Hiding Data Accesses in Steganographic File System

Xuan ZHOU[1] HweeHwa PANG[2] Kian-Lee TAN[1]

[1]Department of Computer Science
National University of Singapore
3 Science Dr 2, Singapore 117543

[2]Institute for Infocomm Research
21 Heng Mui Keng Terrace
Singapore 119613

Abstract

To support ubiquitous computing, the underlying data have to be persistent and available anywhere-anytime. The data thus have to migrate from devices local to individual computers, to shared storage volumes that are accessible over open network. This potentially exposes the data to heightened security risks. We propose two mechanisms, in the context of a steganographic file system, to mitigate the risk of attacks initiated through analyzing data accesses from user applications. The first mechanism is intended to counter attempts to locate data through updates in between snapshots – in short, update analysis*. The second mechanism prevents* traffic analysis *– identifying data from I/O traffic patterns. We have implemented the first mechanism on Linux and conducted experiments to demonstrate its effectiveness and practicality. Simulation results on the second mechanism also show its potential for real world applications.*

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, 7]), 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. [10]).

While shared network storage provides the availability needed for ubiquitous computing, it introduces new challenges in data security – Since data reside on open networks, there are several avenues from which an attacker could attempt to steal user data. Even if data is encrypted, the existence of protected data could itself be useful information to the attacker (see [6] for examples); the attacker may try to guess the decryption password, or even coerce the owner to disclose the data. The protected data could be discovered through the directory structures on the storage volume, by looking for changes in between snapshots, or by analyzing I/O traffic patterns.

In [12], we introduced a steganographic file system that foils attempts to deduce the existence of hidden data or to locate hidden data through the directory structure. In this paper, we propose additional mechanisms to protect against attacks initiated through analyzing data accesses from user applications. The first mechanism is intended to counter attempts to locate data through updates in between snapshots – in short, *update analysis*. It works by relocating data blocks systematically, and by introducing updates to dummy blocks. To prevent *traffic analysis* – identifying data from I/O traffic patterns – we design an oblivious storage that employs a multi-tiered buffer to mask off any regular I/O patterns inherent in the user applications.

The mechanisms are constructed to balance between three different objectives: (a) security: an attacker cannot deduce whether the blocks involved in any observable updates or traffic patterns contain genuine data; (b) integrity: the data relocations and 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.

The remainder of this paper is organized as follows. Section 2 summarizes related work on steganographic file systems and traffic analysis. Section 3 gives an overview of update and traffic analysis attacks, and defines our system model. We introduce the mechanism to guard against update analysis in Section 4, while Section 5 presents our

oblivious storage for countering traffic analysis. Following that, Section 6 describes the system implementation and performance evaluation results. Finally, Section 7 concludes the paper and discusses directions for future work.

2. Related Work

This section first summarizes previous work on steganographic file systems, then reviews some related work on traffic analysis.

2.1. Steganographic File system

User access control and encryption are traditional protection mechanisms for file systems. However, as they leave evidence of the existence of valuable data, they may prompt an adversary to attempt to circumvent the protection, or even to coerce the owner into disclosing the access key. The steganographic file system provides a higher level of protection, by hiding the very *existence* of the files. This kind of file system grants access to the protected files only if the correct access keys are supplied. Without them, an adversary could get no information about whether the protected directory/file ever exists, even though the adversary 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. With such a file system, if the owner is compelled by an adversary to disclose data, he can deny their existence without arousing suspicion. This is called *plausible deniability*, and is especially useful in safeguarding sensitive information.

Classical approaches to steganography are concerned with embedding relatively small messages within large cover texts, e.g. using the least significant bit of the pixels in an image to hide copyright information. While some products apply these approaches directly to secure data files, e.g. DriveCrypt [2] is capable of hiding entire disk volumes in music files, the resulting overhead in storage space is unacceptable for a general-purpose file system that needs to hold large volumes of data with high space usage efficiency.

In [6], Ross Anderson et al presented the first two schemes for a steganographic file system that hide data directly on a raw disk volume. However, both schemes proposed in this work are not satisfactory for real-life application due to their high processing overhead and/or risks of data loss. In [12], we proposed a practical StegFS solution that overcame those limitations.

