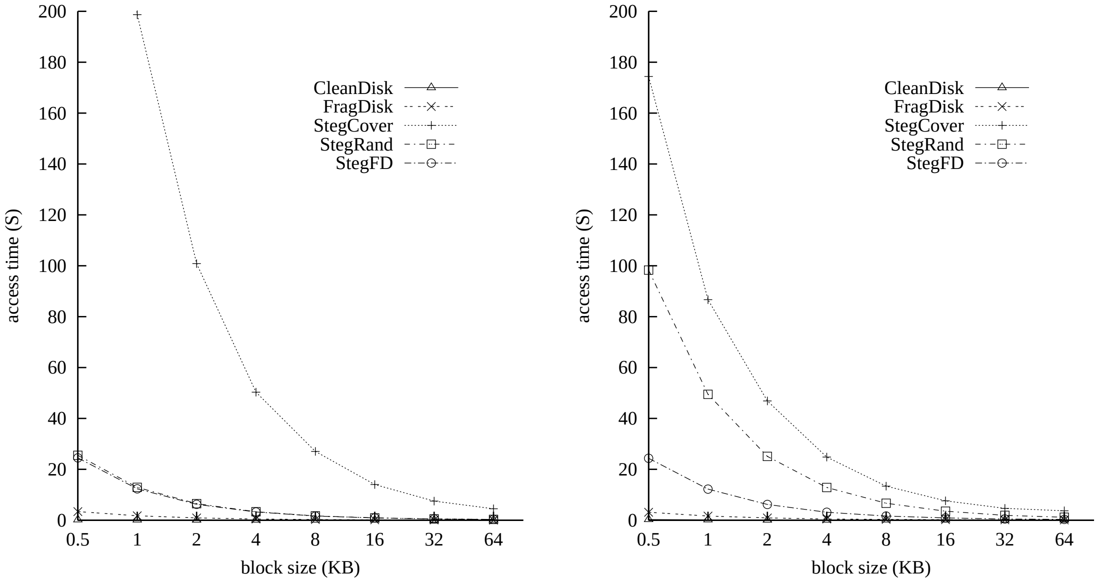
blocks, particularly for read operations that benefit from the read-ahead feature of the disk. However, the differentiation diminishes with increased workload, as file operations become increasingly interleaved. In fact, StegFD matches both CleanDisk and FragDisk from 16 concurrent users onwards for read operations. For write operations, the performance of StegFD also converges toward those of CleanDisk and FragDisk with more concurrent users. Finally, the relative trade-offs between the various schemes are independent of the file size, as shown in Figures 3.7(a) and (b) (for single user context).

In summary, this experiment shows that both of the previous steganographic schemes introduce very high read and/or write penalties and are not suitable for file servers that must handle heavy loads. In contrast, StegFD is a practical steganographic file system that delivers similar performance to the native Linux file system in a multi-user environment.

3.3.5 Sensitivity to File Access Patterns

The next experiment is aimed at discovering the sensitivity of the various file systems' performance to the file access pattern. Specifically, we are looking at a situation where each file is retrieved in its entirety before the next file is opened, as may happen in a very lightly loaded file server. We fix the number of concurrent users at 1, while maintaining the other workload parameters at their settings in the previous experiment.

Figures 3.8(a) and (b) show the read and write access times for the various file systems, with the file size fixed at 1 MBytes. Here, CleanDisk delivers the best performance as expected since all its files occupy contiguous blocks. FragDisk, which breaks each file into fragments of 8 blocks, is slower due to the overhead in seeking to each fragment. This indicates that as the file system gets more



(a) Read

(b) Write

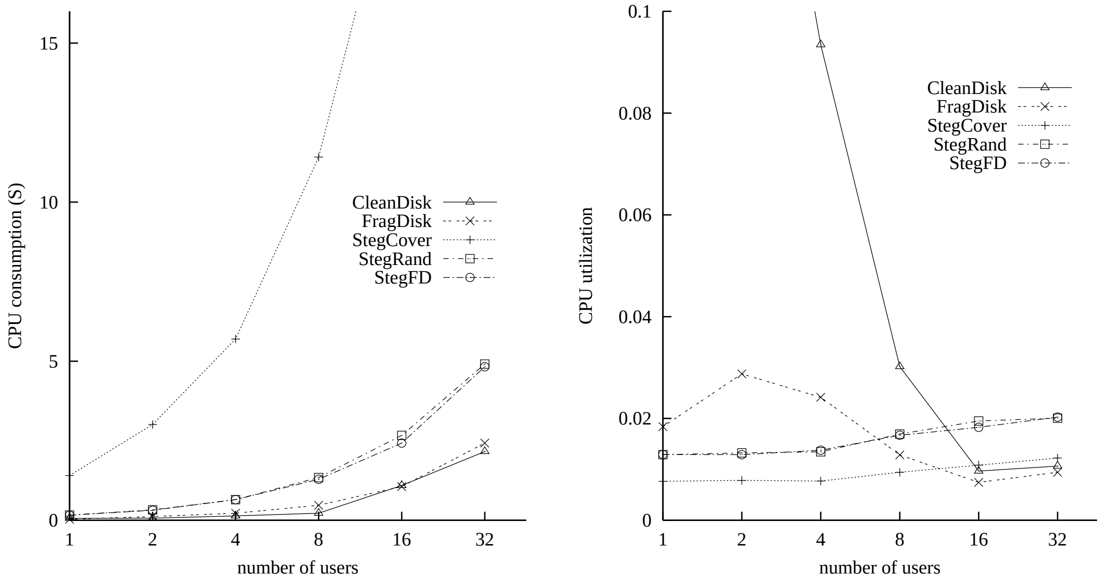
Figure 3.8: Serial File Operations

fragmented, its performance would gradually degrade to that of StegFD even in single-user environments where file operations are not interleaved. The difference in performance is more pronounced with small block sizes where FragDisk has to perform more fragment seeks, and StegFD and StegRand incur more block seeks.

This experiment demonstrates that while StegFD achieves similar performance to the Linux file system in a multi-user environment, the penalty that StegFD incurs in hiding data files is noticeable when the load is so light that file I/Os are not interleaved. Even then, StegFD still delivers acceptable access times and outperforms the previous steganographic schemes significantly.

3.3.6 CPU Usage

The last set of experiments aims to evaluate the CPU usage of the various file systems. We vary the number of concurrent users, and measure the CPU consumption



(a) CPU Consumption (b) CPU Utilization

Figure 3.9: CPU Usage

and utilization for retrieving 1-MByte data files.

As shown in Figure 3.9(a), StegCover has the highest CPU consumption since it needs to retrieve 16 times more data than the other schemes. As StegRand and StegFD need to execute some cryptographic functions in each data retrieval or update, they incur more CPU overhead than CleanDisk and FragDisk. However, at low concurrency, StegRand and StegFD have lower CPU utilizations because their I/O costs are higher than those of CleanDisk and FragDisk. Nevertheless, with the exception of StegCover, the CPU utilizations of the tested file systems are no more than 10% as shown in Figure 3.9(b). This confirms that I/O cost is still the dominant performance determinant.

3.4 Steganographic B-Tree

Having devised a steganographic file system and demonstrated that it incurs only marginal access time and space utilization penalties over conventional file systems, we are keen to investigate its efficacy in supporting specialized applications; in particular, relational DBMSs that must be highly optimized. In this section, we study how efficiently operations can be carried out on B-trees, one of the key index structures in relational DBMSs, within a StegFD volume.

3.4.1 Construction of Steganographic B-Tree

A straightforward way to hide the existence of a database is to install a conventional DBMS on a StegFD volume. This causes the DBMS to store the database, including its B-tree indices, as one or more hidden files that are managed by StegFD. The advantage is that this entails no modification to the DBMS. However, if there is a mismatch in the block sizes of the DBMS and StegFD, StegFD would either need multiple I/O operations to satisfy each node access, or it would fetch more data than necessary each time. Even when the DBMS is configured with the same block size as StegFD, the node boundaries in the DBMS may not align with the block boundaries in StegFD. Hence there is an expected performance degradation. In an attempt to overcome this penalty, we propose two schemes for implementing B-trees directly in a steganographic disk volume.

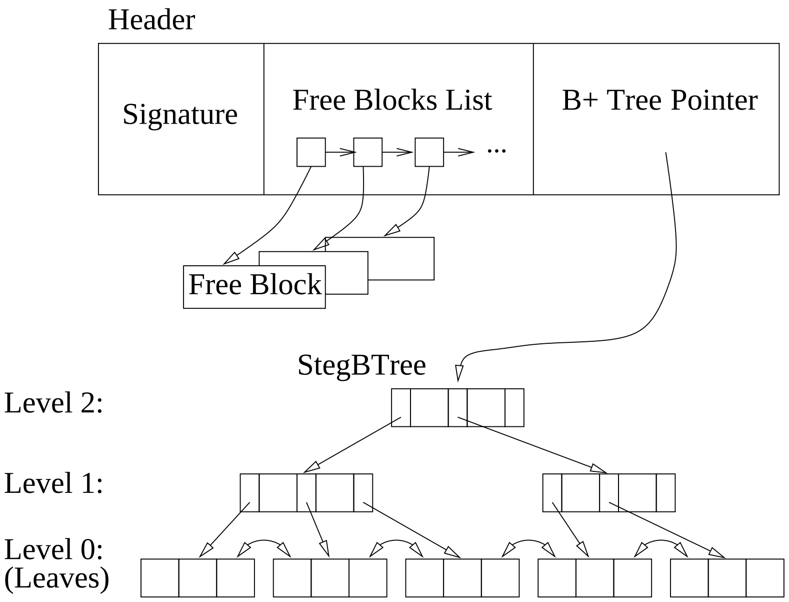
In the first scheme, each B-tree begins with a header as illustrated in Figure 3.10(a). The first two structures in the header, signature and free blocks list, work the same way as with hidden files (see Section 3.1). Unlike a hidden file that links its data blocks in a linear chain, here the index nodes are linked into a B-tree structure. Having located the B-tree through its header, operations like insertion, search and deletion can be carried out according to the usual algorithms. We denote this scheme as StegBtree.

The second scheme for implementing a steganographic B-tree is similar to StegBtree, except that the child pointers in the non-leaf nodes are not stored explicitly. Instead, the address of a node P_i is calculated on-the-fly, by applying a hash function on the corresponding index entry K_i, the node's level number and the file access key, i.e.,

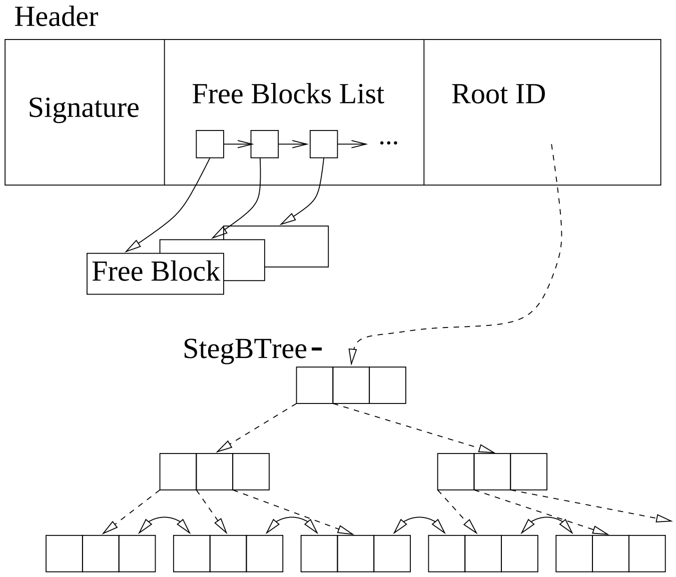
$$\begin{cases} P_0 = HASH(NodeAddress, level\#, FAK) \\ P_i = HASH(K_i, level\#, FAK) \qquad \text{for all } i > 0 \end{cases}$$

where NodeAddress is the physical address of P_0's father node. The address of the root node is calculated by applying the hash function to the root id, which is recorded in the file header. Address collisions that may be encountered by the B-tree nodes are handled the same way as with file headers in StegFD. This pointer-less scheme, StegBtree-, is shown in Figure 3.10(b). The space saving from omitting the child pointers allows each non-leaf node to hold more keys, leading to a higher fan-out and fewer nodes, which can potentially speed up operations on the B-tree.

Algorithms for node allocation, search and insertion on StegBtree- are given in Figure 3.12. Function *allocate()* allocates a new node to StegBtree-. It repeatedly applies a hash function on the input arguments until a free page is found, and returns this page as the new node. Function *locate()* makes use of the same hash function and the same procedure as *allocate()* to locate an existing node from the storage space. The procedure *search()* for StegBtree- is similar to that of a regular B^+-tree, except that it does not use pointers to locate tree nodes, but uses the function *locate()* to calculate the node addresses instead. The procedure *insert()* employs a similar insertion algorithm as B^+-tree, except that it calls the *allocate()*



(a) StegBtree



(b) StegBtree-

Figure 3.10: Structure of StegBtree(-)

```
func allocate (K, level#, FAK) returns address
P = HASH (K, level#, FAK);
loop
    if Block *P is a free block, then
        return P;
    else,
        P = HASH (P);
end loop;
endfunc

func locate (K, level#, FAK) returns address
P = HASH (K, level#, FAK);
loop
    if Block *P's signature is correct, then
        return P;
    else,
        P = HASH (P);
end loop;
endfunc

func search (nodeaddress, K) returns address
// level# is the current level number;
// $K_m$ is the last entry in this node;

if *nodeaddress is a leaf, return nodeaddress;
else,
    if $K < K_1$ then
        P = locate (nodeaddress, level#-1, FAK);
    else if $K \geq K_m$ then
        P = locate ($K_m$, level#-1, FAK);
    else,
        find i such that $K_i \leq K < K_{i+1}$;
        P = locate ($K_i$, level#-1, FAK);
    return search (P, K);
endfunc

proc range_search ($K_1$, $K_2$, (out) results)
$P_1$ = search (root, $K_1$);
begin from $P_1$, and follow the leaf link list until get
$P_2$ which contains the 1st entry greater than $K_2$;
add all the leaf nodes between $P_1$ and $P_2$ to results;
endproc
```

Figure 3.11: Algorithm: Search StegBTree-

```
proc insert (nodeaddress, entry, newchildentry)
// insert 'entry' into subtree with root '*nodeaddress';
// degree is d; 'newchildentry' is null initially, and
// null upon return unless child is split;
// level# is the current level number;

if *nodeaddress is a non leaf node, say N,
    if $K < K_1$ then
        P = locate (nodeaddress, level#-1, FAK);
    else if $K \geq K_m$ then
        P = locate ($K_m$, level#-1, FAK);
    else,
        find i such that $K_i \leq K < K_{i+1}$;
        P = locate ($K_i$, level#-1, FAK);
    insert(P, entry, newchildentry);
    if newchildentry is null, return;
    else,
        if N has space,
            put newchildentry on it,
            set newchildentry to null, return;
        else,          // split N:
            first d key values stay,
            N2 = allocate ($K_{d+1}$, level#, FAK),
            last d keys move to new node N2;
            newchildentry = $< K_{d+1} >$;
            if N is the root,
                $A_0$= allocate (New Root ID, level#+1, FAK),
                insert $< K_{d+1} >$ into *$A_0$;
                replace Root ID with New Root ID;
                // relocate the 1st node of each level:
                B = nodeaddress;
                for i = level# to 0, loop
                    $A_1$ = allocate ($A_0$, i, FAK);
                    copy *B to *$A_1$, release *B ;
                    B = locate (B, i-1, FAK), $A_0$=$A_1$;
                end loop;
            return;

else if *nodeaddress is leaf node, say L,
    if L has space,
        put entry on it and return;
    else,
        split L: first d entries stay,
        L2 = allocate ($K_{d+1}$, 0, FAK),
        rest move to brand new node L2;
        newchildentry = $< K_{d+1} >$;
        return;
endproc
```

Figure 3.12: Algorithm: Insert a Node in StegBTree-

Parameter	Default
Table size	35,000 Tuples
Tuple size	256 Bytes
Node size	4 KBytes
Key size	16 Bytes
Pointer size	4 Bytes

Table 3.3: B-Tree Parameters

function to create new nodes for the B-tree. As Figure 3.12 shows, when a node is split during insertion, the middle entry is passed to the *allocate()* function to create a new node, and thereafter all the index entries in the original node with larger key values than the middle entry are shifted to the new node. As all the existing nodes of StegBtree- remain unchanged during insertion, it does not incur extra overhead. Only when the root node is split and the tree grows up a level, it takes a bit more effort to reorganize the StegBtree-. In that case, a new root node is allocated by passing a new root id to the *allocate()* function. The update of root id requires the first node of each level of the StegBtree- to be reallocated accordingly, as its address is directly or indirectly determined by the root id through the hash function.

To provide native support for B-tree indices in StegFD, we have added two new sets of APIs, one for StegBtree and the other for StegBtree-. The APIs can be found at the StegFD web site (http://xena1.ddns. comp.nus.edu.sg/SecureDBMS/).

3.4.2 Experiments

To investigate the efficacy of *StegBtree* and *StegBtree-*, we compare them with the alternatives of (a) constructing the B-trees directly on a raw disk (Btree), and (b) storing the B-trees in hidden files on a StegFD volume (Btree on StegFD). Table 3.3 summarizes the experiment parameters. The physical resource and workload parameters remain the same as in Tables 3.1 and 3.2.

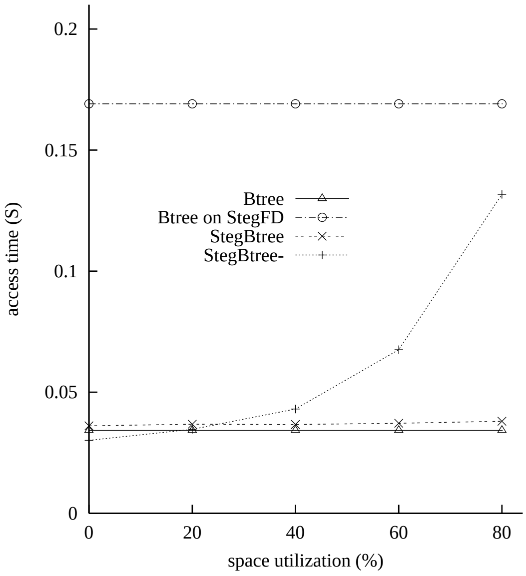
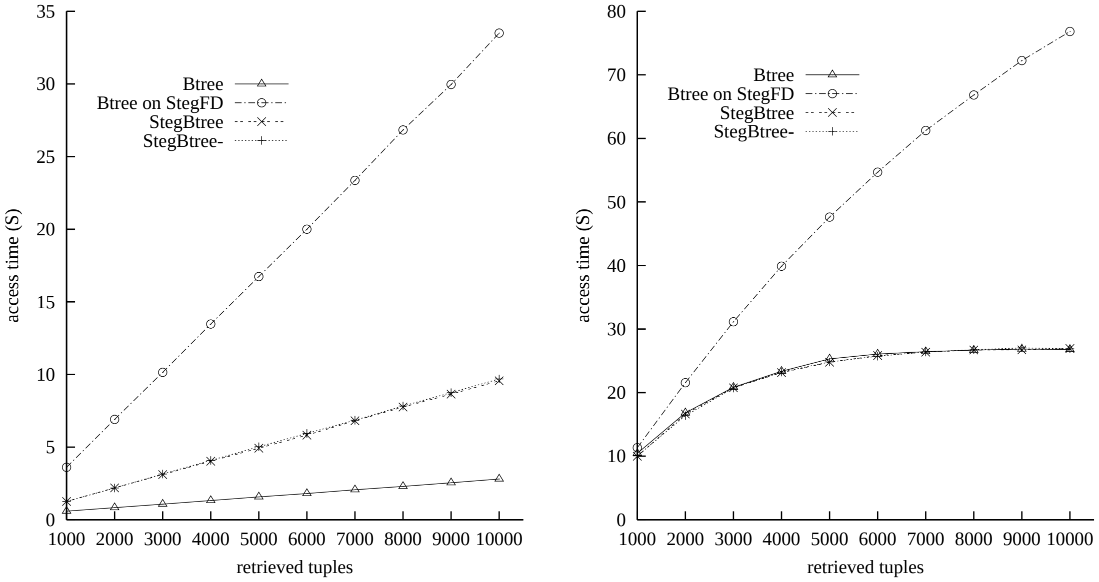


Figure 3.13: Sensitivity to Space Utilization

Sensitivity to Space Utilization

We begin the profiling of the steganographic B-tree schemes by evaluating their sensitivity to the utilization level of the StegFD volume. Figure 3.13 shows the average access time of 400 exact-match queries for the various B-tree schemes.

As expected, *Btree on StegFD* is much slower than the other schemes because it has a different node size from StegFD's block size, and the node boundaries are not aligned with StegFD's block boundaries, thus incurring multiple I/O operations for each node access. For *StegBtree*, there is some overhead in processing the header block to locate the B-tree, but the resulting penalty over *Btree* is well within 20%. In contrast, *StegBtree-* performs just as well as *Btree* initially because the former's larger fan-out and hence shorter height compensate for the I/Os on the header block. However, higher space utilizations lead to more frequent address collisions, and the extra I/Os in tracking down index nodes cause performance to degrade rapidly beyond 40% utilization.