In a StegFS, a number of randomly selected blocks are initially filled with random data and abandoned by the system. After that, the data blocks of useful files, which are encrypted under the files' access keys, are scattered across the storage space in such a way that they can only be located through the access keys. Therefore, an attacker without the files' access keys cannot distinguish between useful blocks of hidden files and abandon blocks, and thus cannot deduce the existence of the files. As StegFS was designed for local storage devices, it does not address the additional risk of update analysis or traffic analysis that shared network storage encounters.

2.2. Traffic Analysis and Related Problems

Even with a steganographic file system, an attacker who is monitoring the storage might be able to analyze the patterns of the update or data traffic activities, and from there deduce the existence of hidden files. This is the *traffic analysis* problem [14].

Traffic analysis has been studied extensively in the context of privacy-providing systems, such as the MIX networks [5, 13]. While all these techniques serve to prevent private information from being released to adversaries through the data traffic or access patterns, different mechanisms are adopted according to the peculiar objectives and requirements of individual systems. Two privacy protection mechanisms that could be adapted to solve our problems are *oblivious RAM* [9] and *private information retrieval (PIR)* [8].

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 [8, 11] 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 generic file system.

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 [9], the oblivious RAM's processing overhead is reduced to $O((log\ t)^3)$ where t is the number of computation steps of the RAM. Our proposed counter-measure against traffic analysis, oblivious storage, is inspired by the oblivious RAM (see Section 5.1).

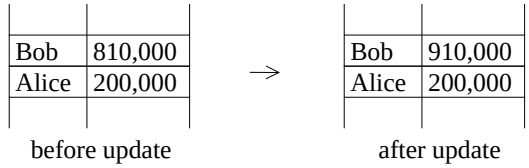
3. Overview of Update and Traffic Analysis

In this section, we first give an overview of the problems of update analysis and traffic analysis. Following that, we outline the model for the steganographic file system that we designed to counter those attacks.

update from user's view

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

update from table's view



update from disk's view

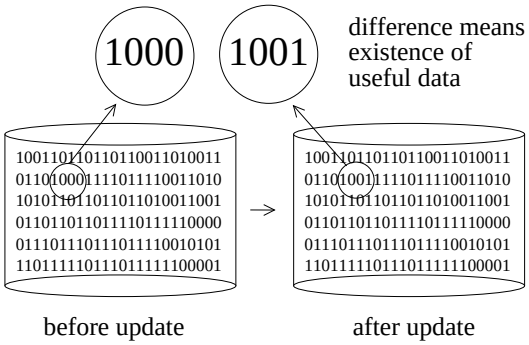


Figure 1. Hidden Data is Exposed by Update

3.1. Problem Definition

Existing steganographic file systems are primarily designed to ensure that an attacker cannot easily deduce the existence of hidden files by examining the directory structures in local storage devices. They do not address the additional risk faced by shared network storage. Specifically, if an attacker can compare consecutive snapshots, 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 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 coerce the user to disclose the table being updated.

Besides update analysis, an attacker may also be able to gain access to the activity log, or even trap the requests at runtime, then analyze the I/O operations for hints of hidden files. This second attack is commonly known as *traffic analysis*.

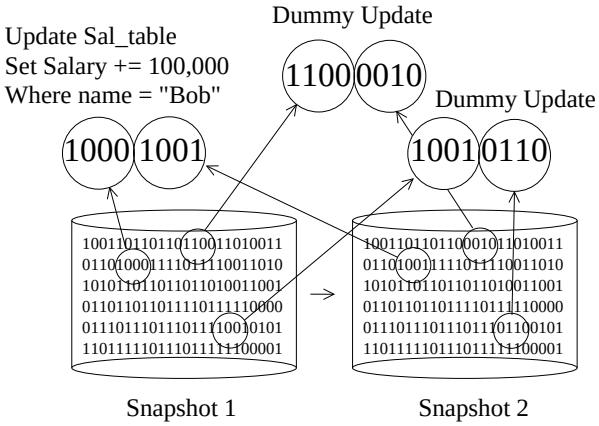


Figure 2. Effect of Dummy Accesses

To hide data accesses from the above attacks, we could issue a stream of purposeless accesses on the storage. If these *dummy accesses* could be made to appear indistinguishable from the data accesses, they would prevent the attacker from deducing the existence of hidden data from any updates or I/O traffic. As figure 2 shows, since the system has been conducting dummy updates on the storage periodically, the attacker cannot tell whether a changed block is due to a real or dummy update. This is the basic idea for the extensions to our steganographic file system.

3.2. System Model

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

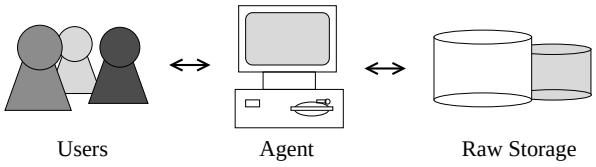


Figure 3. Model of Steg File System

3.2.1. System. Figure 3 shows the model. The users on the left hand of figure 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 I/O operations on the raw storage. Therefore, any attacker who might be monitoring the raw storage would not be able to isolate user activities from dummy requests, and thus cannot deduce the existence of hidden files.

Commonly, the users and the agent can communicate through some trusted channels deliberately, whereas the raw storage is the shared resource in the network, which is always in an unsecured environment. Hence it is necessary to protect the traffic between the agent and the raw storage. Common practical scenarios for such a model include shared storage area networks (SAN), data grids [1, 7], peer-to-peer storage platforms [10], and storage services hosted by external data centers.

3.2.2. Attacker. Attackers of such a steganographic file system are classified into two groups. The first group of attackers are able to scan the whole raw storage repeatedly, so they can identify any updates conducted on the raw storage. The second group of attackers are able to observe the I/O requests between the agent and the storage, either from the activity log or by trapping requests directly at runtime, and could deduce the existence of hidden files through traffic analysis. Both groups 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. We assume that the users can communicate with the agent through a secure channel. We also assume that the agent is a computer that is properly shielded from external probes.

3.2.3. 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 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.

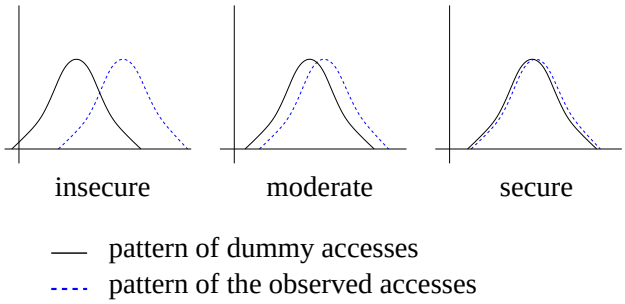


Figure 4. Effectiveness of Hiding Accesses

3.2.4. 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 StegFS in [12]. At the same time, the agent should hide user accesses by mixing in dummy traffic. As illustrated in figure 4, the data access pattern, should appear the same as the pattern of dummy accesses. (The access pattern is usually expressed as the probability distribution of the accesses.) Otherwise, an attacker may be able to isolate the data accesses and prove the existence of hidden files. Here we give the definition of security for hiding data accesses in a steganographic file system.

Definition 1: Let X denote the sequence of accesses the agent performs on the raw storage. Its probability distribution is P_X. Y denotes the set of access 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 accesses.) A system is *secure* if and only if, whatever Y is, $P_{X|Y}$ and $P_{X|\emptyset}$ are so similar that it is computationally infeasible for an attacker to distinguish between them from a sufficiently large set of samples. A system is *perfectly secure* if and only if $P_{X|Y}$ and $P_{X|\emptyset}$ are exactly the same. □

4. Mechanism to Counter Update Analysis

This section presents the first mechanism to equip our steganographic file system to counter update analysis, where attackers might take multiple snapshots of the raw