(a) Clustered (b) Unclustered

Figure 3.14: Sensitivity to Query Selectivity

This experiment confirms that native support for B-tree should be built into StegFD. Among the two steganographic B-tree schemes, *StegBtree-* is ideal for sparsely populated volumes, whereas *StegBtree* consistently achieves performance that is just marginally slower than *Btree.*

Sensitivity to Query Selectivity

The second set of experiments is intended to study the behavior of *StegBtree* and *StegBtree-* with range queries. Here we vary the query selectivity from 1000 tuples to 10000 tuples. Figures 3.14(a) and 3.14(b) give the results for clustered and unclustered indices, respectively.

For clustered indices, *Btree* is clearly the fastest, especially at high selectivity factors where data access time dominates index access time. This is because *Btree* benefits from sequential I/Os as data pages are stored at contiguous addresses,

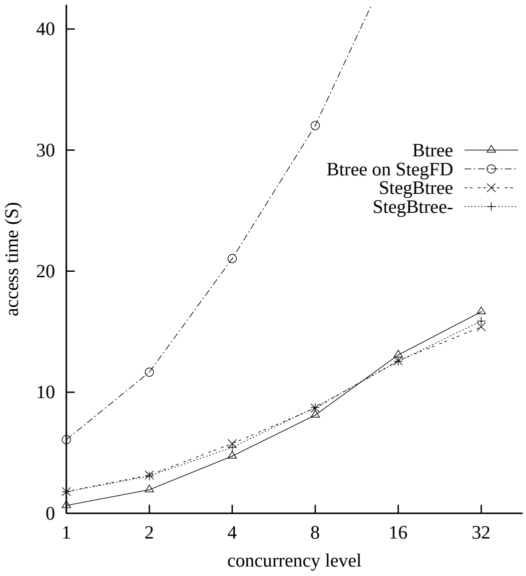


Figure 3.15: Sensitivity to Concurrency

whereas the other three schemes incur random I/O operations. However, for unclustered indices, *Btree* has no advantage over *StegBtree* and *StegBtree-*. Finally, we observe that *Btree on StegFD* is still the worst performer.

Sensitivity to Concurrency

Having discovered that *Btree* can be superior to the steganographic B-tree schemes, we are interested to find out whether this relative performance still holds in a multi-user environment. Instead of issuing queries one after another as in the earlier experiments, we now generate multiple range queries (for 2000 tuples each) concurrently on a clustered index. Figure 3.15 plots the access time against the number of concurrent queries.

As shown in the figure, increased concurrency slows down all of the schemes. Moreover, the access time of *Btree* gradually approaches those of *StegBtree* and *StegBtree-*. This is due to the larger amount of random I/O operations when queries are interleaved. Hence, in practice, *StegBtree* and *StegBtree-* are likely to fare favorably relative to *Btree*, and even clustered B-trees.

3.5 Summary

In this chapter, we have introduced StegFD, a practical scheme to implement a steganographic file system that offers plausible deniability to owners of protected files. StegFD securely hides user-selected files in a file system so that, without the corresponding access keys, an attacker would not be able to deduce their existence, even if the attacker understands the hardware and software of the file system completely, and is able to scour through its data structures and the content on the raw disks. StegFD achieves this steganographic property while ensuring the integrity of the files, and maintaining efficient space utilization at the same time. We have also proposed two schemes for implementing Steganographic B-trees in a StegFD volume.

We have implemented StegFD as a file system driver in the Linux kernel 2.4. Extensive experiments on the system confirm that StegFD is capable of achieving an order of magnitude improvements in performance and/or space utilization over the existing steganographic schemes. In fact, StegFD is just as fast in a multi-user environment as the native Linux file system, which is the best that any file protection scheme can aim for.

However, the applicability of StegFD is limited to local systems such as desktop PCs, laptops and traditional application servers whose storage is protected locally. In these platforms, data storage would only be temporarily exposed to adversaries. With the advances of internet and the emergence of new technologies like pervasive computing, data are increasingly being migrated from local storage devices to shared storage on open networks. The shared storage would be contiguously inspected by adversaries, and expose the existence of hidden data through other avenues, such as the I/O activities. Therefore, the scheme of StegFD would be unable to secure hidden data in those shared storages. In the rest of this thesis, we will address the problem of designing adequate steganographic file systems for shared storage on open platform.

Chapter 4

A Model for Steganographic File System

StegFD, the steganographic file system introduced in last chapter, is designed for personal computers and servers with local storage. It is not necessarily applicable to platforms other than local systems, such as a distributed storage, which is faced with additional security threats. (As what will be introduced shortly, it is vulnerable to attacks such as update analysis and data traffic analysis.) Although there have been a number of proposals of steganographic file system before StegFD, no study has been conducted on their applicabilities in different application environments. Cryptographic file systems, such as EFS and CFS, have been designed accordingly to meet the requirements of different applications. Steganographic file systems are also challenged by different security threats in different application environments. They have to be constructed accordingly to provide sufficient protection for hidden data. This chapter introduces a security model to generalize the objective and design of steganographic file system, so that we could construct adequate file system for platforms that are exposed to different levels of risks. The model also enables us to measure the effectiveness of a system construction under different security threats.

The rest of this chapter is organized as follows. Section 1 presents the model we proposed for steganographic file system. Section 2 analyzes the potential threats to a steganographic file systems through the model and proposes a metric to measure the security levels provided by different system constructions. Summary is given in Section 3.

4.1 System Model

In a typical model for network security, activity space is divided into secure and insecure domains. A user encrypts and decrypts data in systems located in the secure domain, and the encrypted data is transmitted over the insecure network, so that the data remains unaccessible to outside attackers. Analogously, our model of steganographic file system also divides the activity space of a file system into secure and insecure domains, as shown in figure 4.1. A user is located within the secure domain, and the interactions between him and the file system in this domain cannot be observed or interfered by adversaries. An attacker located in the insecure domain is able to monitor the information and/or activities exposed in this domain. The file system, accessible to both users and attackers, stretches across both domains. In this model, the user attempts to hide data within the file system so that its existence cannot be detected by the attacker, while the attacker endeavors to collect as many evidences as possible from the insecure domain to prove the existence of hidden data.

The division of the activity space between the secure domain and the insecure domain is determined by the specific application environment of the steganographic file system. The space could be divided from (i) the dimension of geographic location (some parts of the system are potentially exposed to attackers and the other parts can be well protected) or (ii) the dimension of time (during some period

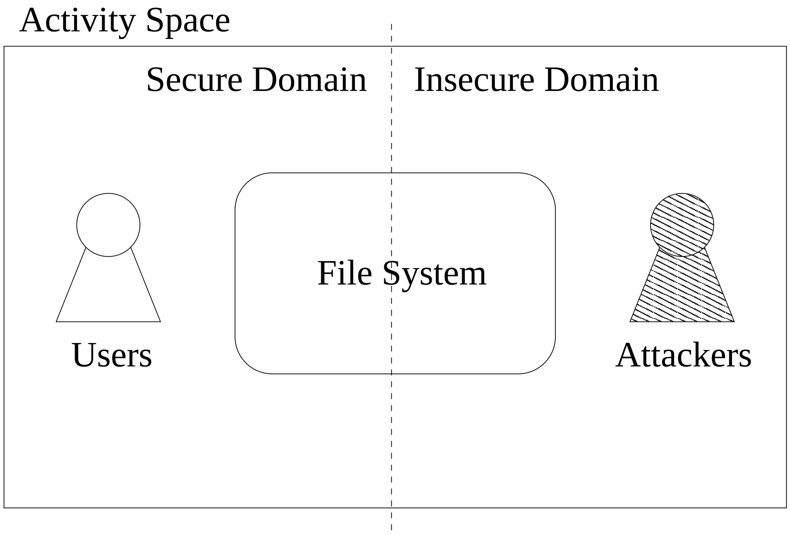


Figure 4.1: Model of Steganographic File System

of time the system is suspected to be exposed to attackers, and at the other time it is well protected.)

Example 1. If a steganographic file system is constructed on a personal computer to protect the owner's privacy from accidents like theft or robbery, the activity space of the system is actually divided from the dimension of time. That is to say, when the PC is in the control of the owner, it is considered be located in the secure domain. At that time, the owner can create and access his hidden files freely without worrying about being inspected by attackers. Contrarily, when the PC is not attended by the owner but stolen or snatched by attackers, it is entirely placed in the insecure domain, where attackers could directly access the secondary storage to look for evidence of hidden files. Thus, a steganographic file system for PC should automatically clear up the evidences that could indicate the existence of hidden files whenever the owner finishes using them. Actually, StegFD fits in with this scenario, for it is designed for personal computers.

Example 2. If the steganographic file system is constructed on a shared network storage, the division of its activity space would be based on geographic location. As the shared storage is not trusted by users to protect the confidentiality of their data files, it would be considered being permanently placed in the insecure domain. Attacker could monitor the operations on the storage at run time to discover the evidence of hidden files. Thus a steganographic file system on shared storage should be able to prevent the activities of the storage device from exposing any information about hidden files. In chapter 5 and chapter 6, we will address the design of steganographic file system on shared network storages.

4.2 Threats and Security

As stated previously, the primary function of a steganographic file system is to hide the protected data files in the physical storage such that attacker cannot determine whether the files have ever existed. To simplify the subsequent discussion, we assume that the file system is only threatened by passive attacks, in which adversaries attempt to prove the existence of hidden files through their observations in the insecure domain. We omit the discussion on active attacks in this thesis. Although active attacks, aiming to modify data files, are also possible, they could be handled by conventional cryptographic methods, e.g. digital signature.

In passive attacks, attackers would collect as many as possible evidences to prove the existence of hidden files. As mentioned previously, in different application environments, the division of activity space between secure and insecure domains would be defined differently. Attackers could therefore observe different evidences. For a steganographic file system constructed on a PC, when it is snatched an attacker, the attacker could look for evidences from the whole system. The contents in the raw storage, either plain or encrypted, and the activity log that records the past operations on the hidden files could all help attackers in estimating the existence of hidden files. In contrast, for a steganographic file system constructed on a shared storage, all the I/O traffics on the storage could be inspected and analyzed by attackers to detect the existence of hidden file. To design an effective steganographic file system, we have to assume that all the information exposed to the insecure domain is available to attackers, and construct a system such that these information is not sufficient for proving the existence of hidden files.

From the perspective of an attacker, proving the existence of hidden files is a decision making process. He sets up a deterministic function which takes as input his observation in the insecure domain and produces as output a decision on whether there is any hidden file. Usually, if the probability of a correct decision is sufficiently high, e.g. $\geq 90\%$, an attacker would regard his attacks successful. Therefore, the security of a steganographic file system could be measured by the accuracy attackers could achieve in determining whether the system contains hidden files. The more accurate the decision made by attackers, the less secure the file system. For instance, it is reasonable to assume that in the sample space, 50% of the steganographic file systems contain hidden files and the other 50% do not contain hidden files. Then, even attackers' decision is based on random guess, it could be 50% accurate. For a steganographic file system, if no deterministic function attack can achieve an accuracy of more than 50%, the system would be extremely secure. On the contrary, if a particular attack can 100% accurately identify the file systems with hidden files, the steganographic file system is actually useless.

Theoretically, Suppose I denotes the observation obtained by attackers in the insecure domain, and F denotes the state indicating whether the file system contains hidden files, i.e. $F = true$ if hidden file exists, $F = false$ if no hidden file. P is the function of probability. Attackers' objective is to estimate $P_{F|I}$. If

$P_{F=true|I=i} > 50\%$, when the observation is i, he will determine there is hidden files. Otherwise, he will determine there is no hidden file. According to Bayesian theory,

$$P_{F=true|I} = \frac{P_{I|F=true}P_{F=ture}}{P_{I|F=true}P_{F=ture} + P_{I|F=false}P_{F=false}} \tag{4.1}$$

If we assume that $P_{F=true} = P_{F=false} = \frac{1}{2}$, then

$$P_{F=true|I} = \frac{P_{I|F=true}}{P_{I|F=true} + P_{I|F=false}} \tag{4.2}$$

Equation 4.2 indicates the accuracy of attackers' decision is determined by how his observation is affected by the existence/non-existence of hidden files, namely $P_{I|F=true}$ and $P_{I|F=false}$.

Theorem 4.2.1. *Suppose $F(I)$ is the deterministic function an attacker used to decide the existence of hidden files through his observation. If and only if there exists an observation i such that $P_{I=i|F=true} \neq P_{I=i|F=false}$, there must be a deterministic function $F(I)$ that could make more accurate decision on whether a file system contains hidden files than random guess.*

Proof of Theorem 4.2.1. Since $P_{F=true} = P_{F=false} = \frac{1}{2}$, the highest accuracy of random guess is 50%. According to equation 4.2, attacker's deterministic function could be

$$F(i) = \begin{cases} true \quad if \quad P_{I=i|F=true} > P_{I=i|F=false} \\ false \quad if \quad P_{I=i|F=true} < P_{I=i|F=false} \end{cases}$$

$F(I) = 50\%$, if only if $P_{I=i|F=true} = P_{I=i|F=false} = 50\%$ for all is. Otherwise, $F(I) > 50\%$, and this function could always make more accurate decision than random guess. □

Theorem 4.2.1 points out that if the existence of hidden file would affect the

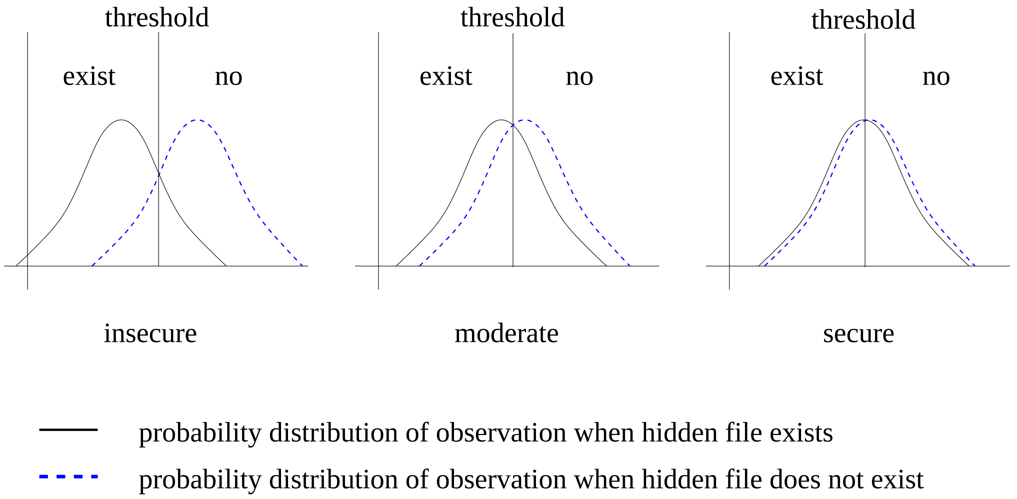


Figure 4.2: System Security VS the Probability Distributions of Observations

probability distribution of attacker's observation, attacker could be able to more accurately estimate the existence of hidden file than random guess. As an example, if the existence of hidden file will cause the value of a particular meta-data to turn from 0 to 1, attackers could base his decision on whether this value is 0 or 1. Thus, this meta-data helps attackers more accurately assess the existence of hidden files. To construct a secure steganographic file system, such evidence should be eliminated.

According to above proof, if $P_{I|F=true} \neq P_{I|F=false}$, an attacker could actually set up a threshold in the observation space to maximize the accuracy of his decision on whether there is hidden file. This is illustrated in figure 4.2. When $P_{I|F=true} > P_{I|F=false}$, he determines that hidden file exists. When $P_{I|F=true} < P_{I|F=false}$, he determines that hidden file does not exist. The more similar between the probability distributions $P_{I|F=true}$ and $P_{I|F=false}$, the less effective the threshold to determine the existence of hidden file, and the more secure the file system. Thus, to design an effective steganographic file system, it is crucial to ensure that the observation in the insecure domain is not affected significantly by the existence/non-existence of hidden files, so that $P_{I|F=true}$ and $P_{I|F=false}$ could sufficient similar. We first propose the definition of a unconditionally secure steganographic file system.

Definition 4.2.2. If the probability distribution of the observation in the insecure domain given that hidden files exist exactly matches the probability distribution of the observation given that no hidden file exist, i.e. $P_{I=i|F=true} = P_{I=i|F=false}$ for all the is, we call the steganographic file system **unconditionally secure**.

An unconditionally secure steganographic file system would be perfect but not absolutely necessary. Users would still regard the file system secure enough, if the two probability distributions are so similar that the accuracy of attacker's determination is limited to a very small range.

However, in some circumstances an attacker could obtain unlimited observations in the insecure domain. Namely, if he spends more time and effort, he could obtain more observation, and would be likely to discover more evidence to prove the existence of hidden files. For example, in a steganographic file system built on a shared network storage, the I/O operations on the storage could potentially expose the existence of hidden files. If an attacker spends more time to monitor the I/O operations, he could probably find more hints about the existence of hidden file. Thus, if a steganographic file system is not unconditionally secure, attackers could always accumulate enough evidences to accurately determine whether it contains hidden file.

Theoretically, we suppose that each time an attacker could obtain one observation I from the insecure domain, and he is able to obtain multiple Is if he spends more time and energy to observe. Let S_I denotes the set of observations obtained by an attacker. According to the Weak Law of Large Numbers in information theory [46], if $|I|$ is finite and $P_{I|F=true}$ and $P_{I|F=false}$ do not exactly match, by accumulating sufficiently large number of Is, $P_{S_I|F=true}$ and $P_{S_I|F=false}$ would be

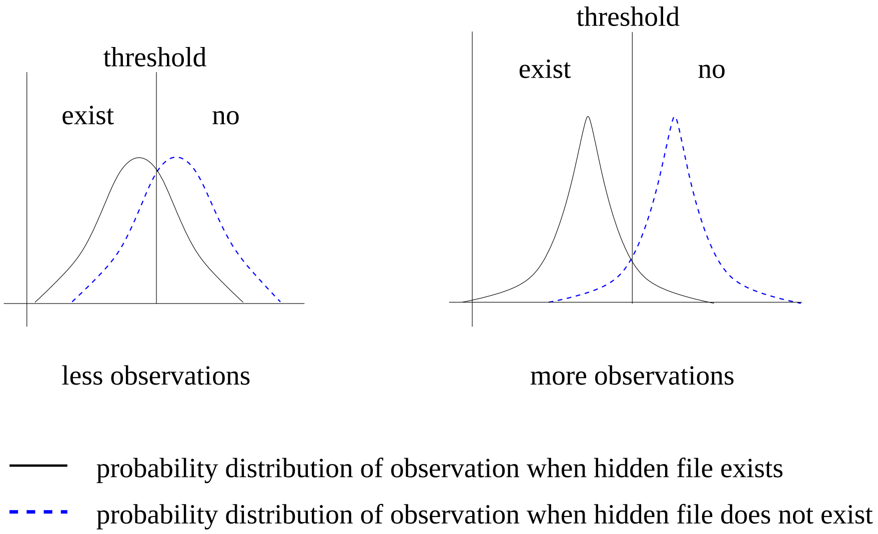


Figure 4.3: More Observations Increase the Accuracy of Attacker's Decision

very different. This is illustrated in figure 4.3. Therefore, if $P_{I|F=true} \neq P_{I|F=false}$, an attacker could always be able to accurately determine the existence of hidden files if he can collect sufficiently large number of observations.

In practice, however, the computation resource of an attacker is limited. In order to effectively detect the existence of hidden files, he may need a long time and a large amount of computational power, which may be infeasible to obtain. In this case, even though there is no perfect match between $P_{I|F=true}$ and $P_{I|F=false}$, we can also consider the system to be secure. Based on this rationale, we propose the definition of a computationally secure steganographic file system.

Definition 4.2.3. Let T denote the maximum tolerable error for attacker to determine the existence of hidden files. $P(S_I)$ denotes the computation cost to collect and analyze the set of observation S_I. Furthermore, to determine the existence of hidden files with an error rate less than T, it requires some minimum computation cost $P(S_I)$. Thus, $P(S_I)$ is proportional to the security level of the steganographic file system. If $P(S_I)$ is infeasible for attackers to acquire, the system is **computa-**

tionally secure.

In real world applications, a computationally secure steganographic file system is already adequate. So, the objective of a system designer is to transform $P_{I|F=true}$ and $P_{I|F=false}$ to be sufficiently similar such that the system could satisfy the condition to be computationally secure. We would like to thank Claude Elwood Shannon, the founder of the communication theory, as the above definitions are inspired by his definitions of security in the context of cryptographic systems [62].

4.3 A Security Analysis of StegFD

We apply the above model to StegFD. The activity space of StegFD should be divided along the dimension of time, because it is designed for systems whose storages are protected locally. Only when the system is occupied by an attacker, it is exposed in the insecure domain, and the attacker could examine the secondary storage to look for hidden files. As attackers could not monitor the file system at run time, all he can observe in the insecure domain is a snapshot of the storage space. Hence, his estimation of whether there is any hidden file would be based on this snapshot.