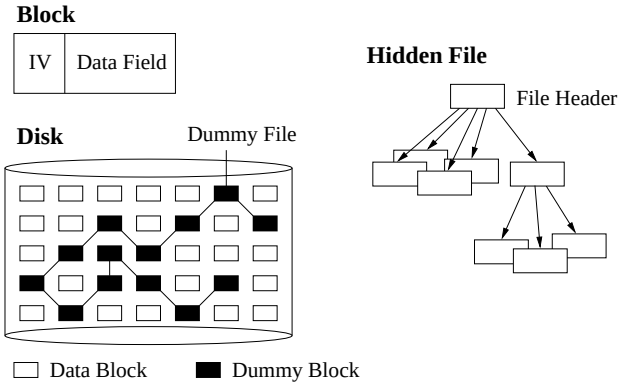


Figure 5. File System Construction

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.

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

4.1. 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.

4.1.1. Data blocks. Figure 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 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.

4.1.2. 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 StegFS in [12]. 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.

4.1.3. 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.

4.1.4. 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 6, combines the procedures for dummy update and data update.

```
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 6. Update Algorithm

Now, we prove that this scheme is perfectly secure.

Proof: For a data update, each block in the storage space has the same probability of being selected to hold the new data. Hence the data updates produce random block I/Os, and follow exactly the same pattern as the dummy updates. 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. According to the definition in Section 3.2.4, the scheme is perfectly secure. Without knowing the agent's encryption key, attackers can get no information on the hidden data no matter how long they monitor the raw storage. □

4.1.5. 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.

4.2. 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.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

5. Mechanism to Counter Traffic Analysis

Having tackled update analysis, we turn our attention to the threat of traffic analysis, carried out by attackers who can observe the I/O traffic between the agent and the raw storage. To prevent the read and write operations from exposing the existence of hidden files, the agent needs to mix in dummy reads and writes on the raw storage to conceal the data traffic.

Our solution to the traffic analysis problem works by randomizing the physical I/Os that arise from data reads and writes. Write operations can be masked in the same way as updates: a data block is written to a randomly selected position each time it is updated, so that the data writes follow the pattern of dummy writes. Read operations are more difficult to hide, since data have to be retrieved from specific locations and the read pattern is determined by the user applications. Thus, we need another way to mould the read pattern into the pattern of dummy reads. A naive solution is to scan through the entire storage for each dummy and data read operation. This is perfectly secure but way too expensive.

In this section, we propose an *oblivious storage* scheme for hiding data read operations, which draws inspiration from the oblivious RAM [9]. We carve out a partition on the raw storage and construct it to be an oblivious storage (shown in figure 7), which serves as a cache of the file system. The remaining space on the storage is used for the StegFS (steganographic file system) partition. All the hidden files and dummy files are stored in the StegFS partition

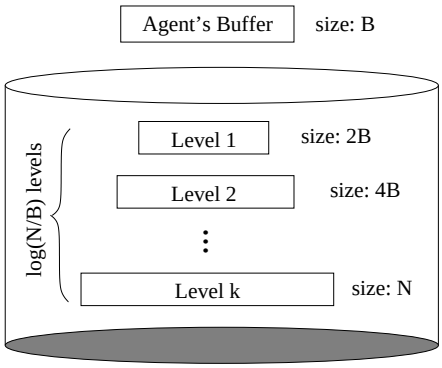


Figure 7. Structure of Oblivious Storage

as before; the update operations are performed in this partition as well. The difference is that here read operations are diverted to the oblivious storage. As both the oblivious storage and the StegFS partition are parts of the raw storage, both two of them are visible to attackers.

As will be explained shortly, the oblivious storage works by shuffling data blocks frequently. Hence it can only serve as a cache, but not the persistent storage because the FAKs of hidden files are not available before their owners log in, and until then the agent has no way of updating their headers to reflect any block relocations. This is why we need a separate StegFS partition for persistent storage.

5.1. System Construction

Here, we introduce the agent's operations on the StegFS partition and the oblivious storage.

5.1.1. StegFS partition. The StegFS partition is constructed in the same manner as the one that can hide updates. Update operations are conducted in the same way as before. But read operations are conducted at most once for each data block, as the retrieved blocks will be cached in the oblivious storage subsequently. Dummy reads are also mixed in to conceal the real reads.

Figure 8 (a) 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 required by users, they are copied from the StegFS partition to the oblivious storage. Each dummy read operation is random. As data blocks are scattered randomly across the storage space and each data block needs to be read only once, the real read operation in figure 8 (a) is also random. Therefore, the process of data retrieval from the StegFS partition looks like a random process and does not expose any information to attackers.

5.1.2. Oblivious storage. 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 following paragraphs give an overview of the construction and operations of the oblivious storage.

Figure 7 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.

The algorithm for a read operation of the oblivious storage is shown in figure 8 (b). 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 there 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 an attacker, it appears that every time the agent would read a randomly selected block from each level of the oblivious storage, so there is no way to deduce whether the observed read operation serves for a data read or a dummy read. Proof of the security of the oblivious storage can be carried out similarly as that of the oblivious RAM in [9]. As the oblivious RAM is computationally secure, this scheme of oblivious storage satisfies our *secure* definition in section 3.2.4.

Write/update operations on data blocks within the oblivious storage can be hidden in the same way as reads. The writes would also need to be repeated on the StegFS partition to ensure consistency.

A secondary hash index is built for each level for locating its data blocks. If these indices are too big to be accommodated by the agent's memory, the agent can encrypt them using a temporary key and store each of them in the front of the corresponding level. Each hash index has to be rebuilt