The snapshot of the storage space shows to the attacker the blocks abandoned during system creation, the encrypted data blocks occupied by some hidden files and some unallocated blocks. (For simplicity, we omit dummy files in our analysis.) As we assume that the encryption used by StegFD is very secure, attackers should not be able to distinguish between an encrypted data block and an abandoned block filled with randomly generated data. For all encrypted data blocks and abandoned blocks are randomly distributed across the storage space, attackers should get no information from the distribution of these blocks. Therefore, an attacker can only base his determination of the existence of hidden files on the storage space utilization, i.e. the total number of abandoned blocks and encrypted data blocks.

Suppose the storage space consists of N blocks, in which a blocks are abandoned blocks and d blocks are occupied by hidden files. Suppose F denotes the existence/non-existence of hidden file. According to our model, the similarity between probability distributions $P_{a+d|F=true}$ and $P_{a+d|F=false}$ determines the security level of StegFD. Here, $P_{a+d=k|F=false} = P_{a=k,d=0}$ and $P_{a+d=k|F=true} = \sum_{i=1}^{k} P_{a=k-i,d=i}$, since the existence/non-existence of hidden file is equivalent to the existence/non-existence of encrypted data blocks. In order for StegFD to be unconditionally secure, these probability distributions should satisfy $P_{a+d|F=true} = P_{a+d|F=false}$, namely $P_{a=k,d=0} = \sum_{i=1}^{k} P_{a=k-i,d=i}$ for each $k \leq N$. That is to say, the probability distribution of the number of abandoned blocks P_a should satisfy the following set of equations.

$$\begin{cases} P_{d=0|a=1}P_{a=1} = P_{d=1|a=0}P_{a=0} \\ P_{d=0|a=2}P_{a=2} = P_{d=2|a=0}P_{a=0} + P_{d=1|a=1}P_{a=1} \\ \ldots\ldots \\ P_{d=0|a=N}P_{a=N} = P_{d=N|a=0}P_{a=0} + \ldots + P_{d=1|a=N-1}P_{a=N-1} \\ \sum_{i=0}^{N} P_{a=i} = 1 \end{cases} \tag{4.3}$$

Actually, there is a valid P_a that satisfies the above set of equations. If the number of abandoned blocks of a StegFD follow this distribution P_a, the StegFD would be unconditionally secure. However, in this P_a, $P_{a=i}$ would increase sharply with i, so that the expected number of abandoned blocks would be very high and the effective space utilization would be limited to a very low level. In practice, to balance security with space utilization, the system administrator need not introduce so many abandoned blocks to make a StegFD unconditionally secure. When the system is not unconditionally secure, although an attacker could make more accurate estimation about whether hidden files exist than random guess, it is unlikely that this small improvement on accuracy could enable him to perform effective attacks. As the attacker cannot obtain any other observation than the snapshot of the raw storage, he cannot further improve the accuracy of his judgement by collecting more evidences. Therefore, even when a StegFD is not unconditionally secure, it could still remains a certain security. As mentioned in Chapter 3, the expected fraction of the abandoned blocks in StegFD is normally below 50%.

4.4 Summary

This chapter proposed a model for steganographic file system. In this model, the activity space of a file system is divided into secure and insecure domains according to particular applications, and the information exposed in the insecure domain is used by attacker to discover hidden files. Then the security of a system construction could be assessed by whether the attacker could obtain sufficient information to accurately estimate the existence of hidden files. This model will be used frequently in the following chapters, when we design steganographic file system on various platforms that are faced with different levels of risks.

Chapter 5

Hiding updates in Steganographic File System

The system model introduced in chapter 4 provides theoretical fundaments for developing effective steganographic file system in various applications are faced with different types of risks. In this chapter, we attempt to extend the application of steganographic file system from local exclusive platforms to distributed shared platforms, which are challenged by additional threats such as *update analysis*. We first introduce and study update analysis attacks, and then propose a construction of steganographic file system that is unconditionally secure against update analysis attacks. Finally, we presents some experiment results that confirm the effectiveness and practicality of the construction.

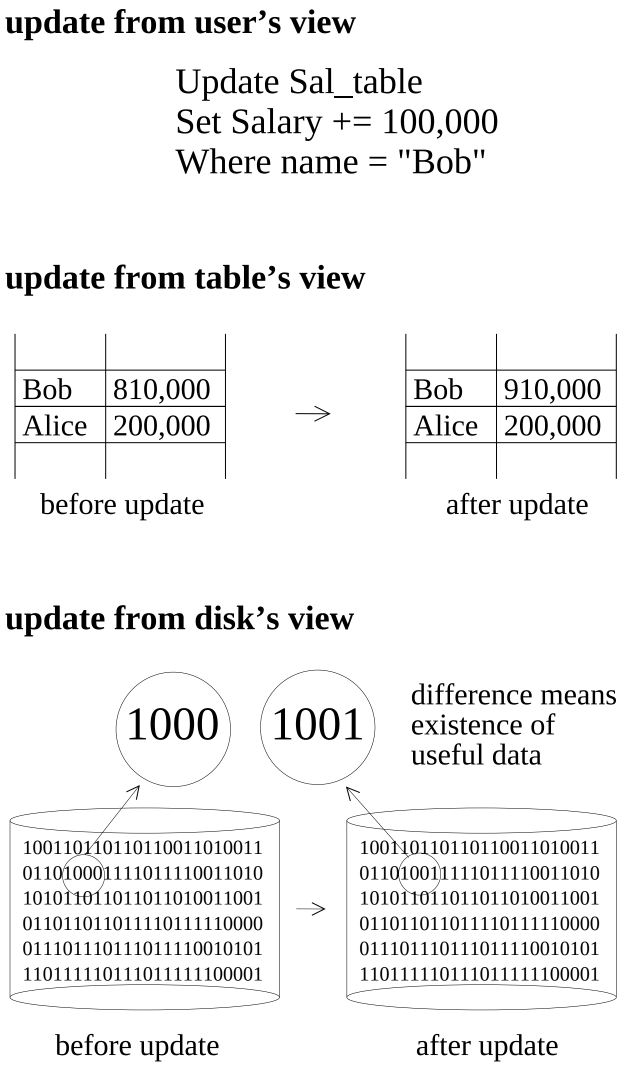
The chapter is organized as follows. Section 1 introduces some emerging applications and systems that are faced with update analysis attacks. Section 2 gives an overview of update analysis attacks, and defines the specific model of steganographic file system to counter update analysis. The system construction to guard against update analysis is given in Section 3. Following that, Section 4 describes the system implementation and performance evaluation results. Finally, Section 5 concludes the chapter.

5.1 Introduction

Ubiquitous computing entails the permeation of computing in every facet of our lives, be it work, personal or leisure, to a point where users take it for granted and stop to notice it. The data that underlie the ubiquitous services have to be persistent and available anywhere-anytime. This means that the data must migrate from devices local to individual computers, to shared network storage. A development that would facilitate this migration is the emergence of data grids (e.g. see [1, 21]), which enable arrays of storage nodes, possibly separated over long distances, to function together as a single integrated block-access volume. Another supporting development is the recent interest in building reliable logical storage volumes on unreliable nodes in a peer-to-peer platform (e.g. [44]). We are then motivated to apply steganographic file system to such platforms to provide a strong protection for private data – without being authored, one cannot determine whether the data exists.

While shared network storage provides the availability needed for ubiquitous computing, it introduces new challenges in data security. For a steganographic file system built on shared storage, there are new avenues from which an attacker could attempt to break it. Specifically, if an attacker can compare consecutive snapshots of the storage space, he can detect changes on blocks that do not belong to any plain files, and conclude that one or more hidden files exist. We call this attack *update analysis*. Figure 5.1 illustrates the update analysis problem. A small update on Sal_table leads to a difference between the snapshot taken before the update and the next snapshot after the update. This difference suggests that the DBMS has updated some hidden data, and can be used by an attacker as evidence to disclose the table being updated.

The StegFD introduced in chapter 3 as well as other previous steganographic



```
Update Sal_table
Set Salary += 100,000
Where name = "Bob"
```

Bob	810,000
Alice	200,000

before update

Bob	910,000
Alice	200,000

after update

Figure 5.1: Hidden Data is Exposed by Update

file systems are primarily designed to ensure that an attacker cannot easily deduce the existence of hidden files by examining a single snapshot of local storage devices. They do not address the additional risk faced by shared storage. In this chapter, we propose another system construction to protect against update analysis attacks. The mechanisms are constructed to balance between three different objectives: (a) security: an attacker cannot deduce whether the blocks involved in any observable updates patterns contain genuine data; (b) integrity: the data relocations and

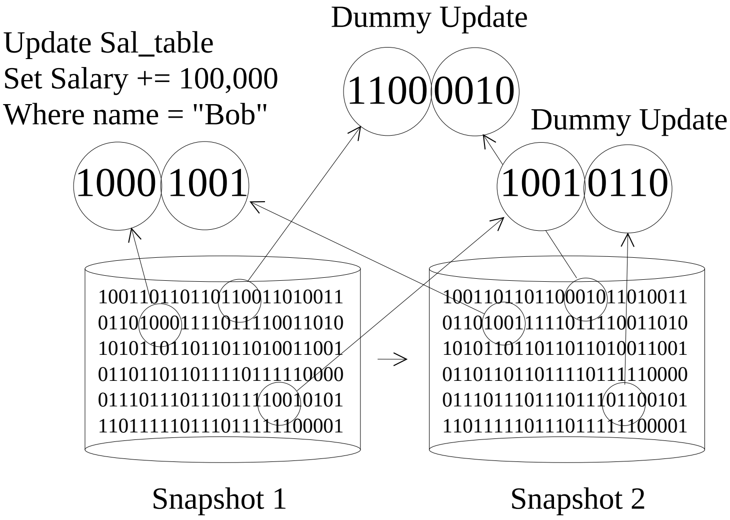


Figure 5.2: Effect of Dummy Accesses

dummy updates should not compromise the integrity of the hidden files, resulting in irrecoverable data loss; and (c) performance: any performance degradation from the overheads introduced should be minimized.

5.2 System Model against Update Analysis

In this section, we outline the specific model for the steganographic file system that we designed to counter those attacks.

5.2.1 Dummy Update

To prevent updates (as illustrated in figure 5.1) from exposing the existence of hidden data, a counter measure is to issue a stream of purposeless updates on the storage. If these *dummy updates* could be made to appear indistinguishable from the genuine data accesses, attacker would not be able to deduce the existence of hidden data from any observed updates. As figure 5.2 shows, since the system has been conducting dummy updates on the storage periodically, the attacker cannot

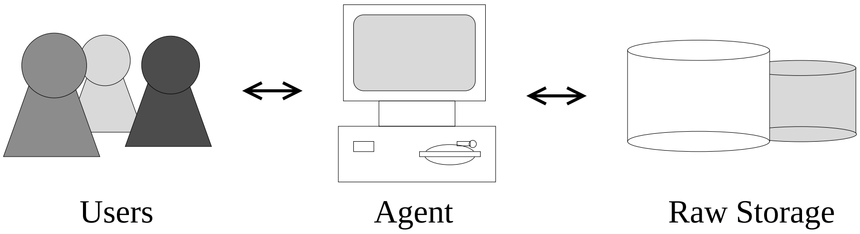


Figure 5.3: Model of Steganographic File System to counter update analysis

tell whether a changed block is due to a real or dummy update. Hiding real data updates among dummy updates is the basic idea for constructing a steganographic file system to defend against update analysis attacks.

5.2.2 System Model

In this subsection, we describe a model of the steganographic file system that is able to hide data updates. We also give a security notion to measure the effectiveness of hiding data accesses based on the model of chapter 4.

System. Figure 5.3 shows the model. The users on the left hand of figure 5.3 have their data files stored in the raw storage. Between the users and the storage is an agent that is fully trusted by the users and is authorized to access the storage directly. Whenever users need to access their data files in the raw storage, they have to route the requests through the agent. Upon receiving the requests, the agent translates them to corresponding I/O operations, and afterward returns the results to the users. When there is no active workload, the agent would issue dummy updates on the raw storage. Therefore, any attacker who might be monitoring the raw storage would not be able to isolate users' update operations from dummy updates, and thus cannot deduce the existence of hidden files.

Commonly, as the users and the agent can communicate through some trusted channels deliberately, they are always located in the secure domain. But the raw storage is the shared resource in the network, which is always in the insecure domain. Common practical scenarios for such a model include shared storage area networks (SAN), data grids [1, 21] and peer-to-peer storage platforms [44].

Attacker. Attackers of such a steganographic file system are able to scan the whole raw storage repeatedly, so they can identify any updates conducted on the raw storage. Or they are able to examine the activity log to discover the updates conducted in the past. We assume that attackers have a complete understanding of the scheme running in the system. However, they do not know any secret access keys held by users or the agent. Neither can they observe the real-time operations within the agent and the interactions between the agent and users in the secure domain. We assume that the users can communicate with the agent through a secure channel and the agent is a computer that is properly shielded from external probes.

Memory. The raw storage is the only permanent mass storage in the system. However, we allow the clients and the agent to have some local cache. A user should keep track of the access key(s) to his hidden files, through which the agent can authenticate the user's identity and locate the corresponding hidden files. The access keys may be committed to the user's memory, or stored within a tamper-proof device like a smartcard. The agent needs some working memory to carry out its processing. Its working memory is volatile and thus leaves behind no information to attackers. We distinguish between an agent that has a non-volatile memory for storing some secret information about the file system, and one that does not:

- **Non-volatile Agent** This category of agent runs in a very safe environment that is immune to any attacks. It possesses a non-volatile memory for keeping

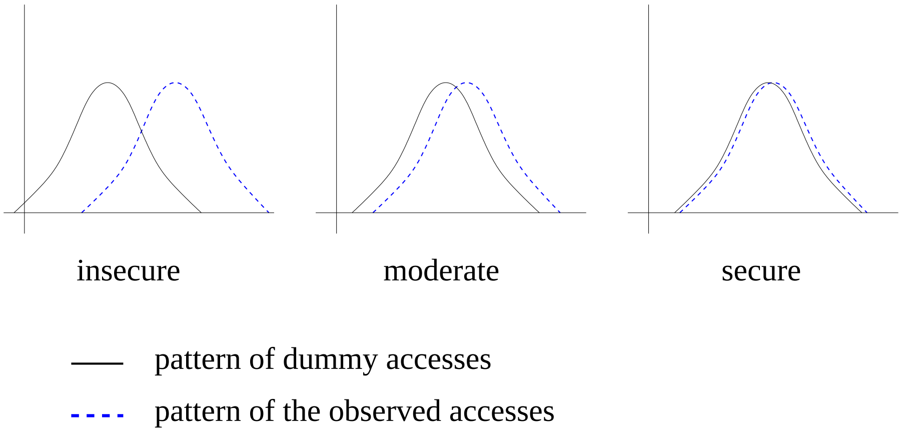


Figure 5.4: Effectiveness of Hiding Updates

some secrets on the file system. The shortcoming is that the system administrator could be at risk of being coerced by attacker to disclose the hidden data.

- **Volatile Agent** This category of agent does not retain user information in persistent memory, and is less likely to compromise the system even if the protection around the agent is breached. The trade-off is that there is a higher maintenance cost.

While the user machines and the agent are allowed to have some local cache, they are not of the same order of magnitude as the raw storage. Thus, user data still have to be stored on the raw storage.

Definition of Security. To conceal the existence of files, the agent can encrypt the files, introduce random data, and scatter them across the storage space just like the StegFD in chapter 3. At the same time, the agent should hide user updates by mixing in dummy updates. According to chapter 4, the pattern of real data updates should appear the same as the pattern of dummy updates. Otherwise, an attacker may be able to isolate the real data updates and prove the existence of hidden files. This is illustrated in figure 5.4, where the update pattern is expressed as the probability distribution of the update sequence. Here we give the definition of security for hiding data updates in a steganographic file system, as an extension of definition 4.2.2 and definition 4.2.3 in chapter 4.

Definition 5.2.1. Let X denote the sequence of updates the agent performs on the raw storage. Its probability distribution is P_X. Y denotes the set of update requests users submit to the agent, and when there is no request, $Y = \emptyset$. $P_{X|Y}$ is the conditional probability distribution of X given a particular Y. (Thus, $P_{X|\emptyset}$ is the probability distribution of dummy updates.) A system is *unconditionally secure* if and only if, whatever Y is, $P_{X|Y} = P_{X|\emptyset}$. A system is *computationally secure* iff $P_{X|Y}$ and $P_{X|\emptyset}$ are so similar that it is computationally infeasible for an attacker to distinguish between them from a sufficiently large sequence of updates.

5.3 A Construction to Counter Update Analysis

This section presents the mechanism to equip steganographic file system to counter update analysis, where attackers might take multiple snapshots of the raw storage and detect updates on hidden files. We make a strong assumption that attackers can observe all the updates in the raw storage, although not all the attackers are so powerful in reality. The task of the agent is to hide the data updates from attackers by introducing dummy updates.

For simplicity, the agent's dummy updates are generated from a random process. However, as users' update operations could exhibit some regular patterns, e.g. table scans, an attacker might be able to isolate the data updates through some statistical methods. The proposed mechanism counters this threat, by changing the location of data blocks systematically to remove any regular pattern in the update operations.

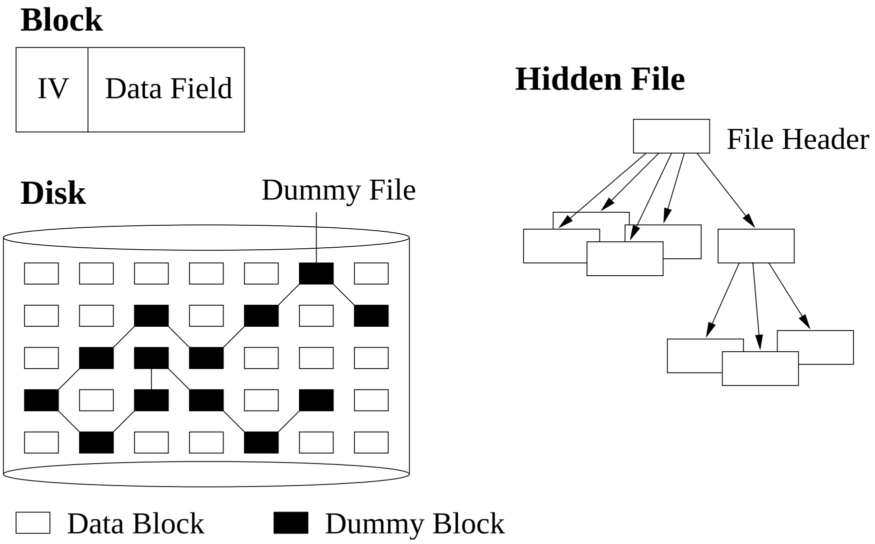


Figure 5.5: File System Construction

We begin with a construction that works with a non-volatile agent, and subsequently extend the mechanism to work with a volatile agent.

5.3.1 Construction 1: Non-Volatile Agent

A non-volatile agent is able to retain some critical user information, so that it has a complete view of the file system at any time and can freely reorganize it. This simplifies the task of hiding updates and system maintenance.

Data blocks

Figure 5.5 shows the basic construction of this scheme. As in conventional file systems, it partitions the raw storage into standard-size blocks, and classifies them into *data blocks* that contain useful data and *dummy blocks* that contain only random bytes. Both groups of blocks are scattered randomly across the storage volume.

As figure 5.5 shows, each block contains an *initial vector (IV)* and a *data field*.

The data field contains real data in the case of a data block, and random bytes if it is a dummy block. For each block in the raw storage, whether a data block or a dummy block, its data field is encrypted by the agent using a CBC (Cipher Block Chaining) block cipher with the IV as seed. Whenever the agent re-encrypts a block, it resets the IV so that the content of the whole encrypted block changes. This enables the agent to carry out dummy updates on any block, by simply changing its IV. An attacker without the encryption key cannot tell whether the data field is actually modified.

Hidden files

A hidden file is a set of data blocks that are organized in a tree structure, with the file header as the root note. This structure of hidden file is similar to that of StegFD in chapter 3. The location of the header of a hidden file is derivable from its access key FAK and path name. Once these are provided by the owner, the agent can recover the file content from the raw storage. An attacker without the FAK would not be able to deduce the existence of the hidden file even if he scours through the raw storage.

All the dummy blocks in the raw storage belong to a single *dummy file*, a hidden file whose FAK is held by the agent. Hence the agent keeps two keys in its non-volatile memory. One is the FAK of the dummy file, the other is the secret key for encrypting all the storage blocks.

Dummy updates

Whenever there is no user activity, the agent would issue dummy updates on randomly selected blocks in the storage volume. In each dummy update, the agent reads in the selected block, decrypts it, assigns a new random number to its IV, re-encrypts it, and then writes it back. The dummy updates are completely random, i.e., every data block has the same probability of being selected. The dummy updates do not compromise data integrity since only the IVs are changed.

As the data blocks are encrypted, without the agent's encryption key, an attacker cannot differentiate a dummy update that only changes the IV from an update that modifies the data content. As the dummy updates are inserted in between data updates, their frequencies are similar so the attacker cannot isolate the data updates through any variance in update frequency.

Data updates

The introduction of dummy updates alone is not enough to hide the existence of data updates. The pattern of data updates must also be made similar to that of a random process. We achieve that by relocating a data block each time it is updated, so that the access pattern for a logical data block cannot be established by attackers.