```
/ * 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 ≤ 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
```

(a) Read on StegFS Partition

```
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
```

(b) Read on Oblivious Storage

Figure 8. Read Algorithms

whenever the corresponding level is re-ordered. The key for the hash index is composed of the block's logical address and a random number generated when the hash index is re-built. Therefore, attackers could not detect anything from the accesses to the indices.

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 [9].) Here, we apply the external merge sort algorithm.

5.2. 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. Optimization issues such as this will be addressed in future work.

6. Implementation and Evaluation

We have implemented a steganographic file system based on the volatile agent scheme introduced in Section 4.2. We also simulated the non-volatile agent and the oblivious storage, and conducted experiments to evaluate their performance. This section begins by describing the implementation, then presents results from some interesting experiments.

6.1. System Implementation

We implemented our file system in Linux. Figure 9 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 [10]. We use AES [3] for the block cipher, and the pseudo-random number generator is constructed from SHA256 [4].

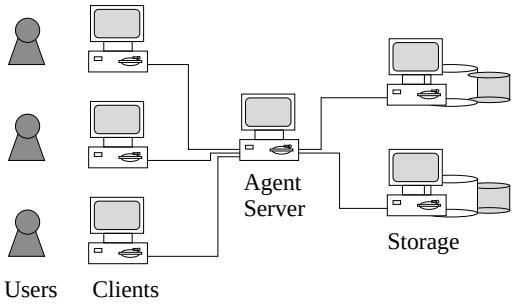


Figure 9. System Architecture

6.2. Experiments on Countering Update Analysis

We first conduct experiments to evaluate the I/O performance of the schemes that can counter update analysis (see Section 4). The platform we used for the experiments is an Intel PC, whose key parameters are listed in Table 1. And Table 2 summarizes the workload parameters. For comparison, we use as baselines the native Linux file system and the previous steganographic file system in [12]. The notations for the various file systems are shown in Table 3.

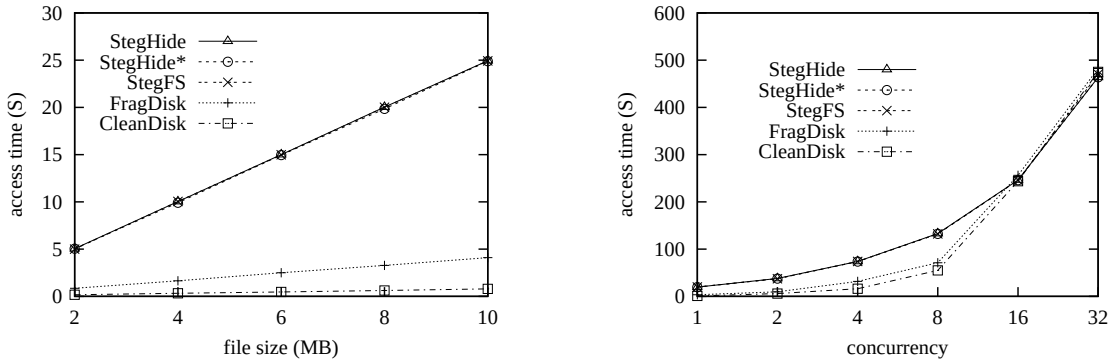
Parameter	Value
Model of the CPU	Intel Pentium 4
Clock speed of the CPU	1.6 GHz
Type of the hard disk	Ultra ATA/100
Capacity of the hard disk	20 GB

Table 1. Physical Resource Parameters

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 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



(a) Sensitivity to File Size (b) Sensitivity to Concurrency

Figure 10. Performance on Data Retrieval

Parameter	Meaning
$StegHide$	Construction 2: volatile agent
$StegHide*$	Construction 1: non-volatile agent
$StegFS$	The former StegFS in [12]
$CleanDisk$	A fresh Linux file system
$FragDisk$	A well-used Linux file system with fragmentation

Table 3. Algorithm Indicators

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 Figure 6. *StegFS* is our former steganographic file system introduced in [12]. *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.