When there is a request to update a data block, the agent first randomly selects a block within the storage volume. If the selected block is exactly the same block that is being updated, the agent simply performs the required update on it. If the selected block is a dummy block, the agent swaps it with the data block and updates its content in the process. Otherwise, if another data block is selected, the agent does a dummy update on it, and starts over again to look for another block. The update algorithm, given in figure 5.6, combines the procedures for dummy update and data update.

Proof of Security

Now, we prove that this scheme is unconditionally secure against update analysis.

```
func update ()
if there is a request to update block B1, then
  Re: randomly pick up a block B2 from the storage space;
      if B2 = B1, then
          read in B1, decrypt it,
          update B1's IV and data field,
          encrypt B1, write it back;
      else if B2 is a dummy block, then
          read in B1,
          substitute B2 for B1,
          update B2's IV and data field,
          encrypt B2, write it back;
      else
          read in B2, decrypt it,
          update B2's IV,
          encrypt B2, write it back;
          goto Re;
else            // dummy update
      randomly pick up a block B3 from the storage space;
      read in B3, decrypt it,
      update B3's IV,
      encrypt B3, write it back;
func end
```

Figure 5.6: Update Algorithm

Proof. When there is no data update, all the updates on the raw storage are dummy updates, which follow random distribution, i.e. $P_{X|\emptyset} = P_{ran}$. When there is data update, as each updated block is still randomly selected from the whole storage space (based on the above algorithm), all the updates on the raw storage also follow random distribution, i.e. $P_{X|Y} = P_{ran}$. Therefore, whether there is any data update or not, the updates on the raw storage follow the same probability distribution as that of dummy updates, i.e. $P_{X|\emptyset} = P_{X|Y}$. According to the definition 5.2.1, the scheme is unconditionally secure. $\square$

Being unconditionally secure means that the system is very vigorous against update analysis – without knowing the agent's encryption key, attackers can get no information of the hidden data no matter how much effort they spend on analyzing the updates on the raw storage.

Processing Overhead

An update in a conventional file system would incur two I/O operations – read in the block, update it and write it back. With our scheme, the agent needs to repeat a block selection procedure until it successfully completes the update. Each iteration in this procedure incurs two I/Os – to read in a block and write out the block. Therefore, the processing overhead is decided by the number of iterations. Suppose the raw storage has N blocks, out of which D are dummy blocks. The probability that a randomly selected block is a dummy block is $p = \frac{D}{N}$, and the probability that i iterations are needed is $(1-p)^{i-1}p$. Thus the expected overhead, defined as the total number of I/Os in our scheme divided by the number of I/Os in a conventional file system, depends on the fraction of dummy blocks in the storage volume:

$$E = p + 2 \times (1-p)p + 3 \times (1-p)^2 p + ... = \frac{N}{D}$$

If at least half of the storage space is occupied by dummy blocks, i.e., the space utilization is kept below 50%, the expected overhead is 2 at the very most. As storage space is cheap today, it makes sense to sacrifice some space to achieve better processing throughput.

Another overhead of our scheme is the block relocation upon each update. As each data block is traced through its file header, we need to update the header whenever a block is relocated. However, since the file header is always placed in the cache and is written out only when the file is saved, this overhead will not add significantly to the response time. For database objects, such as B-tree, the relocation of a block would require a propagation of updates on a number of other blocks and thus incur a higher overhead. While the performance optimization on steganographic DBMS is beyond the scope of this thesis, it is scheduled in our future works (see Section 7.2.3).

5.3.2 Construction 2: Volatile Agent

While the above construction for non-volatile agent protects against update analysis on the raw storage, the encryption key for all the data and the FAK for the dummy file are kept centrally in the persistent memory of the agent. This could subject the administrator of the agent to coercion from attackers. In this subsection, we extend the construction to work with a volatile agent that does not use a persistent memory to store any secret about the file system, so that attackers cannot elicit any useful information from the administrator. In this second scheme, the encryption key of the hidden files are retained by the owners, and each user possesses his own dummy file(s). The encryption key and the FAK of the dummy file(s) are disclosed to the agent only when the user logs on.

Distributing secrets to users

Instead of using the agent's key to encrypt all the blocks, this construction assigns each hidden file encrypting keys. Actually, the FAK of each hidden file comprises 3 components – the location of the file header, a header key for encrypting the header information, and a content key for encrypting the file content. Moreover, dummy blocks in the raw storage are organized into dummy files of approximately the size of data files, and distributed to the users. Within the FAK of a dummy file, only the location of the header and the header key are used; the content key is not utilized because the file contains only random bytes.

With this scheme, a user who is being compelled to disclose his hidden files can just expose some dummy files and remain silent on his hidden data. He can even reveal the header key for a hidden file but give a wrong content key, and claim that the file is a dummy.

Operations of the volatile agent

The volatile agent performs updates on the raw storage in the same way as the non-volatile agent, except that here the agent can only update files that users have disclosed to it.

When the agent starts up, it has zero knowledge of the hidden and dummy files in the raw storage. As each user logs on to the system, he shares the FAKs to his hidden files and dummy files with the agent. As more users log in, the agent would discover more hidden files and dummy blocks to carry out dummy updates on. Thus, while an attacker may find part of the raw storage being accessed at any one time, this does not disclose any meaningful information since the updated blocks do not necessarily contain useful data.

Key management

Most security systems provide key management mechanisms to carry out the operations like key generation, verification and backup. But our steganographic file system lets each individual user to manage their own keys. Whenever the FAK of a hidden file is generated, the user keeps it in his local memory and uses his local key management facility to maintain his FAKs. Sometimes, he can also refer to some third-party key management service outside the steganographic file system.

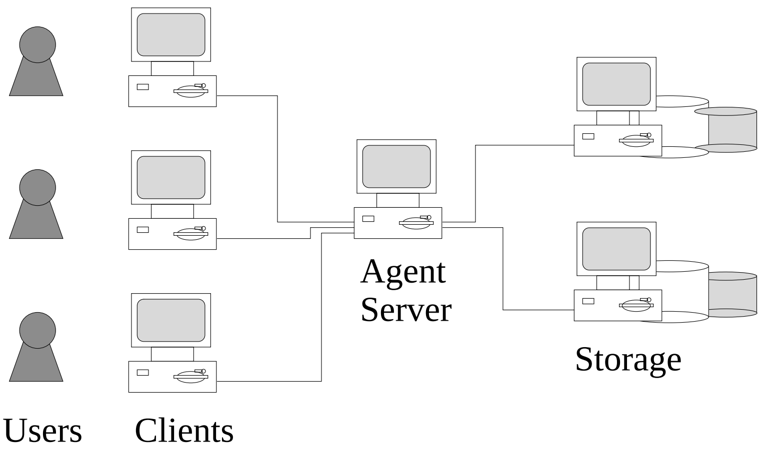


Figure 5.7: System Architecture

5.4 Implementation and Evaluation

We have implemented a steganographic file system based on the volatile agent scheme introduced in Section 5.3.2 and conducted experiments to evaluate their performance. This section begins by describing the implementation, then presents results from some interesting experiments.

5.4.1 System Implementation

We implemented the proposed steganographic file system in Linux. Figure 5.7 shows the architecture of the implementation. It consists of three components: the client, the agent and the storage. The client component provides an interface through which users can access their hidden files in a similar way as in a conventional file system. The agent component acts as a server that processes all the client requests and manages the storage. The storage component provides storage resources and may be located either on the same machine as the agent, on a different machine, or on a networked storage system like OceanStore [44]. We use AES [4] for the block cipher, and the pseudo-random number generator is constructed

Parameter	**Value**
Model of the CPU	Intel Pentium 4
Clock speed of the CPU	1.6 GHz
Type of the hard disk	Ultra ATA/100
Capacity of the hard disk	20 GB

Table 5.1: Physical Resource Parameters

from SHA256 [6].

5.4.2 Experimental Evaluation

We first conduct experiments to evaluate the I/O performance of the schemes that can counter update analysis (see Section 5.3). The platform we used for the experiments is an Intel PC, whose key parameters are listed in Table 6.3. And Table 6.7 summarizes the workload parameters. For comparison, we use as baselines the native Linux file system and the StegFD introduced in chapter 3. The notations for the various file systems are shown in Table 5.3.

Parameter	**Default**
Size of each disk block	4 KBytes
Size of each file	(4, 8] MBytes
Capacity of the disk volume	1 GBytes
Space Utilization	(0, 50%]

Table 5.2: Workload Parameters

StegHide indicates the volatile agent scheme which we have implemented as a real file system. We installed the file system on the Intel PC, with the agent and the storage components running together on the PC. *StegHide** indicates the non-volatile agent scheme we have simulated. The simulation is conducted on a 1GB disk volume. We use a bitmap to mark data blocks against dummy blocks, and conduct updates on randomly selected data blocks, using the algorithm in

Parameter	**Meaning**
$SteqHide$	Construction 2: volatile agent
$SteqHide*$	Construction 1: non-volatile agent
$SteqFD$	The file system in chapter 3
$CleanDisk$	A fresh Linux file system
$FragDisk$	A well-used Linux file system with fragmentation

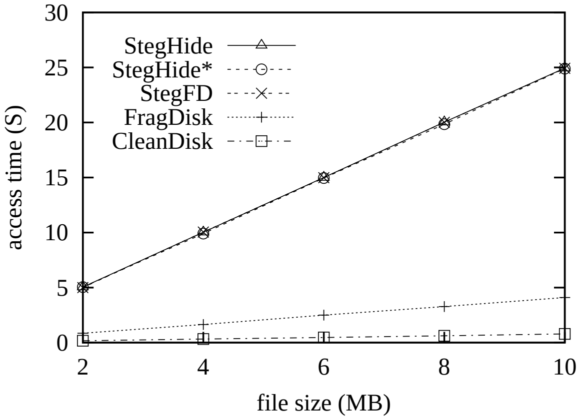
Table 5.3: Algorithm Indicators

Figure 5.6. *StegFD* is our former steganographic file system introduced in chapter 3. *CleanDisk* and *FragDisk* are native file systems in Linux - *CleanDisk* is a fresh file system, whose files reside on contiguous data blocks. *FragDisk* is a well used file system whose storage are fragmented, and we simulate it by breaking each file into fragments of 8 blocks.

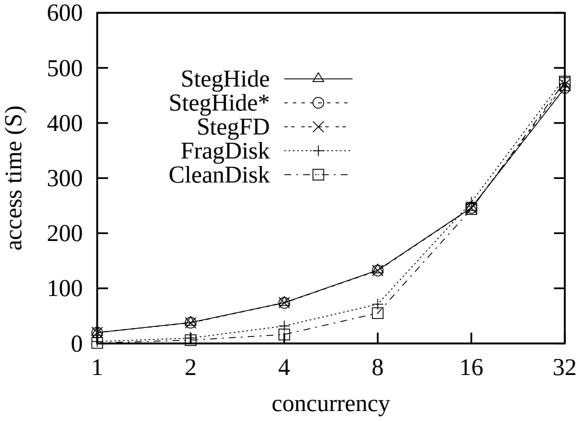
Performance on data retrieval

The first group of experiments aims to study the performance of retrieving files from the steganographic file system. We vary the file size and the number of concurrent users, and study how they affect the access time of retrieving a file from various file systems. Figure 5.8 (a) shows the access times of retrieving files of different sizes in a single user environment. Figure 5.8 (b) shows the sensitivity of the access time to the number of concurrent users.

StegHide, StegHide* and StegFD display similar performance in data retrieval, since their data blocks are distributed across the storage in the same manner. In a single user environment, FragDisk and CleanDisk outperform the three steganographic file systems, as they can perform sequential I/O on their contiguously located data blocks. But their advantage diminishes as the degree of concurrency



(a) Sensitivity to File Size



(b) Sensitivity to Concurrency

Figure 5.8: Performance on Data Retrieval

increases. As shown in figure 5.8 (b), when the number of users increases to 16 onward, random I/Os dominate the whole process, the access times of the five systems become very close.

Performance on updates

Having demonstrated our file system's performance on data retrieval, we proceed to profile its update performance.

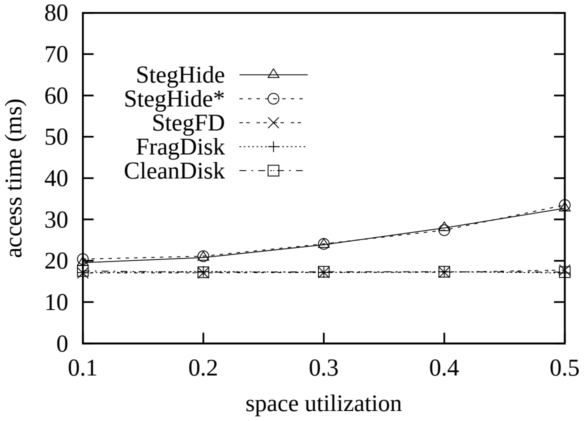
As our system intends to counter update analysis, it introduces extra overhead to update operations. This overhead is affected by the space utilization, which is explained in Section 4.1.5. Thus we first study the sensitivity of the update performance to space utilization. We vary the space utilization from 10% to 50%, and plot the access time of updating a randomly selected data block of a file. The results are shown in figure 5.9 (a).

The update overheads of StegHide and StegHide* increase with increasing space utilization. This matches our analysis in Section 4.1.5, where we state that $E(overhead) = \frac{N}{D}$. As the storage space is cheap today, it is feasible to use extra storage space to exchange for a better update performance. Actually, in our implementation, we limit the space utilization to below 50%.

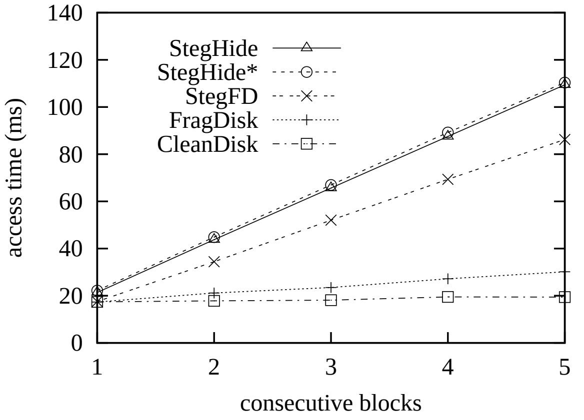
Sometimes an update is performed on a large range of data which may occupy more than one consecutive data blocks. In the second set of experiments, we study the sensitivity of update performance to the number of consecutive blocks being updated. We fix the space utilization of StegHide and StegHide* to 25%, and vary the update range from 1 to 5 data blocks. The results are shown in figure 5.9 (b). The access times of FragDisk and CleanDisk do not vary significantly with the increasing update range because of the benefits of sequential I/O, while those of the three steganographic file systems increase linearly with the number of updated blocks.

The third set of experiments aims to study the performance of updates in a multi-user environment. We fix the update range to 5 data blocks, and plot the access times of various file systems for different degree of concurreny. Figure 5.9 (c) shows the results. Like the experimental results on data retrieval, FragDisk and CleanDisk lose their advantage in utilizing sequential I/O when the degree of concurrency is high.

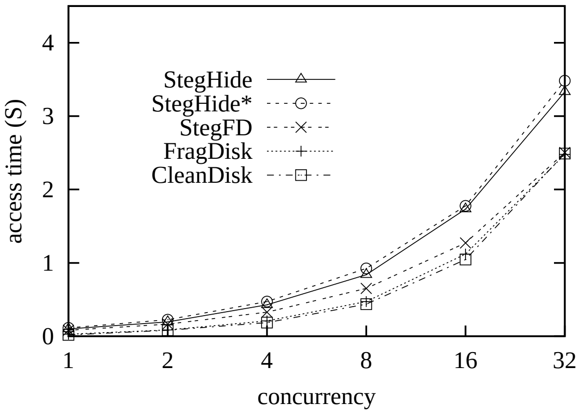
In summary, as a multi-user file system, StegHide and StegHide* can effectively



(a) Sensitivity to Space Utilization



(b) Sensitivity to Update Range



(c) Sensitivity to Concurrency

Figure 5.9: Performance on Update

counter update analysis without incurring heavy overhead over general file systems.

5.5 Summary

In this chapter, we propose a steganographic file system with the ability to counter attacks initiated through analyzing data updates from user applications. It works by introducing dummy updates into the storage to conceal the existence of real data updates. To prevent attackers from distinguishing between real data updates and dummy updates, the system relocates data blocks systematically to completely remove the pattern in data updates. Two constructions are built for this file system, one for a non-volatile agent which is trusted by users to keep their access keys, and the other for a volatile agent which is not so trustworthy. We implemented the constructions in Linux, and conducted experiments to show their reasonable performance and potential for real world applications.

Compared with StegFD, which assumes that attackers could assess the existence of hidden files from only a snapshot of the storage, the system proposed in this chapter relaxes the assumption to that attackers could repeatedly observe the storage to identifies data updates on hidden files. In some scenarios, however, attackers can not only observe data updates but also monitor I/O traffics on the shared storage. An example is a storage service provider, which hosts the storage but is not trusted by users to keep data privacy. To construct a steganographic file system on such storage, designer needs to adopt another mechanism to prevent attackers from detecting hidden files by analyzing I/O traffics. Chapter 6 aims to solve this problem .

Chapter 6

Hiding Data Traffic in Steganographic File System

Last chapter made an initial attempt to extend the application of steganographic file system to shared storage on open networks. The proposed system construction successfully mitigates the risk from update analysis by hiding real data updates into dummy updates. However, in some shared network storages, it is possible for attackers to obtain the full control of the storage device at run-time. Thus, the data traffics between the host file system and the storage becomes a new avenue for attackers to detect the existence of hidden file. In this chapter, we design new steganographic file systems to counter such attacks that attempt to disclose hidden file through analyzing artifacts in data traffics.

6.1 Introduction

With recent technology trends like peer-to-peer storage, data grid [21, 1] and pervasive computing, data are increasingly being migrated from local storage devices to shared storage on open networks. This raises the issue of protecting confidential data in untrusted storage [45], where adversaries may be observing the content and activities. For example, in a storage area network (SAN), storage devices are not attached to any particular server, but distributed over a high-speed network or even the internet. Beyond the protection of the server, the remote storage devices could be controlled and monitored by attackers without raising suspicion from system administrators. Storage service provider emerges as a new internet service, which could provide users massive and stable storage space that is available at anywhere and anytime. However, storage service providers would not necessarily be trusted to protect users' confidential data. On the contrary, they could abuse the data for their own benefits.

Building steganographic file system on platforms like SAN and storage service providers is therefore confronted with additional threats. In particular, attackers, who are monitoring the storage device, could statistically analyze the data updates and I/O traffics on the storage for the existence of hidden data. The steganographic file system introduced in last chapter is able to guard against update analysis attacks, but is vulnerable to traffic analysis attacks, which take account of not only data updates but all the I/O activities. I/O activities are much more difficult to hide, because the block relocation which has been used to remove the patterns in data updates becomes traceable to attackers and is ineffective in removing I/O access patterns.

In this chapter, we propose two constructions of steganographic file system to counter traffic analysis attacks. Both work by hiding real I/O traffic into random dummy I/O traffics. Oblivious Storage, inspired by the Oblivious Ram in [30], is an unconditionally secure file system that could completely conceal users' access patterns in I/O traffics. But it incurs excessive I/O overhead that could be intolerable for some real-world applications. DataCavern, in contrast, is a computationally secure file system that aims to minimize the accuracy of traffic analysis. Instead of attempting to conceal the data traffic completely, DataCavern aims to minimize the accuracy of traffic analysis. It (a) intermixes data and dummy traffics to reduce their correlations; (b) relocates disk pages periodically to alter the user access patterns, and (c) buffers frequently accessed pages to remove any non-uniform distribution in the data accesses. We have conducted extensive studies on the implementation/simulation of the proposed file systems to evaluate their effectiveness and performance. The results confirm that both schemes are effective in counter traffic analysis, and that DataCavern can achieve more practical performance than Oblivious Storage.

The remainder of this chapter is organized as follows. Section 2 defines the specific system model for countering traffic analysis and gives an overview of traffic analysis attacks. Section 3 and Section 4 introduce the constructions of Oblivious Storage and DataCavern respectively. Section 5 evaluates the security and performance of the two constructions. Finally, Section 5 summarizes this chapter.

6.2 Problem Definition

In this section, we describe the specific model of a steganographic file system on an untrusted storage, as well as its major challenge – defending against traffic analysis.

6.2.1 System Model

As discussed above, the specific threat that our system is designed to defend against, over and above previous steganographic file systems, is traffic analysis by an external observer. By analyzing data traffic on the storage, the observer can potentially compromise one or more of the following: (a) data privacy, by reconstructing the logical content; (b) data integrity, by tampering with the hidden data; and (c) user privacy, by deducing the application task.

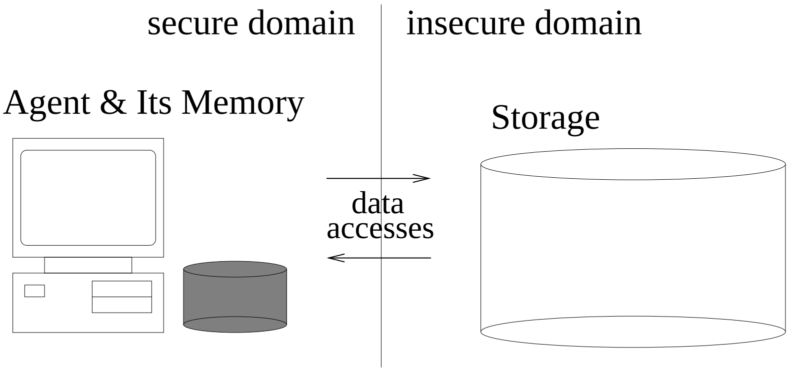


Figure 6.1: System Model

Corresponding to this threat model, the steganographic file system consists of two components as shown in figure 6.1 – an agent located within the secure domain that is typically protected by firewall(s), and a raw storage situated in the insecure domain where the external observers are. The agent is fully authorized by users to manage their data files. It utilizes a limited-sized memory to hold some critical information and to process data. Data files are hidden in the raw storage using the strategy introduced in chapter 3 and chapter 5; only the agent, with the user access keys, knows where to recover the hidden files from. Common practical scenarios for such a model include shared storage area networks (SAN), data grids [1, 21], peer-to-peer storage platforms [44], and storage services hosted by external data centers.

The key challenge in this system model is that data traffic on the raw storage could yield evidence of hidden files. As a counter-measure, the agent can mix dummy requests into the data accesses, and keep the storage active with dummy I/Os when there is no user activity, so that the visible traffic between the agent and the storage does not necessarily indicate the existence of hidden data. As data/dummy traffic result from the agent's data/dummy accesses on the storage, in the rest of this chapter, we shall refer to both simply as data/dummy accesses.

6.2.2 Traffic Analysis

In the above system model, as the whole storage and its I/O channel are exposed to insecure domain, the accesses conducted by the agent on the storage can be analyzed statistically to determine whether they include any genuine data accesses, which would then point to the existence of hidden data. This is always possible if the data accesses and dummy accesses exhibit different patterns. To illustrate, suppose that the dummy accesses in our system follow an absolutely random process, i.e., in generating a dummy access, the agent randomly picks a data block from the storage space and performs a dummy read or write operation. In contrast, users' data accesses are almost always clustered, e.g. on files, indexes or tables, and exhibit patterns like sequential scan, binary search, etc. Exploiting these differences, an attacker can employ statistical tests to accurately assess whether the observed activity includes any data traffic.

In a statistical test, a deterministic algorithm takes as input an access sequence observed at the shared storage, and produces as output a binary decision on whether the sequence contains any data accesses. A typical test would make a hypothesis that there is no data access, and uses a test statistic k to assess whether the hypothesis is correct. As shown in figure 6.2, knowing the probability distribution of k under the hypothesis, a threshold can be set to make the decision whether to accept or reject the hypothesis. The accuracy of the statistical test is determined by the probability of type I error α and that of type II error β, and an attacker would want to cap both so that $\alpha < p$ and $\beta < q$ for some pre-set p and q.

Based on the Neyman-Pearson theorem [25], the attacker could first limit α to an acceptable level, then select the statistical test that yields the minimum β. However, without knowing the actual probability distribution of k, the attacker is not able to compute the β accurately. Thus, instead of the test with minimum

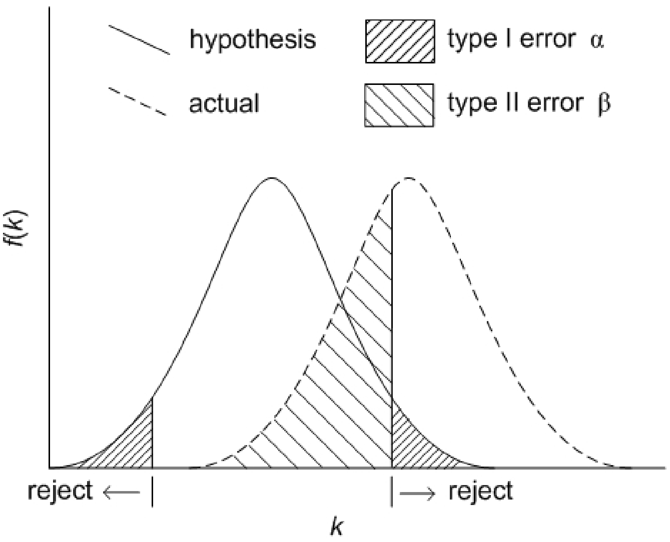


Figure 6.2: Testing for Data Accesses

β, the attacker would have to depend on some regular tests that seem effective in differentiating dummy accesses from data accesses. Many existing statistical tests for random number generators [36] could be used here, including *frequency test*, *gap test*, *run test*, *auto-correlation test*, *serial test*, or some *universal tests* [43]. In this chapter, we use the gap test to demonstrate the effectiveness of the proposed file system – DataCavern, though it works for the other tests too.

6.2.3 Overview of Solution Approach

A counter-measure against traffic analysis is to minimize the accuracy of the statistical tests described above. This can be achieved by transforming the data access pattern to be so close to that of a random process that there would be a high type I or II error associated with any test statistic.

As we have stated in chapter 4, according to the Weak Law of Large Numbers in information theory, if the data access pattern cannot be transformed to a random process perfectly, there is always an accurate statistical test for the existence of data access, given a sufficiently long access sequence. Thus theoretically a successful defense would necessitate a perfect match between the data and dummy access patterns. In practice, however, if the resources needed to crack a system are beyond what attackers can be expected to muster (e.g. too long an observation period, or too much computation power), then the system is computationally secure and offers adequate protection.

We extend definition 4.2.2 and 4.2.3 in chapter 4, and propose the following definitions to characterize the security level of a steganographic file system to counter traffic analysis:

Prerequisite: The agent accesses the storage continuously. Whenever there is no user activity, the agent issues dummy requests. When a user operates on a hidden file, the agent transforms the required data accesses into *stego accesses*, that fulfill the intention of the data accesses but exhibit similar pattern as the dummy traffic.

Definition 6.2.1. If the probability distribution of the stego accesses and that of the dummy traffic match exactly, the file system is *unconditionally secure.*

Definition 6.2.2. Suppose the set of statistical tests that could be employed by attackers to break the system is A. Let T_α and T_β denote the maximum tolerable type I error α and type II error β of any given test in A. Furthermore, to distinguish between stego accesses and dummy accesses with $\alpha < T_\alpha$ and $\beta < T_\beta$, the best statistical test in A requires some minimum computation cost P. Thus, P is proportional to the security level of the steganographic file system. If P is infeasible for attackers to acquire, the file system is *computationally secure.*

Following the above reasoning, we propose two approaches to securing steganographic file system against traffic analysis. Oblivious storage is an unconditionally secure approach, which works by completely removing the patterns in data accesses. DataCavern is a is computationally secure approach, which works by minimizing the accuracy of all possible statistical tests, so that P becomes so large that the system is computationally secure.

6.3 Oblivious Storage: An Unconditionally Secure Approach

A naive solution for an unconditionally secure steganographic file system is to scan through the entire storage for each dummy and data access. But this is way too expensive to real applications. In [30], Oded Goldreich et al. have proved that to completely remove the access patterns on a Random Access Memory one need only incur an order of overhead of $O((log\ t)^3)$, where t is the size of the memory. Oblivious RAM is their proposed memory architecture to achieve that performance. Inspired by oblivious RAM, we propose the scheme of *oblivious storage* to conceal users' access pattern in the data traffics of steganographic file system.

We carve out a partition on the raw storage and construct it to be an oblivious storage, which serves as a cache of the file system. The remaining space on the storage is used for the StegFS (steganographic file system) partition.

6.3.1 StegFS Partition

Data is permanently stored in the StegFS partition, which is organized in the same way as that of chapter 5. As shown in figure 6.3, the storage space is partitioned into standard-size blocks, which could be either *data blocks* that contain useful data or *dummy blocks* that contain only random bytes. Both groups of blocks are encrypted and organized into hidden data files and dummy files respectively.

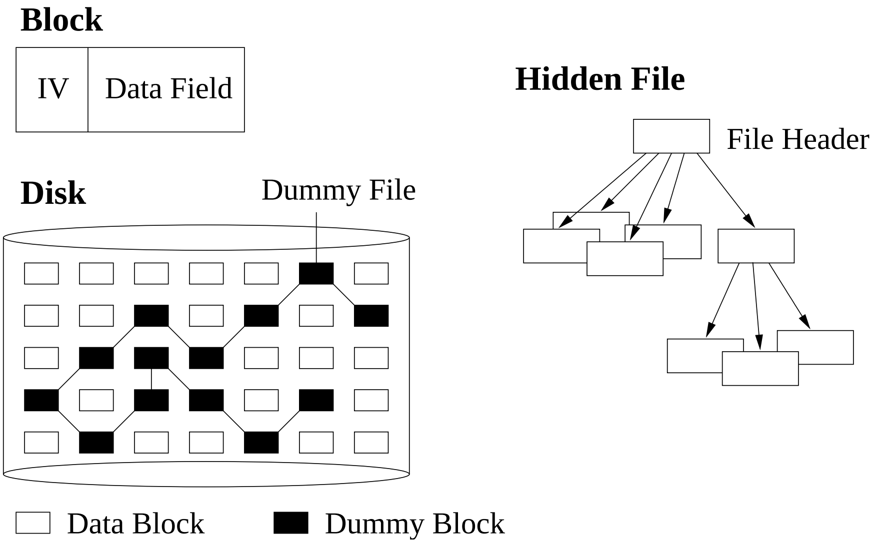


Figure 6.3: Structure of StegFS Partition

Without knowing the access keys, an attacker cannot deduce the existence of hidden files even if he scours through the storage space. Dummy updates are periodically issued to randomly selected blocks to conceal the existence of genuine data updates on hidden files. Genuine data update is conducted in the same way as that in chapter 5, i.e., a data block is relocated to a randomly selected position each time it is updated. Therefore the StegFS partition is able to defend against update analysis attacks.

6.3.2 Oblivious Storage

To counter traffic analysis, all the read accesses on the file system are diverted to the oblivious storage. The oblivious storage serve as a huge cache of the StegFS partition. Whenever a data block is first read from the StegFS partition, it is cached in the Oblivious storage so that the following accesses to the block need only be conducted on the Oblivious Storage alone, except data updates. Oblivious Storage could remove the patterns in users' data accesses so that they could be hidden

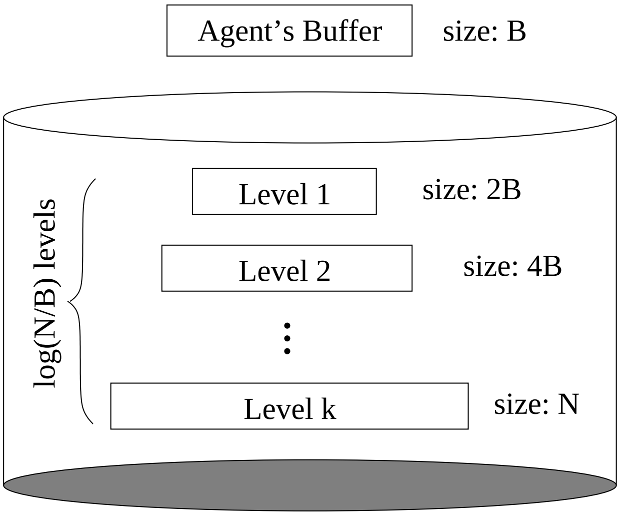


Figure 6.4: Structure of Oblivious Storage

among dummy accesses. But it does not serve as a persistent storage because its data blocks are shuffled frequently.

Figure 6.4 shows the oblivious storage, which is made up of a hierarchy of memories. The first level is twice as large as the agent's buffer cache, and each subsequent level doubles in size until the last level is enough to accommodate all the data blocks that could be read by users. The last level contains all the data blocks that can be found in the oblivious storage, and the other levels may also contain some copies of these blocks. To hide access patterns, the oblivious storage periodically shuffles each level, so that users' access patterns can be distorted and concealed.

6.3.3 Data Processing

Here, we introduce the agent's operations on the StegFS partition and the oblivious storage to hide users' data access patterns. As the StegFS partition is able to hide data updates, we only discuss read accesses here. Whenever there is no user activity, the agent would issue dummy read accesses on randomly selected blocks. When

```
/ * steg-store - StegFS partition whose size is M
    obli-store - oblivious storage whose size is N
    S - the set of data blocks already in obli-store * /

func read_stegfs ()
if a block B1 is required but not in obli-store, then
  Re: generate a random number $X$ that $0 \leq X < M$;
      if $X <$ $sizeof(S)$, then
          randomly pick up a block B2 from S;
          read B2 from steg-store;
          goto Re;
      else
          copy B1 from steg-store to obli-store;
else          // dummy read
      randomly pick up a block $B3$ from steg-store;
      read B3 from steg-store;
func end
```

Figure 6.5: Algorithm: Read on StegFS Partition

there is user request, the agent would read the requested data blocks.

Figure 6.5 gives the algorithm of the read operations on the StegFS partition. When the system starts up, the oblivious storage is empty. Only when the data blocks are accessed, they are copied from the StegFS partition to the oblivious storage and cached in the oblivious storage subsequently. As data blocks are scattered randomly across the storage space and each data block needs to be read only once, the read operation on data blocks in figure 6.5 would look random and does not expose any information to attackers.

The oblivious storage can hide any access pattern on its data blocks by distorting the data accesses into a random process. Therefore, dummy reads and data reads on the oblivious storage can be mixed seamlessly and simply: To satisfy a dummy read, a randomly selected block is retrieved; whereas in the case of a data read, the required block is retrieved. As the oblivious storage exposes no access pattern, attackers cannot distinguish between dummy reads and data reads, and cannot deduce the happening of data read from the observed read operations.

The algorithm for a read operation of the oblivious storage is shown in figure 6.6. To read a data block, the agent first looks in its buffer. If the block is not there, the agent retrieves it from the highest level in the oblivious storage where it can be found. At the same time, it issues a read on a randomly selected block from each of the other levels. After a data block is read, it is added to the agent's buffer until it becomes full, at which time all its blocks are flushed into the first level of the oblivious storage, then all the blocks in that level are re-encrypted and re-ordered (shuffled) to an arbitrary permutation. Similarly, when $Level_i$ of the oblivious storage is full, all its data blocks are flushed into $Level_{i+1}$, and the blocks in $Level_{i+1}$ are then re-encrypted and re-ordered. Consequently, within each level of the oblivious storage, any given data block will be read at most once before the blocks in that level is re-ordered to a random permutation, so that the repeated accesses to any logical block are untraceable. To an attacker, it appears that every time the agent would read a randomly selected block from each level of the oblivious storage, so the probability distribution of dummy accesses and data accesses exactly match in oblivious storage. (Detailed proofs of security could follow those in [30].) According to definition 6.2.1, oblivious storage is unconditionally secure to counter traffic analysis attacks.

For re-ordering a particular level, we should be able to re-order it to a random permutation in a concealed way. (Arguments for this can be found in [30].) Here, we apply the external merge sort algorithm. A hash index is built for each level for locating its data blocks. Write/update operations on data blocks within the oblivious storage can be hidden in the same way as reads. A dummy write/update on a randomly selected block could be conducted by resetting its IV and re-encrypting the block. When a data block is updated on a higher level, it would be automatically

```
/ * steg-store - StegFS partition whose size is M
    obli-store - oblivious storage whose size is N
    S - the set of data blocks already in obli-store * /

func dump (i)
if i = k-1, then
   re-order $level_k$;
   empty $level_i$;
else
    if $level_{i+1}$ is full, then call dump (i+1);
    copy $level_i$ into $level_{i+1}$;
    re-order $level_{i+1}$;
    empty $level_i$;
func end

func read_oblivious (block B1)
if B1 is in the buffer, then
    read B1 from the buffer;
    return;
for i = from 1 to k, do
    if B1 is in $level_i$, then
        read B1 from $level_i$;
        break;
    else
        read a random block from $level_i$;
end loop;
for j = from i to k, do
    read a random block from $level_j$;
end loop;
add B1 to buffer;
if buffer is full, then
    if $level_1$ is full, then call dump (1);
    copy buffer into $level_1$;
    re-order $level_1$;
    empty buffer;
func end
```

Figure 6.6: Algorithm: Read on Oblivious Storage

written to the lower levels during flush. Thus, data integrity could be ensured. The updates/writes would also need to be repeated on the StegFS partition to ensure consistency.

6.3.4 Processing overhead

Let B denote the size of the buffer, and N the size of the lowest level of the oblivious storage. Thus $N = 2^k \times B$, where k is the number of levels. Whenever a data block is to be read, the agent would locate and retrieve a block from every level. This incurs a *retrieving overhead* that is proportional to $2k$. Moreover, the oblivious storage is re-ordered periodically, and this incurs a *sorting overhead.* The ith level of size $2^i \times B$ is sorted at a frequency of once per $2^{i-1} \times B$ reads. If we employ external merge sort, the sorting cost for $Level_i$ is $2^{i+1}B \times \lceil log_B 2^i + 1 \rceil$, and the average sorting cost for each read would be less than $4k \times \lceil log_B 2^k + 1 \rceil$. Therefore, the overall cost for each read in the oblivious storage is $2k + 4k \times \lceil log_B 2^k + 1 \rceil$ where $k = log \frac{N}{B}$. For a normal file system whose N is 20GB and B is 80MB, the average cost is about $14 + 28 \times 2 = 70$ times that of a read operation in a conventional file system. In real-world systems, the sorting overhead is smaller than the retrieving overheads although it incurs more I/Os, as its I/Os are mostly sequential I/Os. This will be further discussed in the performance evaluation subsequently.

To lower the performance penalty, it is possible to relax the security requirement and reduce the storage's height or the frequency that the blocks are re-sorted.

6.3.5 Experiments on Oblivious Storage

We simulated the oblivious storage and conducted performance study to estimate its potential for real world applications. The hardware parameters of our simulation are listed in table 6.1. We construct an oblivious storage on a 2GBytes partition of

Parameter	**Value**
Model of the CPU	Intel Pentium 4
Clock speed of the CPU	1.6 GHz
Type of the hard disk	Ultra ATA/100
Capacity of the hard disk	20 GB

Table 6.1: Physical Resource Parameters

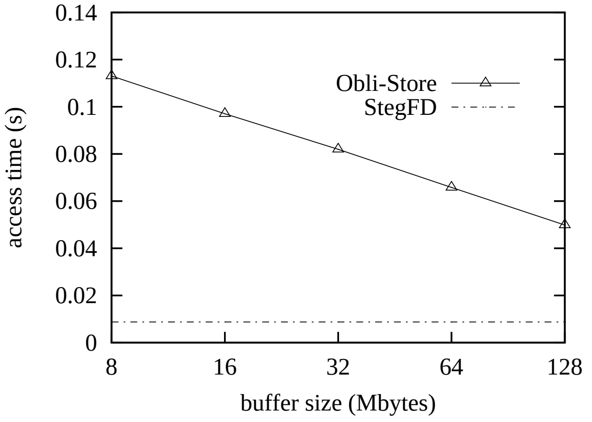
buffer size	8M	16M	32M	64M	128M
height	7	6	5	4	3
overhead	70	60	50	40	30

Table 6.2: Overhead factor vs. Buffer size

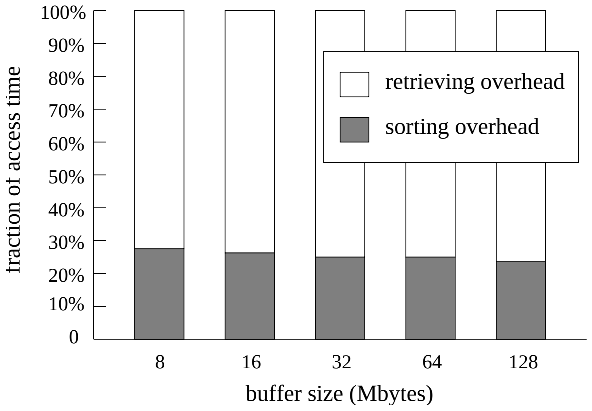
the hard disk, where the size of the last level is 1GBytes. Besides, we use another 1GBytes partition as sorting space for reordering the oblivious storage. The sort algorithm we adopt to resort each level of the oblivious storage is the external merge sort.

We vary the agent's buffer size from 8MBytes to 128MBytes, and see how it affects the oblivious storage's performance. Table 6.2 shows the oblivious storage's height and its overhead factor according to different buffer sizes. When the buffer size is 8MBytes, the oblivious storage contains 7 levels, and its overhead factor is 70, which means it takes averagely 70 I/O operations to satisfy one I/O request. When the buffer size is as large as 128MBytes, its height is reduced to 3, and the overhead factor is reduced to 30.

The first set of experiments reads through the whole oblivious storage to measure the average access time for retrieving a single data block. We compare it against the StegFS in [53]. Figure 6.7 (a) shows the results. The performance of oblivious storage improves linearly with the size of agent's buffer. Generally, retrieving a data block from an oblivious storage spends 5 to 12 times of the cost of retrieving a data block from StegFS. This is better than the theoretic result, for



(a) Access Time vs. Buffer Size



(b) Proportion of Overheads

Figure 6.7: Performance of Oblivious Storage

we utilized sequential I/Os.