6.2.1. 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 10 (a) shows the access times of retrieving files of different sizes in a single user environment. Figure 10 (b) shows the sensitivity of the access time to the number of concurrent users.

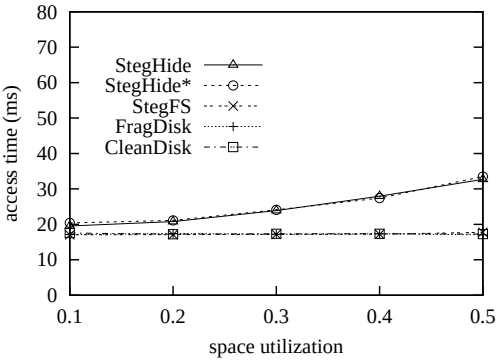
StegHide, StegHide* and StegFS display similar performance in data retrieval, since their data blocks are distributed across the storage in the same manner. In a single user environment, FragDisk and CleanDisk outperform the three steganographic file systems, as they can perform sequential I/O on their contiguously located data blocks. But their advantage diminishes as the degree of concurrency increases. As shown in figure 10 (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.

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

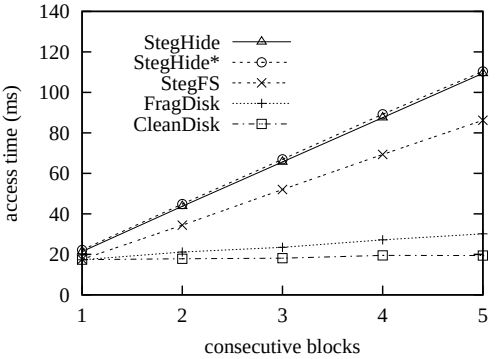
As our system intends to counter update analysis, it introduces extra overhead to update operations. This overhead is affected by the space utilization, which is explained in Section 4.1.5. Thus we first study the sensitivity of the update performance to space utilization. We vary the space utilization 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 11 (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%.

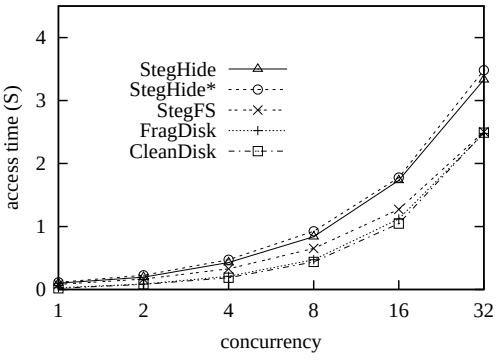
Sometimes an update is performed on a large range of data which may occupy more than one consecutive data blocks. In the second set of experiments, we study the sensitivity of update performance to the number of consecutive blocks being updated. We fix the space utilization of StegHide and StegHide* to 25%, and vary the update range from 1 to 5 data blocks. The results are shown in figure 11 (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.



(a) Sensitivity to Space Utilization



(b) Sensitivity to Update Range



(c) Sensitivity to Concurrency

Figure 11. Performance on Update

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

6.3. Experiments on Oblivious Storage

We also simulated the oblivious storage and conducted experiments to estimate its potential for real world applications. The hardware parameters of our simulation are listed in table 1. We construct an oblivious storage on a 2GBytes partition of 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 is the external merge sort.

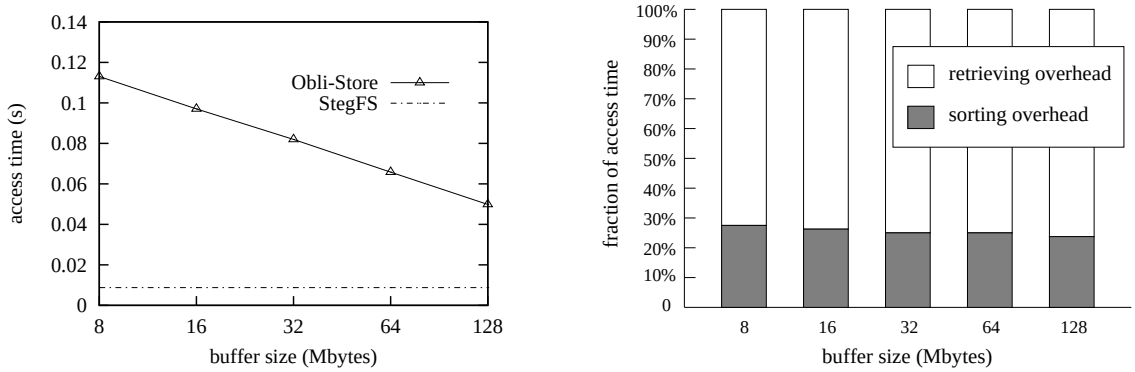
We vary the agent's buffer size from 8MBytes to 128MBytes, and see how it affects the oblivious storage's performance. Table 4 shows the oblivious storage's height and its overhead factor according to different buffer sizes. When the buffer size is 8MBytes, the oblivious storage contains 7 levels, and 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.

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