As we have mentioned in section 6.3.2, the overhead of the oblivious storage is composed of two parts - retrieving overhead and sorting overhead. In the second set of experiments, we intend to gauge the proportion each of the two overheads takes. Figure 6.7 (b) shows the contrast. Although the sorting overhead costs a larger fraction of I/O operations, it incurs less time. As shown in our results, the sorting overhead occupies less than 30% of the total access time. This is because the sorting process mostly produces sequential I/Os on contiguous data blocks, while the retrieving process performs random I/Os most of the time.

6.4 DataCavern: A Computationally Secure Approach

While oblivious storage is unconditionally secure against traffic analysis, its processing overhead would be unacceptable for many real-time applications. In practice, user would prefer a computationally secure steganographic file system which could achieve a more optimized performance but without losing effectiveness in protecting hidden data. In this section, we introduce such a computationally secure system named DataCavern. First, we outline a conceptual model of DataCavern and discuss its security properties in the face of various traffic analysis attacks. Following that, we expand it into a concrete construction that can be implemented for practical applications.

6.4.1 Conceptual Model

As shown in figure 6.1, DataCavern contains two memories – the raw storage situated in an unsecure domain, and the agent's memory in a secure domain. Similar to the schemes in chapter 3 and chapter 5, the storage holds user data blocks and dummy blocks that are filled with random bytes. The two kinds of blocks are intermixed randomly. The agent's memory is for caching or shuffling blocks, with the aim of transforming the data accesses into steg accesses on the storage that do not display any statistical properties that point to the existence of hidden data.

The conceptual model consists of three components – a *request mixer*, a *shuffler* and a *buffer*, each corresponding to a partition of the agent's memory as shown in figure 6.8.

The request mixer is used to reorder the user requests. When the agent receives a request for a storage block, it first pushes the request into the mixer. When the

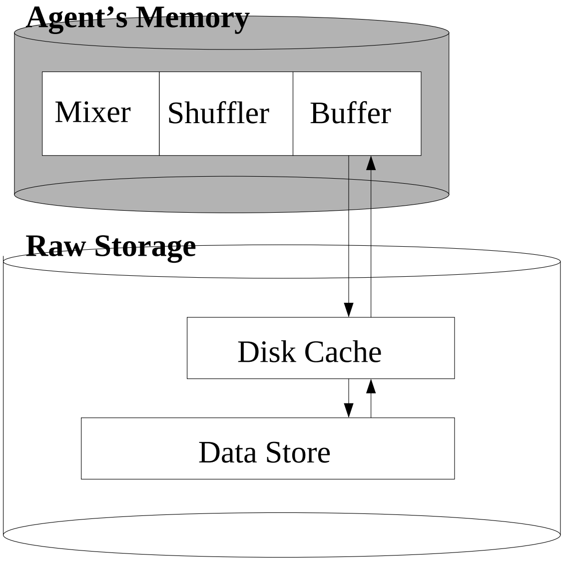


Figure 6.8: Conceptual Model of DataCavern

mixer is full, the agent reorders the requests there and sends them to the storage for execution. If not enough requests are received within some specified time, dummy accesses to randomly selected storage locations are added to the mixer. This procedure, called request mixing, weakens the correlations among the user requests. Although the request mixer increases the expected response time of an access, the maximum throughput of the file system remains the same. A parameter of the request mixer is its size S_{mixer}; a larger mixer randomizes the access sequence better but slows down the response time.

The shuffler is responsible for relocating the data blocks in the storage. Periodically, the agent randomly retrieves some blocks into the shuffler, and swaps their content before writing them back. (In the system implementation section, we will explain how related directory and index entries are updated.) This shuffling procedure relocates blocks covertly, thus concealing any repeated access patterns. There are two parameters for the shuffler – the size S_{shu}, and the shuffle frequency F_{shu} (i.e., the frequency in which shuffling is performed). Naturally, the effectiveness of shuffling improves with S_{shu} and F_{shu}, at the cost of a concomitant increase in I/O overhead in the file system.

After a data block is accessed, it is cached in the buffer in the hope of fulfilling the next request without involving the raw storage. This cuts down the physical I/Os on frequently used pages, thus reducing the risk of data blocks being exposed through uneven access frequencies. The larger the buffer size S_{buffer}, the more uniform the access frequencies on the storage. Buffering also magnifies the impact of shuffling by lengthening the distance between successive accesses on any given data block. (This will be explained in detail shortly.)

Sometimes, the buffer may not be large enough to smoothen the access frequencies sufficiently. Instead of using only main memory as cache, we then carve out a portion of the raw storage for a (much larger) disk cache. As shown in figure 6.8, the raw storage contains a persistent data store, as well as a disk cache for frequently used blocks in the data store. Data blocks in the disk cache are hidden the same way as in the data store. Section 6.4.3 will give a more concrete construction of the disk cache.

6.4.2 Attacks and System Security

Having introduced the DataCavern model, we now examine its security from the perspective of an attacker.

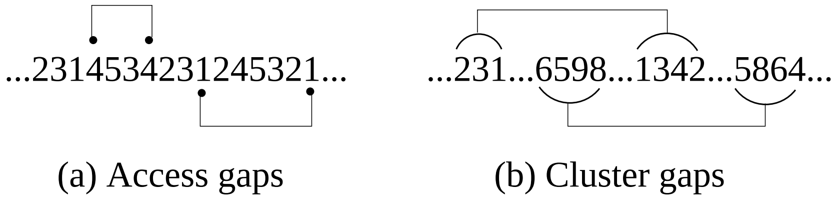
Traffic Analysis Attacks

As explained in Section 6.2.1, the additional protection that DataCavern is designed to offer, over existing steganographic file systems, is against passive attackers who conduct traffic analysis on the channel between the agent and the storage. In other words, an attacker's decision on whether the storage contains any hidden files can be based only on the sequence of accesses observed on the storage. Assuming he is aware that the agent introduces dummy requests into the request mixier especially when there is no user activity, the decision is about whether the access traffic on the storage are randomly generated or genuine. To determine this, the attacker needs to know how genuine user activity may affect the statistical properties of the access traffic, and devise an appropriate statistical test to differentiate between data versus dummy accesses (see section 6.2.2).

The data accesses could exhibit many properties that seldom appear in random dummy traffic. The most common and noticeable properties include:

- **Non-uniformity**. Blocks in the storage are accessed with different frequencies; some very frequently, such as those blocks containing the index of a phone book, while others only rarely. Yet others like the dummy blocks may never be accessed.
- **Sequential pattern**. Blocks containing related information are accessed in a particular order. Examples are sequential scans on a file, and index tree traversals.
- **Clustering**. Blocks containing related information are accessed together. In file systems and databases, data blocks are organized into files or tables, and data accesses can thus be expected to cluster around those logical organizations.

Clustering is a more general form of sequential pattern, in the sense that in a sequential pattern, the blocks concerned are accessed not only as a group but also in a specific order. Since the request mixer already disrupts any ordering within a pattern, we need only to focus on removing non-uniformity and clustering. These two remaining properties can be detected by statistical tests such as the *frequency*



(a) Access gaps (b) Cluster gaps

Figure 6.9: Gaps in Access Sequence

test, *serial test* or *gap test* [43]. The basic idea is to surface the repeated patterns in an observed access sequence, and compare their distribution against the expected distribution of a random access pattern to decide whether they are indeed random. In this paper, we assume that the attacker uses the gap test, which works by examining the gaps between repeated occurrences of the same block access or group of block accesses. However, our proposed scheme works for the other statistical tests too.

Figure 6.9 illustrates the access gaps and the cluster gaps in an access sequence. An *access gap* is the distance between successive accesses to the same block. A *cluster gap* is the distance between adjacent clusters of accesses to the same group of blocks. For example, in figure 6.9(b), block 1, 2 and 3 are always accessed in close proximity, and can be treated as a cluster. In order to identify a cluster gap in an access sequence automatically, we characterize it by two parameters: the *cluster range* is the number of the consecutive blocks within a given cluster, while the *cluster similarity* is the percentage of identical blocks between two clusters. The cluster gap test examines the data accesses in two groups of range R; if the similarity is higher than some threshold S, the gap between the two groups is a candidate cluster gap. Intuitively, the higher the similarity, the more recognizable they are.

Suppose that in a random sequence, a gap (either an access gap or a cluster gap) of length r appears with a probability p_r. Due to non-uniform data accesses or the existence of access clusters, a gap of length r may appear with a different probability p'_r in an access sequence. If so, the observed gaps would deviate from the distribution of p_r. The gap test thus utilizes the Chi-square test to check whether the observed gaps follow the expected distribution of p_r, and from there determine the existence of data accesses. Below are two examples to illustrate the threat.

Example: Access Gap. Suppose that the raw storage contains 3 blocks. In a random access sequence, each block has the same probability of $\frac{1}{3}$ to be accessed. In a data access sequence, the probabilities may be $(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$. Therefore, the probability distribution of the gap length for dummy and data access sequences are as follows:

$$\begin{aligned} \mathit{dummy} &: \left\{ p_0 = \frac{1}{3}, p_1 = \frac{2}{9}, p_2 = \frac{4}{27}, p_3 = \frac{8}{81}, \ldots \right\} \\ \mathit{data} &: \left\{ p'_0 = \frac{3}{8}, p'_1 = \frac{7}{32}, p'_2 = \frac{17}{128}, p'_3 = \frac{43}{512}, \ldots \right\} \end{aligned}$$

Such differences can be picked up easily by the chi-square test, given a sufficiently long access sequence. □

Example: Cluster Gap. In a storage of N blocks, for a cluster of range 4, the probability of another similar cluster (with similarity larger than 50%) occurring in a dummy access sequence should be only $p = \frac{\binom{4}{2}\binom{N}{2}}{\binom{N}{4}} = \frac{72}{(N-2)(N-3)}$, $(N \gg 72)$. However, if each pair of blocks in the storage were always retrieved together in data requests, the probability of a similar cluster occurring in data accesses would become larger than $1/N$, which is almost $\frac{N}{72} \times p$. This affects the gap length significantly. For example, the probability of a cluster gap of length r in a dummy access sequence is

$$p_r = p(1-p)^r$$

but in the data access sequence, it could be

$$p'_r = \frac{N}{72}p(1 - \frac{N}{72}p)^r$$

Again, such differences can be detected readily through the chi-square test. □

The approach that DataCavern takes to counter the gap test is to reduce its accuracy by transforming p'_r and p_r to be as close as possible.

Suppose that p'_r and p_r satisfy $\frac{p'_r}{p_r} \leq 1 + \theta$ (θ is a small value). Based on the chi-square test, we obtain the following inequality if the gap test attack is accurate (i.e., $\alpha < T_\alpha$ and $\beta < T_\beta$ according to Definition 2):

$$n \times \sum_{r=0}^{t} \frac{(p'_r - p_r)^2}{p_r} > \chi^2_{T_\alpha} - \chi^2_{1-T_\beta} \times \big(1 + \theta\big) \tag{6.1}$$

where χ^2_x is the critical value of the chi-square test, and n is the number of observed gaps, which is proportional to the length of the access sequence.

Proof. According to triangle inequality, the following inequality

$$n \times \sum_{r=0}^{t} \frac{(p'_r - p_r)^2}{p_r} \geq \sum_{r=0}^{t} \frac{(N_r - np_r)^2}{np_r} - \sum_{r=0}^{t} \frac{(N_r - np'_r)^2}{np_r}, \quad \Big(where\ n = \sum_{r=0}^{t} N_r\Big)$$

holds for any $\{N_0, N_1, ..., N_t\}$. If $\frac{P'_r}{P_t} \leq 1 + \theta$, then

$$n \times \sum_{r=0}^{t} \frac{(p'_r - p_r)^2}{p_r} \geq \sum_{r=0}^{t} \frac{(N_r - np_r)^2}{np_r} - \sum_{r=0}^{t} \frac{(N_r - np'_r)^2}{np'_r}(1 + \theta) \tag{6.2}$$

In order that the Type I and Type II errors, α and β of the chi-square test do not exceed the maximum tolerable levels T_α and T_β, the attacker must guarantee the following inference

$$\chi'^2 < \chi^2_{1-T_\beta} \rightarrow \chi^2 < \chi^2_{T_\alpha}, \quad \Big(where\ \chi'^2 = \sum_{r=0}^{t} \frac{(N_r - np'_r)^2}{np'_r}\ and\ \chi^2 = \sum_{r=0}^{t} \frac{(N_r - np_r)^2}{np_r}\Big)$$

does not hold. Otherwise, when Type I error does not exceed T_α, Type II error will always be larger than T_β. That is to say, there must be a $\{N_0, N_1, ..., N_t\}$ such that $\chi'^2 < \chi_{1-T_\beta}$ and $\chi^2 \geq \chi_{T_\alpha}$. Applying this to inequality 6.2, inequality 6.1 holds:

$$n \times \sum_{r=0}^{t} \frac{(p'_r - p_r)^2}{p_r} > \chi^2_{T_\alpha} - \chi^2_{1-T_\beta}(1+\theta)$$

□

According to inequality 6.1, a more accurate gap test with lower T_α and T_β requires the following value (left hand side of inequality (6.1)) to be larger:

$$V_t = n \times \sum_{r=0}^{t} \frac{(p'_r - p_r)^2}{p_r} \tag{6.3}$$

Formula (6.3) is a measure of the accuracy of the gap test. The larger the value of V_t, the more accurate the gap test will be in distinguishing between dummy accesses and data accesses. To counter the gap test, DataCavern should thus transform p'_r and p_r to be sufficiently similar, so that an attacker must obtain an impossibly long access sequence (i.e., n is arbitrarily large) in order to satisfy inequality (6.1). This makes DataCavern computationally secure according to Definition 2.

Effect of the request mixer

We expect the request mixer to have varying success in removing the three properties inherent in data accesses. Specifically, it has little effect on the non-uniform access frequencies. An access frequency could still be as high as $\frac{1}{S_{mixer}}$ or as low as $\frac{1}{S_{storage}} \times F_{dummy}$, where F_{dummy} is the fraction of dummy accesses. However, the mixer can transform most of the sequential patterns to weaker clustering patterns that are more difficult to detect, as explained previously. In addition, the mixer is effective in dispersing access clusters; the similarity between the original clusters would be reduced and, at the same time, the cluster ranges would be enlarged. As an example, if the user accesses contain clusters of range 4 and 80% similarity; after passing a mixer of size 8, those clusters would be dispersed to clusters of range 8 and 40% similarity. In general, clusters of range R ($R < S_{mixer}$) and similarity S would be transformed to clusters of range S_{mixer} and similarity $S \times \frac{R}{S_{mixer}}$ by the request mixer.

Effect of the shuffler and buffer

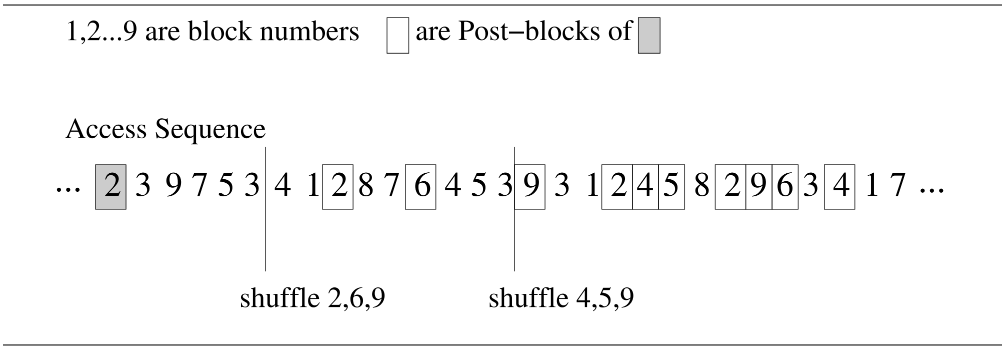


Figure 6.10: Post-blocks in an Access Sequence

In the process of shuffling, S_{shu} blocks are loaded into the shuffler, permuted randomly and then written back. A logical block could thus be moved to many possible physical locations, and become untraceable to attackers. To the gap test, the gap between logical accesses become unidentifiable too, as there could be several possible gaps for it in the physical access sequence. As illustrated in figure 6.10, in an access sequence, there are many physical blocks from where a logical block could be accessed again after shuffling; these are *post-blocks* of the previously accessed block. If a post-block indeed contains the original logical block, it is a *true post-*

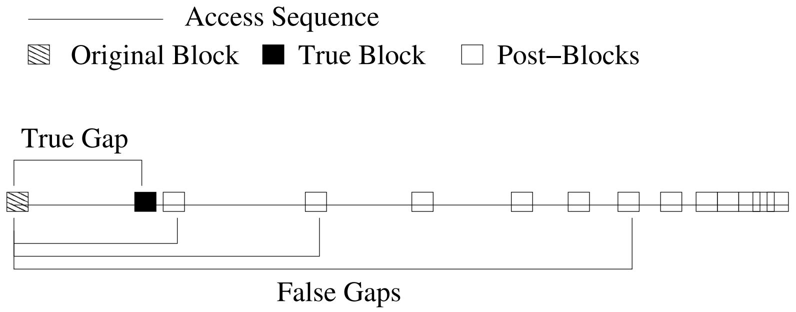
block; otherwise, it is a *false post-block.* Intuitively, as shuffling goes on, a logical block could be relocated to more and more possible places, so its post-blocks would become denser in the access sequence. In other words, $post_i$, the probability that a post-block occurs at the ith block after the original block in the access sequence, increases with i. A larger shuffler and a higher shuffle frequency both accelerate this increase in $post_i$.

After shuffling, the access gaps can only be assessed through the post-blocks. As each post-block can form an access gap with the original block, there could be many possible access gaps in the access sequence, among which only one is the *true gap* that measures the user access pattern while the rest are *false gaps.* Without knowing the true gap, the gap test has to take all the possible gaps into consideration, so its accuracy would be significantly reduced.

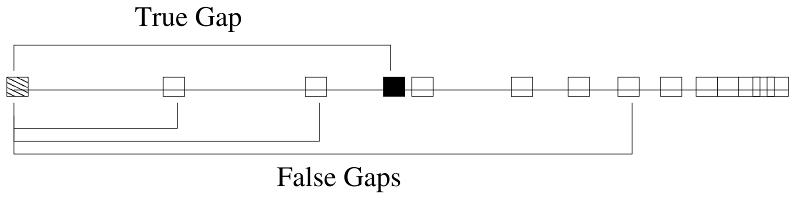
One reasonable variant of the gap test is to consider only the shortest of all the possible gaps. As figure 6.11 shows, if the true post-block emerges before the other post-blocks, the true gap is indeed the shortest one and is exposed by the gap test. Otherwise, the true gap is hidden behind the false gaps and is not detectable by this variant test.

Similarly, there could be several possible cluster gaps in the access sequence, and the gap test can just consider the shortest one. As illustrated in figure 6.12, if the post-blocks are sparse, there would be few false cluster gaps. Thus the true cluster gap is likely to emerge first, and be caught by the gap test. However, when the post-blocks are very dense, false cluster gaps would appear with high probability, and the gap test is likely to miss the true cluster gap.

Denoting the length of the true gap by a, and the length of the shortest false gap by s, we have $p'_r = p_{a=r} \times p_{s\geq r} + p_{s=r} \times p_{a>r}$ and $p_r = p_{s=r}$. Applying formula (6.3), the accuracy of the gap test becomes:

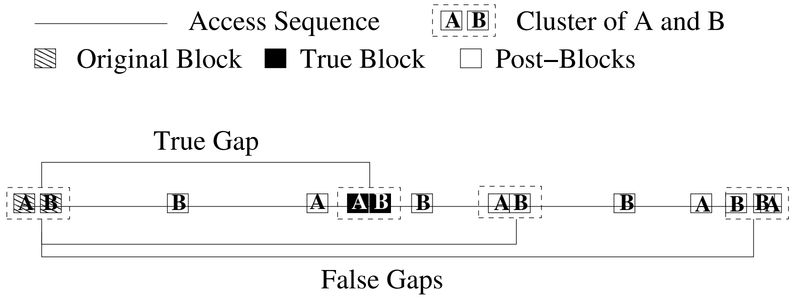


(a) True gap is exposed

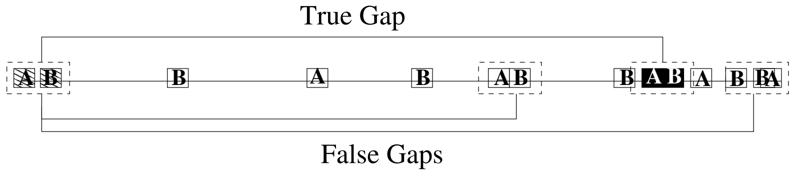


(b) True gap is hidden

Figure 6.11: Hiding Access Gaps



(a) True gap is exposed



(b) True gap is hidden

Figure 6.12: Hiding Cluster Gaps

$$V_t = n \times \sum_{r=0}^{t} \frac{\left[p_{a=r} \times p_{s>r} - p_{s=r} \times p_{a<r}\right]^2}{p_{s=r}} \tag{6.4}$$

$$< 4n \times \sum_{r=0}^{t} \frac{\left[p_{s\geq r} \times p_{a\leq r}\right]^2}{p_{s=r}} \tag{6.5}$$

Proof. Suppose p'_r denotes the probability that the length of the observed shortest gap is r. Suppose the length of the shortest true gap is a and the length of the shortest false gap is S. Then if the observed shortest gap is a true gap, then $r = a$ and $s \geq r$. Otherwise, if it is a fake gap, then $r = s$ and $a > r$. Thus $p'_r = p_{a=r} \times p_{s\geq r} + p_{s=r} \times p_{a>r}$. For random accesses, as there are only false gaps, $p_r = p_{s=r}$. Hence,

$$\begin{aligned} V_t &= n \times \textstyle\sum_{r=0}^{t} \frac{(p'_r - p_r)^2}{p_r} \\ &= n \times \textstyle\sum_{r=0}^{t} \frac{(p_{a=r} \times p_{s\geq r} + p_{s=r} \times p_{a>r} - p_{s=r})^2}{p_{s=r}} \\ &= n \times \textstyle\sum_{r=0}^{t} \frac{(p_{a=r} \times p_{s\geq r} + p_{s=r} \times p_{a\leq r})^2}{p_{s=r}} \end{aligned}$$

Therefore, equation holds.

because

$$\left(p_{a=r} \times p_{s\geq r} + p_{s=r} \times p_{a\leq r}\right)^2 < \left(2 \times p_{s\geq r} \times p_{a\leq r}\right)^2$$

equation also holds:

$$V_t < 4n \times \sum_{r=0}^{t} \frac{(p_{s\geq r} \times p_{a\leq r})^2}{p_{s=r}}.$$

□

For access gaps, the $p_{s\geq r}$ and $p_{s=r}$ in equation 6.4.2 and 6.4.2 are:

$$p_{s\geq r} = \prod_{i=1}^{r-1} (1 - post_i)$$

$$p_{s=r} = \prod_{i=1}^{r-1}(1 - post_i) \times post_r$$

For cluster gaps of range k and similarity q , they are:

$$p_{s\geq r} \approx \prod_{i=1}^{r/k}\Big(1 - \sum_{j=kq}^{k} \tbinom{k}{j}\big[1-(1-post_{ik})^k\big]^j(1-post_{ik})^{k(k-j)}\Big)$$

$$p_{s=r} \approx p_{s\geq r} \times \sum_{j=kq}^{k} \tbinom{k}{j}\big[1-(1-post_r)^k\big]^j(1-post_r)^{k(k-j)}$$

Equation 6.4.2 gives an upper bound on the accuracy of the gap test. The right hand of the equation can be split into two factors: $[p_{a\leq r}]^2$ and $[p_{s\geq r}]^2/p_{s=r}$. The accuracy of the gap test could be reduced by decreasing either factor: (a) According to the above equations, a larger $post_i$ produces a smaller $[p_{s\geq r}]^2/p_{s=r}$. Recall that raising the shuffle frequency accelerates the increase in $post_i$, and therefore reduces the accuracy of the gap test. (b) Caching the frequently requested data blocks lengthens the true gaps and increases $[p_{a\leq r}]^2$. Thus a larger buffer also lowers the accuracy of the gap test. Together, shuffling and buffering can reduce the accuracy of gap tests to an arbitrarily low level.

To summarize, the shuffler and the buffer are intended to hide the true gaps among the false gaps. Increasing the shuffle frequency causes false gaps to occur more rapidly, while enlarging the buffer lengthens the true gaps, thus increasing the probability that the true gaps are hidden behind false gaps. Buffering is particularly important where data accesses so highly skewed that the true gaps of some frequently accessed blocks are too short to be masked by shuffling alone. Besides the gap test, DataCavern can also counter the other statistical tests in a similar way.

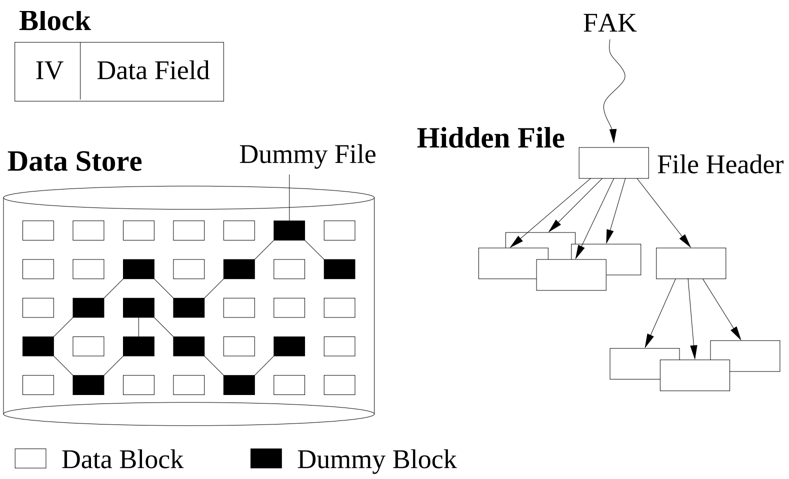


Figure 6.13: Organization of Data Store

6.4.3 System Implementation

Finally, we address various system issues in implementing the DataCavern model in a practical file system.

Data store

Figure 6.13 shows the structure of the data store. Like the construction in [72], our system stores user *data blocks* as well as *dummy blocks* that contain random bytes. Both types of blocks are scattered randomly across the storage volume.

Each block comprises an *initial vector (IV)* and a *data field*. The data field contains real data in the case of a data block, and random bytes if it is a dummy block, and is encrypted by the agent using a CBC (Cipher Block Chaining) block cipher with the IV as seed. Whenever the agent re-encrypts a block, it resets the IV so that the content of the whole block changes. This enables the agent to carry out dummy updates on any block, by simply altering its IV.

A hidden file is a set of data blocks that are organized in a tree structure, with a file header as the root node. The location of the file header can be derived from its

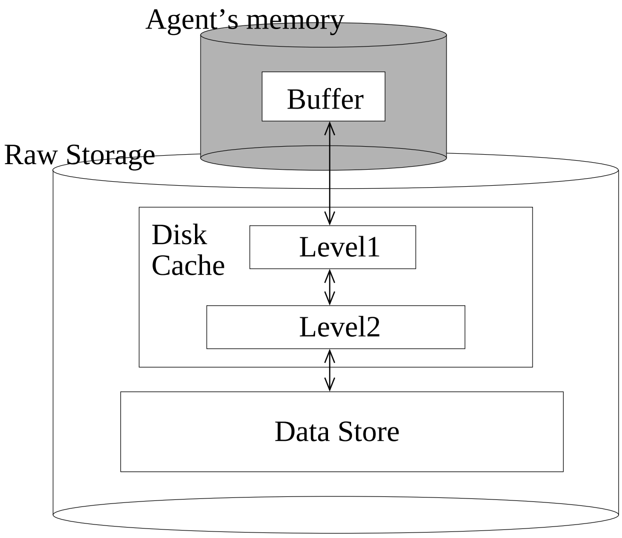


Figure 6.14: Buffer System

access key FAK and the full path. Only with the FAK can the agent reconstruct the file, starting with the file header. Similarly, all the dummy blocks are organized in dummy files.

Each block in the data store, whether data or dummy block, is identified through a file ID and a block ID. Since the blocks are periodically relocated by the shuffler, the agent maintains an index in its memory for identifying the physical location of any block and vice versa.

Buffer system

As explained earlier, the buffer is instrumental in reducing any non-uniformity in the data accesses, and enlarging the access/cluster gaps of frequently used blocks. When the available main memory is not large enough, we construct a hierarchy of disk caches in a partition on the raw storage. The construction of the buffer system is shown in figure 6.14.

Each level of the disk cache has a corresponding index, through which logical blocks can be located. After a data or dummy block is accessed, it will be cached in the agent's buffer. When the buffer is full, a block is eliminated from it, and pushed into level 1 of the disk cache. When level 1 is full, some randomly selected block will be relegated into level 2. With a k-level buffer, this process will be repeated down to level k. Finally, when level k is full, some randomly selected block will be dropped. Thus, while the disk cache as a whole acts as buffer for the data store, internally each layer of the disk cache also treats the level above as its buffer. As dummy blocks are also mixed into each level of the disk cache, the existence of data blocks in the disk cache can be hidden from attackers.

In retrieving a data block, every layer is accessed once: The agent retrieves the data block into the highest level, and issues random accesses in the lower levels. The number of levels and the size of each level are tunable parameters. Assuming a 80-20 rule, $k = 2$ (a 2-level disk cache) should be sufficient for distorting the data access patterns. Our experiment results will confirm its effectiveness shortly.

Data processing algorithms

This section presents the data processing algorithms of the three components of DataCavern.

As described in the conceptual model, the request mixer intermixes the genuine data accesses and random dummy accesses before carrying them out on the storage. The work of the mixer is described in the algorithm in figure 6.15.

While conducting data or dummy accesses on the raw storage, the agent also shuffles the content of the raw storage periodically. The shuffling algorithm is presented in figure 6.16. The data store, level 2 and level 1 are continuously shuffled in a certain frequency. Instead of retrieving individual blocks into the shuffler, the agent can divide the storage into larger *shuffle blocks*, each of which consists of

```
func Mix ()
  set Timer = 0;
  when there is user's request, then
        push the ids of the requested blocks into mixer;
        if mixer is full, then
             reorder the ids in mixer;
             execute the operations in mixer;
  when Timer = 100, then
        fulfill mixer with random ids;
        reorder the ids in mixer;
        execute the operations in mixer;
        set Timer = 0;
func end
```

Figure 6.15: Request Mixing Algorithm

```
func Shuffle ()
  pick $S_{shu}$ blocks from the data store to the shuffler;
  loop
      shuffle;
      write $S_{shu}$ blocks back to the data store,
      simultaneously pick $S_{shu}$ blocks from
      Level2 to the shuffler;
      shuffle;
      write $S_{shu}$ blocks back to the level2,
      simultaneously pick $S_{shu}$ blocks from
      Level1 to the shuffler;
      shuffle;
      write $S_{shu}$ blocks back to the level1,
      simultaneously pick $S_{shu}$ blocks from
      data store to the shuffler;
  end loop;
func end
```

Figure 6.16: Shuffling Algorithm

```
func Retrieve (Addr)
  set Ret = 0;
  if Addr is in the buffer, then
      retrieve *Addr from buffer, set Ret = 1;
      return Ret;
  end if;
  if Addr is in Level1, then
      retrieve *Addr from Level1, set Ret = 1;
  else
      retrieve a randomly selected block from Level1;
  end if;
  if Ret = 1, then
      retrieve a randomly selected block from Level2;
  else if Addr is in Level2, then
      retrieve *Addr from Level2, set Ret = 1;
  else
      retrieve a randomly selected block from Level2;
  end if;
  if Ret = 1, then
      retrieve a randomly selected block from data store;
  else
      retrieve *Addr from data store, set Ret = 1;
  end if;

  push *Addr into buffer;
  if buffer overflows, then
      remove $S_{shu}$ blocks from Level2;
      pick $S_{shu}$ blocks from level1 to the shuffler,
      simultaneously, shuffle and write them into level2;
      Replace the $S_{shu}$ blocks in Level1 with
      $S_{shu}$ blocks in shuffler;
      remove the $S_{shu}$ blocks from shuffler;
  end if;
func end
```

Figure 6.17: Data Retrieval Algorithm

several physical blocks, and retrieve an entire shuffle block each time. This produces sequential I/Os that can improve performance significantly.

The algorithm for processing data accesses is presented in figure 6.17. As data update and retrieval are performed similarly, we only give the algorithm for data retrieval. In this algorithm, block replacement is merged into the shuffling procedure. This prevents attackers from tracing the logical blocks in the disk cache.

6.5 Experiments on DataCavern

To evaluate DataCavern's performance and effectiveness in countering I/O traffic analysis, we have implemented the scheme presented in Section 6.4.3, with the parameters of the request mixer, shuffler and buffer modules being tunable. The implementation is in C++, and mounted directly on a disk volume for the experiments. User requests are simulated as sequential scans of data files, and the activity on the disk volume is logged for subsequent statistical analysis to assess the security of DataCavern. The platform we use for the experiments is an Intel PC, the key parameters of which are listed in Table 6.3.

Parameter	**Value**
Model of the CPU	Intel Pentium IV
Clock speed of the CPU	2.6 GHz
Type of the memory	DDR RAM
Capacity of the memory	1 GB
Type of the hard disk	Ultra ATA/100
Capacity of the hard disk	80 GB

Table 6.3: Physical Resource Parameters

Parameter	**Default**
Block size	4 KBytes
Data store	1 GBytes
Disk cache level2	256 MBytes
Disk cache level1	64 MBytes
Buffer	32 MBytes
Shuffler	32 MBytes
Mixer	128 blocks

Table 6.4: Workload Parameters

6.5.1 Effectiveness in Countering Traffic Analysis

The first set of experiments is designed to study the effectiveness of DataCavern in reducing the accuracy of traffic analysis attacks. For this study, the system is mounted on a disk partition of 1GB. The workload parameters of the experiments are summarized in table 6.4.

We constructed 8192 data files in the data store, each 64Kbytes in size. Assuming 80-20 rule, 80% of the user requests are targeted at 1024 of the files, another 16% at a group of 3072 files, with the remaining 4% going to the other 4096 files. Each user request retrieves an entire file.

Our first experiment is intended to study how shuffling produces post-blocks. We run the simulated workload described above, and record a $2560k$-long access sequence on the data store. We then calculate the probability of occurrence of post-block in the 512K blocks after the original block, i.e., the value for $Post_{1K}$ to $Post_{512K}$. The shuffle frequencies used are 1, 2 and 4 times of $1/S_{shu}$ $accesses$, where S_{shu} is the size of the shuffler. As shown in figure 6.18, the probability of occurrence of post-blocks increases as shuffling goes on, because blocks are relocated to more and more possible places in the storage space. As expected, a higher shuffle frequency leads to a faster increase in the occurrence of post-blocks.

In the next experiment, we aim to estimate the effectiveness of the buffer system.

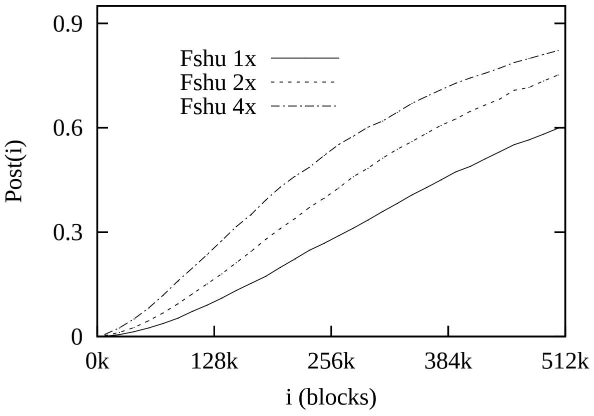
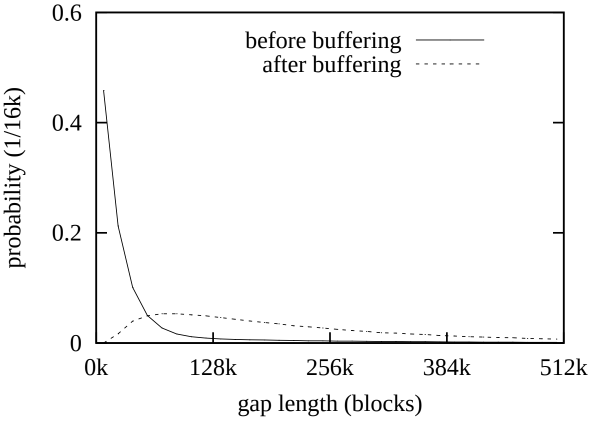


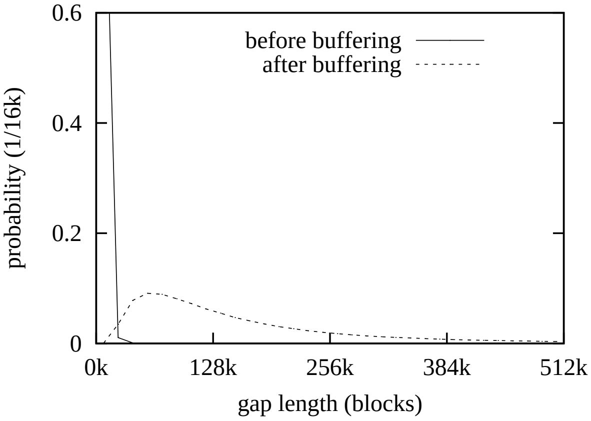
Figure 6.18: Effectiveness of Shuffling

We record a $2560k$-long access sequence on the data store with buffering turned on, and another $2560k$-long access sequence without utilizing the buffer. Figure 6.19(a) charts the probability distribution of access gaps ranging from length 1K to length 512K in the two access sequences, while figure 6.19(b) shows the probability distribution of cluster gaps with cluster range of 128 blocks and similarity of 12.5%, after passing through the request mixer (section 6.4.2). The results confirm that buffering significantly lengthens both access gaps and cluster gaps.

Applying equation (6.4.2), we can derive the V value of any specific gap test from the above results. From the V value and inequality (6.1), we can then compute the minimum length of an access sequence required by an attacker to accurately determine the existence of data accesses. The longer the access sequence, the more expensive the attack, and thus the more secure the file system. The following experiment is intended to study the computational cost of gap test attacks and the security of DataCavern. Assuming that in a gap test attack, an attacker uniformly divides the gap lengths into 4 groups, namely $\{(0, 4K), (5K, 8K), (9k, 12k), (13K, 16K)\}$ (as the smallest false gap rarely exceeds 16K), and uses a degree-3 chi-square test to evaluate the existence of data accesses through the probability distribution of



(a) Access Gap



(b) Cluster Gap

Figure 6.19: Effectiveness of Buffering

the gap length groups. We assume that the attacker's maximum tolerable type I and type II errors are $T_\alpha = 10\%$ and $T_\beta = 10\%$.

Table 6.5(a) shows the approximate minimum computational cost of the access gap test with various shuffle frequencies, with and without buffering. Table 6.5(b) shows the corresponding cost for the cluster gap test. The tables confirm that buffering and raising the shuffling frequency increase the computational cost of the gap test significantly. For example, with buffering and a shuffling frequency of 2×, the cluster gap test needs a very huge access sequence that contains 3×10^{11}

*Cost is represented as length of the access sequence (in number of I/Os)

F_{shu}	1×	2×	4×
with buffer	4×10^{14}	9×10^{16}	2×10^{22}
without buffer	300	1000	20000

(a) Cost of access gap test

F_{shu}	1×	2×	4×
with buffer	1×10^{7}	3×10^{11}	7×10^{23}
without buffer	10000	10000	1×10^{14}

(b) Cost of cluster gap test

Table 6.5: Cost of Gap Test

I/O operations, to achieve an accuracy of 90%. Therefore, with a combination of buffering and shuffling, DataCavern can be fortified very effectively against traffic analysis.

6.5.2 Performance Study

Having demonstrated the effectiveness of DataCavern, we now shift our focus to its performance characteristics. For comparison, we use as baselines the oblivious storage in section 6.3 and the StegFD in chapter 3 that has no protection against traffic analysis. The former will highlight the cost savings achieved by DataCavern, while the latter will provide insight on the overhead incurred to secure the file system. Table 6.6 lists the notation for the various schemes, while Table 6.7 summarizes the workload parameters. Here, we construct data files ranging from 100Kbytes to 1Mbytes in size in the various file systems, and each query retrieves a randomly selected file.

In the first experiment, we profile the performance of the various schemes against different buffer sizes, by varying the agent's memory size from 8 Mbytes to 64

Parameter	**Meaning**
$DataCavern1\|2$	Our proposed scheme, shuffle frequency is $1/S_{shu}$ or $2/S_{shu}$
$DataCavern8M\|16M$	Our proposed scheme, shuffle block size is 8 or 16 Mbytes
ObliStore	Oblivious Storage
$StegFD$	Stegnographic file system in chapter 3

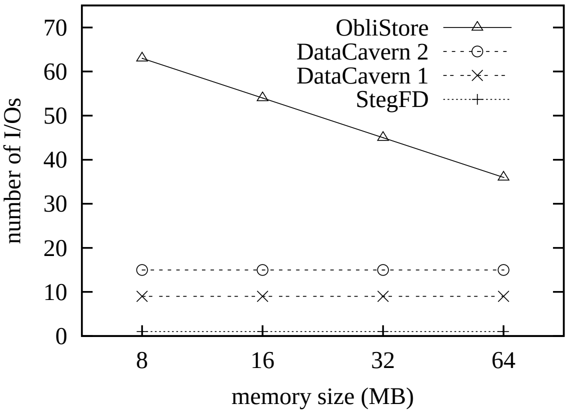
Table 6.6: File System Notations

Scheme	**Parameter**	**Default**
All	Block size	4 KBytes
$DataCavern$	Data store	1 GBytes
	Disk cache level2	256 MBytes
	Disk cache level1	64 MBytes
	Buffer	4 ~ 32 MBytes
	Shuffler	4 ~ 32 MBytes
ObliStore	Bottom level	1 GBytes
	Buffer	8 ~ 64 MBytes
$StegFD$	Disk volume	1 GBytes

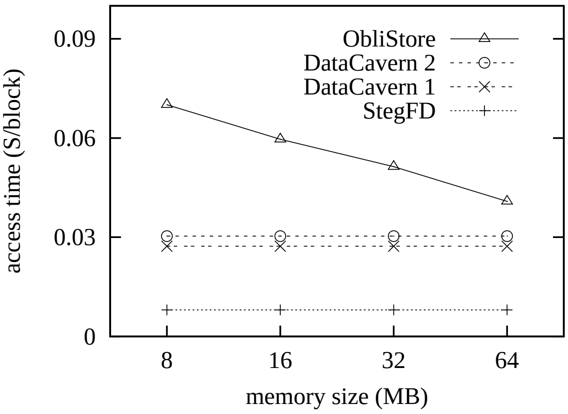
Table 6.7: Workload Parameters

Mbytes. For the oblivious storage, its buffer occupies the entire agent memory, while for DataCavern the agent memory is split equally between the buffer and the shuffler. Moreover, the shuffle frequency of DataCavern is set to $1/S_{shu}$ or $2/S_{shu}$; we shall denote these two versions as $DataCavern$ 1 and $DataCavern$ 2 respectively. And the shuffle block size is fixed at 16Mbytes.