Table 4. Overhead factor vs. Buffer size

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 [12]. Figure 12 (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 we utilized sequential I/Os.

As we have mentioned in section 5.1, 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 12 (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



(a) Access Time vs. Buffer Size

(b) Proportion of Overheads

Figure 12. Performance of Oblivious Storage

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.

7. Conclusion

In this paper, we propose two mechanisms for a steganographic file system to mitigate the risk of attacks initiated through analyzing data accesses from user applications. Both of them introduce dummy accesses into the system to conceal the existence of real data accesses. The first mechanism is intended to counter *update analysis*, which attempts to locate data through updates in between snapshots. Two constructions are built for this mechanism, one for a non-volatile agent and the other for a volatile agent. The second mechanism aims to prevent *traffic analysis* that tries to identify data from I/O traffic patterns. It utilizes an oblivious storage that can mask off any perceivable I/O patterns inherent in the user applications. We implemented the volatile agent scheme to hide updates, and simulated the non-volatile agent scheme and the oblivious storage mechanism. Our performance study showed their reasonable performance and potential for real world applications.

For future work, we plan to optimize the oblivious storage scheme. We will also extend the proposed mechanisms to various kinds of networked storage systems, such as a P2P storage that does not have a centralized controller.

References

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

[2] Drivecrypt secure hard disk encryption. http://www.drivecrypt.com.

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

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

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

[6] 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.

[7] 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.

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

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

[10] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000. citeseer.nj.nec.com/kubiatowicz00oceanstore.html.

[11] R. Ostrovsky and V. Shoup. Private information storage. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 294–303, New York, May 1997.

[12] H. Pang, K. 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.

[13] C. Rackoff and D. R. Simon. Cryptographic defense against traffic analysis. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 672–681, San Diego, California, May 1993.

[14] 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.