Figure 6.20(a) plots the average I/O overhead of the schemes. The I/O overhead of the oblivious storage is proportional to the height of its buffer hierarchy, which reduces with a larger memory size. For example, in retrieving one data block, the oblivious storage needs to execute on average 36 I/Os with a memory size of 64 Mbytes, and 63 I/Os with 8 Mbytes of memory. In contrast, DataCavern incurs



(a) I/O vs Memory Size



(b) Access Time vs Memory Size

Figure 6.20: Sensitivity to Memory Size

only 9 to 16 times more I/Os. We also note that *DataCavern* 2 is only marginally worse than *DataCavern* 1, because shuffling cost constitutes only a small fraction of the total cost. Figure 6.20(b), which plots the average access time per block, shows the performance of oblivious storage and DataCavern are not as poor as suggested by their I/O overheads, because they are able to take advantage of sequential I/Os. Even then, the two DataCavern variants still manage at least a 200% reduction in access time over the oblivious storage.

To achieve better performance, the agent can parallelize the I/Os of oblivious

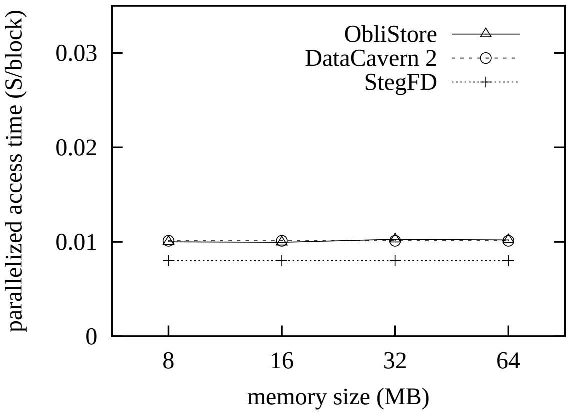
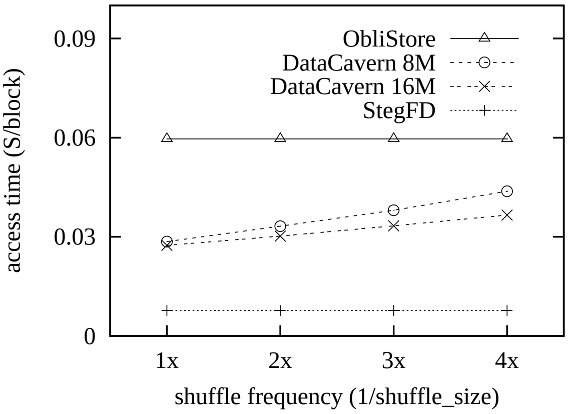


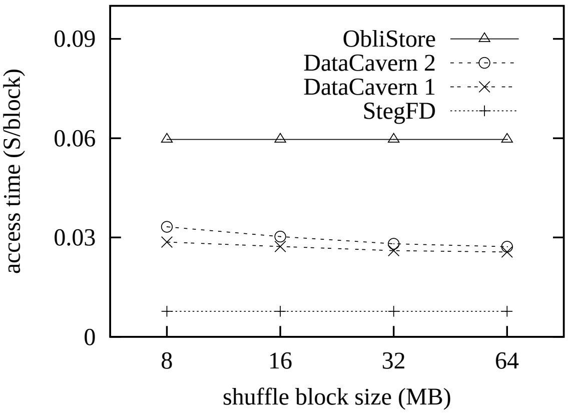
Figure 6.21: Parallelized I/O

storage by distributing its layers across several disks. Similarly, the I/Os of DataCavern can be parallelized by distributing the layers of its disk cache on different disks. If each layer is located on a separate disk, the access time on each disk would be reduced to be very close to StegFD. This is shown in figure 6.21, which is generated by averaging the access time on each layer. However, oblivious storage may contain far too many layers (around 7 layers as explained in [72]), and thus is not practical to be fully parallelized. This is especially so because, for the network storage on different IP addresses, parallelization would significantly increase the communication cost. As DataCavern contains only around 3 layers (including the data store), its parallelization cost would be much more tolerable than that of oblivious storage.

In the next experiment, we study DataCavern's sensitivity to the shuffler by fixing the agent's memory at 16 Mbytes, while varying the shuffle frequency and the shuffle block size. Figure 6.22(a) shows the average access time per block as the shuffle frequency increases from $1/S_{shu}$ to $4/S_{shu}$. Here, the access time of DataCavern degrades only slowly with increasing shuffle frequency; this again confirms that shuffling introduces only a small addition to the total I/O cost. Figure



(a) Access Time vs Shuffle Frequency



(b) Access Time vs Shuffle Block Size

Figure 6.22: Sensitivity to Shuffling

6.22(b) charts the average access time per block, against the shuffle block size. As shown in the figure, a larger shuffle block enables the raw storage to benefit from sequential I/Os, thus resulting in improved performance.

6.6 Summary

In this chapter, we propose two constructions of steganographic file system that are able to defend against traffic analysis on a shared network storage. Both of them mix dummy accesses into users' data accesses to prevent data traffics from exposing the existence of hidden files. Oblivious storage is a construction that could completely hide user access patterns in data traffics, so it is unconditionally secure against traffic analysis attacks. In contrast, DataCavern focuses on reducing the accuracy of traffic analysis to achieve unconditional security. It employs a request mixer to disrupt any logical ordering in the user access activity, a buffer to even out the access frequency of different storage blocks, and a shuffler to minimize repeating access patterns by relocating logical blocks. We show, through analysis and experiments, that both constructions are effective in countering traffic analysis, but DataCavern could achieve much more practical performance than oblivious storage. Plus the scheme for countering update analysis in chapter 5, we believe our work represents a significant advance towards extending the steganographic file system to shared networks that are faced with higher level of risks.

Chapter 7

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss future work on steganographic file system.

7.1 Summary of Contributions

This thesis extended the prototype of steganographic file system in both its theoretical model and its applications. We proposed a model to generalize the purpose, design and security of steganographic file systems. A set of steganographic file systems were then constructed for various application environments that are threatened by different level of risks. The proposed systems were implemented and experiments results showed their effectiveness and potential for real world applications.

Steganographic file system could provide a stronger protection of data than conventional mechanisms such as user access control and encryption by hiding data files within physical storage. However, the existing proposals of system constructions fall short of the requirements of a practical file system that is expected to manage data reliably and efficiently. In this thesis, we first proposed StegFD, a steganographic file system designed for local systems such as PC and server with local storage. It overcomes the weakness of previous system and satisfies the pre-requests of a practical file system through ensuring data integration, preserving an efficient storage utilization and achieving a reasonable performance. We implemented StegFD as a real Linux file system and conducted experiments to evaluate its practicality. We also constructed database components such as B-tree on top of it to evaluate its potential for database applications. Results confirmed that StegFD is a practical system that could be used in real world applications.

Thereafter, we attempted to push the application of steganographic file system beyond local machines to other platforms such as distributed storage, storage area networks (SAN) and storage service providers. As these platforms were confronted with additional security threats that StegFD had not encountered, we had to construct new schemes to handle these various threats.

First, we created a model to generalize the tasks of steganographic file systems and their effectiveness in countering attacks. The model addressed how to divide the activity space of a file system into secure and insecure domains to surface the potential risks and how to determine whether a system construction could enforce adequate security. It was frequently used in the subsequent chapters to design new constructions of steganographic file system to defend various attacks.

Then, a steganographic file system was constructed to counter update analysis attacks, in which attackers attempt to detect hidden files by analyzing the data updates observed on the storage devices. This type of attacks is presented to storage shared on open network, such as Data Grid, SAN and P2P storage, where attackers are able to look into the storage space repeatedly to identify data updates. The counter measure adopted by the proposed system is to continuously issue dummy updates on the storage, so that attackers cannot deduce the existence of hidden files from the observed update operations. By relocating updated data blocks periodically, our system successfully removed the patterns in user updates and achieved unconditional security in countering update analysis.

Finally, we addressed traffic analysis attacks which aims to disclose hidden files through analyzing access patterns in I/O traffics. Sometimes shared storage systems are likely to be compromised and controlled by attackers, who can thus monitor the activities of the storage devices to obtain useful information. A typical application scenario is a storage service provider which is not trusted by user to keep data confidentiality. Thus, a steganographic file system constructed on such a storage is faced with traffic analysis attacks. We proposed two schemes of steganographic file system to defend against traffic analysis attacks. Similar to the idea for countering update analysis, both schemes issue dummy accesses to the storage to hide the existence of users' genuine data accesses. Oblivious storage is a unconditionally secure scheme that could completely remove user access pattern in I/O traffics. DataCavern is a computationally secure scheme that aims to minimize the success rate of traffic analysis attack. We implemented/simulated the proposed schemes and experiment results shows their effectiveness and reasonable performance.

7.2 Future Works

Our future research directions could be classified as follows.

7.2.1 Performance Optimization

In designing the steganographic file systems proposed in this thesis, one criterion is to ensure their performance to be acceptable for real world applications. While the proposed systems such as StegFD could satisfy the basic performance requirements of a practical file system, they are still very inefficient in comparison with regular file systems. Their common bottleneck is that the data blocks of a file are randomly scattered across the storage space, such that the file has to be accessed through random I/O operations, which is much slower than sequential I/Os for today's secondary storage devices. Can we improve the performance of steganographic file system by transforming some random I/Os to sequential I/Os? How would the transformation affect the security of steganographic file system? These questions need to be answered in our future research.

7.2.2 Distributed Steganographic File System

The counter measures against update analysis and traffic analysis enable steganographic file system to be constructed on shared network storage that is exposed to higher risks. However, the proposed schemes such as oblivious storage and Data-cavern all require that data processing be conducted by the agent situated in the local secure domain. The communication cost between the agent and the storage space would be very high. This is acceptable to platforms like storage area network (SAN) which has a high speed connection between server and storage, but unacceptable to platforms like Data grid and P2P networks whose storage is scattered over the internet. So, in our future research, we need to investigate whether it is possible to finish some data processing on the storage side to reduce communication cost. As the data processing activities on the storage side could provide avenue for attackers to detect hidden files, we need also to address the related security problems.

7.2.3 Steganographic DBMS

DBMS has much more complicated structures and functions than a regular file system. There could be many interesting problems if we design a steganographic DBMS using the construction of steganographic file system. First, the access control in DBMS is much finer than that of file system. In steganographic file sytem, a hidden object is either a file or directory. In steganographic DBMS, a hidden object could be a row, a column or a record, which could be too small to be hidden individually. Second, DBMS need to be maintained regularly to keep working efficiently and safely. With hidden objects, maintenance could become much more complicated and difficult. Third, operations in DBMS are usually more costly than that of file system. Examples include the data processing operations like sorting records and joining tables. The performance of current steganographic file systems could hardly satisfy the requirements of DBMS. Hence, it is necessary to do additional performance optimizations to build a practical steganographic DBMS.

Bibliography

[1] The datagrid project. http://eu-datagrid.web.cern.ch/eu-datagrid/.

[2] mp3stego: hide information in mp3 files. http://www.petitcolas.net/fabien/steganography/mp3stego/index.html.

[3] Steganos security suite. http://www.steganos.com/.

[4] *Advanced Encryption Standard.* National Institute of Science and Technology. FIPS 197, 2001.

[5] *Digital Watermarking.* Morgan Kaufmann Publishers, 2001.

[6] *Secure Hashing Algorithm.* National Institute of Science and Technology. FIPS 180-2, 2001.

[7] *Techniques and Applications of Digital Watermarking and Content Protection.* Artech House Publishers, 2003.

[8] M. Abe. Mix-network on permutation networks. In *Advances in cryptology - ASIACRYPT'99*, volume 1716, pages 258–273, Springer-Verlag, 1999.

[9] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding, 2nd International Workshop*, D. Aucsmith, Ed., Portland, Oregon, USA, April 1998.

[10] Ross J. Anderson. Why cryptosystems fail, from communications of the ACM, november, 1994. In *William Stallings, Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

[11] Ross J. Anderson and Fabien A. P. Petitcolas. On the limits of steganography. In *Journal of Selected Areas in Communications*, volume 16, pages 474–481, 1998. Special Issue on Copyright and Privacy Protection.

[12] Oliver Berthold, Hannes Federrath, and Marit Kohntopp. Project "Anonymity and Unobservability in the Internet". In *Workshop on Freedom and Privacy by Design / CFP2000*, 2000.

[13] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. A temporal access control mechanism for database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):67–80, 1996.

[14] Elisa Bertino, Sushi1 Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 94. IEEE Computer Society, 1996.

[15] Matt Blaze. A cryptographic file system for unix. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16. ACM Press, 1993.

[16] Roberta Bragg. The encrypting file system. http://www.microsoft.com/technet/security/topics/crypto/efs.mspx.

[17] Christian Cachin. An information-theoretic model for steganography. *Inf. Comput.*, 192(1):41–56, 2004.

[18] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. *Lecture Notes in Computer Science*, 1592:402+, 1999.

[19] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*, 1995.

[20] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[21] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In *Journal of Network and Computer Applications*, volume 23, pages 187–200, 2001.

[22] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Journal of the ACM*, volume 45, pages 965–982, November 1998.

[23] Ingemar Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Transactions on Image Processing*, 6(12):1673–1687, 1997.

[24] Department of Defence DOD 5200.28-STD. *Trusted Computer System Evaluation Criteria (TCSEC).* National Computer Security Center, 1985. (Orange Book).

[25] Michael J. Evans and Jeffrey S. Rosenthal. *Probability and Statistics: The Science of Uncertainty.* W.H. Freeman, 2003.

[26] Jessica Fridrich and Miroslav Goljan. Practical steganalysis of digital images — state of the art. SPIE-4675:1–13, 2002.

[27] Simson Garfinkel and Gene Spafford. *Practical UNIX and Internet Security, 2nd Edition.* O'Reilly, 1996.

[28] Yael Gertner, Shafi Goldwasser, and Tal Malkin. A random server model for private information retrieval or how to achieve information theoretic PIR avoiding database replication. *Lecture Notes in Computer Science*, 1518:200+, 1998.

[29] James Giles, Reiner Sailer, Dinesh C. Verma, and Suresh Chari. Authentication for distributed web caches. In *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, pages 126–145. Springer-Verlag, 2002.

[30] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *Journal of the ACM*, volume 43, pages 431–473, May 1996.

[31] Oded Goldreich. Secure multi-party computation. *unpublished manuscript*, 2000. http://www.wisdom.weizmann.ac.il/ oded/pp.html.

[32] Ceki Gulcu and Gene Tsudik. Mixing email with babel. In *SNDSS '96: Proceedings of the 1996 Symposium on Network and Distributed System Security (SNDSS '96)*, page 2. IEEE Computer Society, 1996.

[33] Hakan Hacigumus, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In

SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pages 216–227. ACM Press, 2002.

[34] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Electronic Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[35] Frank Hartung and Bernd Girod. Watermarking of uncompressed and compressed video. *Signal Processing*, 66(3):283–301, 1998.

[36] P. Hellekalek. Good random number generators are (not so) easy to find. In *Mathematics and Computers in Simulation*, volume 46, pages 485–505, 1998.

[37] Herodotus. *The History.* J.M. Dent and Sons, Ltd, 1992. translated by George Rawlinson.

[38] J. E. Hoover. The enemy's masterpiece of espionage. *The Reader's Digest*, 48, May 1946.

[39] Andrew Huang. *Hacking the Xbox: An Introduction to Reverse Engineering.* No Starch, 2003.

[40] Neil F. Johnson, Zoran Duric, and Sushil Jajodia. *Information Hiding: Steganography and Watermarking - Attacks and Countermeasures.* Kluwer Academic Press, 2000.

[41] N.F. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. In *Computer, 31(2):26-34*, February 1998.

[42] D. Kahn. *The Codebreakers – The Story of Secret Writing.* Schibner, USA, 1996.

[43] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd ed.* Addison-Wesley, Reading, MA, 1998.

[44] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000. citeseer.nj.nec.com/kubiatowicz00oceanstore.html.

[45] Ping Lin and K. Selcuk Candan. Hiding tree-structured data and queries from untrusted data stores. *Information System Security Journal*, May/June 2004.

[46] David J. C. MacKay. *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, 2003.

[47] E. Mauriello. Tcfs: Transparent cryptographic filesystem. In *Linux Journal No. 40*, August 1997.

[48] David Mazieres. *Self-certifying file system.* PhD thesis, Massachusetts Institute of Technology, USA, May 2000.

[49] A.D. McDonald and M.G. Kuhn. Stegfs: A steganographic file system for linux. In *Proceedings of the Workshop on Information Hiding, IHW'99*, Dresden, Germany, September 1999.

[50] Matt L. Miller, Ingemar J. Cox, Jean-Paul M. G. Linnartz, and Ton Kalker. A review of of watermarking principles and practices. In K. K. Parhi and T. Nishitani, editors, *Digital Signal Processing for Multimedia Systems*, pages 461–485. IEEE, 1999.

[51] J. C. Murphy, D. Dubbel, and R. Benson. *Technology Approaches to Currency Security.* Hopkins Univ. (USA), 1998.

[52] B. Newman. *Secrets of German Espionage.* Robert Hale Ltd, London, 1940.

[53] H. Pang, K.L. Tan, and X. Zhou. Stegfs: A steganographic file system. In *Proceedings of the 19th International Conference on Data Engineering*, pages 657–668, Bangalore, India, March 2003.

[54] H. Pang, K.L. Tan, and X. Zhou. Steganographic schemes for file system and b-tree. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(6):701–713, June 2004.

[55] Fabien A. P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Information hiding — A survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[56] N. Provos and P. Honeyman. Hide and seek: An introduction to steganalysis. *Security and Privacy Magazine, IEEE*, 1(3), 2003.

[57] Niels Provos. Defending against statistical steganalysis. In *Proceedings of the 10th USENIX Security Symposium*, pages 323–336, 2001.

[58] Michael Rabin. How to exchange secrets by oblivious transfer. *Technical Report TR-81, Aiken Computation Laboratory, Harvard University*, 1981.

[59] M.O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. In *Journal of the ACM*, volume 36, No. 2, pages 335–348, April 1989.

[60] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues and open problems. In *Proceedings of Workshop on Design Issues in Anonymity and Unobservability*, volume TR-00-011, pages 7–26, ICSI, July 2000.

[61] R.L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm.* Internet Activities Board, 1992.

[62] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, pages 656–715, 1949.

[63] G. Simmons. The prisoners' problem and the subliminal channel. In *Proceedings of the CRYPTO '83*, pages 51–67. Plenum Press, 1984.

[64] Hal Stern, Mike Eisler, and Ricardo Labiaga. *Managing NFS and NIS, 2nd Edition.* O'Reilly, 2001.

[65] M.D. Swanson, B. Zhu, and A.H. Tewfik. Audio watermarking and data embedding – current state of the art, challenges and future directions. In *Multimedia and Security – Workshop at ACM Multimedia '98*, September 1998.

[66] Mitchell D. Swanson, Bin Zhu, and Ahmed H. Tewfik. Transparent robust image watermarking. In *1996 SPIE Conf. on Visual Communications and Image Proc.*, volume III, pages 211–214, 1996.

[67] P F Syverson, D M Goldschlag, and M G Reed. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy*, pages 44–54, Oakland, California, 1997.

[68] A.S. Tanenbaum and A.S. Woodhul. *Operating Systems: Design and Implementation, 2nd Edition.* Prentice Hall, 1997.

[69] Raymond B. Wolfgang, Christine I. Podilchuk, and Edward J. Delp. Perceptual watermarks for digital images and video. pages 40–51.

[70] Y. Yang, F. Bao, and R. Deng. Improving and cryptanalysis of a key recovery system. In *Proceedings of 2002 Australasian Conference on Information Security and Privacy*, pages 17–24. Springer-Verlag.

[71] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. 1998. citeseer.ist.psu.edu/zadok98cryptfs.html.

[72] X. Zhou, H. Pang, and K.L. Tan. Hiding data accesses in steganographic file system. In *Proceedings of the 20th International Conference on Data Engineering*, pages 572–583, Boston, USA, March 2004.

[73] Jan Zollner, Hannes Federrath, Herbert Klimant, Andreas Pfitzmann, Rudi Piotraschke, Andreas Westfeld, Guntram Wicke, and Gritta Wolf. Modeling the security of steganographic systems. In *Proceedings of the Second International Workshop on Information Hiding*, pages 344–354. Springer-Verlag, 1998.