Traffic Analysis Attacks on a Continuously-Observable Steganographic File System

Carmela Troncoso, Claudia Diaz, Orr Dunkelman, and Bart Preneel

K.U.Leuven, ESAT/COSIC,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
firstname.lastname@esat.kuleuven.be

Abstract. A continuously-observable steganographic file system allows to remotely store user files on a raw storage device; the security goal is to offer plausible deniability even when the raw storage device is continuously monitored by an attacker. Zhou, Pang and Tan have proposed such a system in [7] with a claim of provable security against traffic analysis. In this paper, we disprove their claims by presenting traffic analysis attacks on the file update algorithm of Zhou *et al.* Our attacks are highly effective in detecting file updates and revealing the existence and location of files. For multi-block files, we show that two updates are sufficient to discover the file. One-block files accessed a sufficient number of times can also be revealed. Our results suggest that simple randomization techniques are not sufficient to protect steganographic file systems from traffic analysis attacks.

1 Introduction

The goal of a steganographic file system is to protect the user from compulsion attacks, where the user is forced to hand over file decryption keys under the threat of legal sanctions or physical intimidation. In order to achieve this goal, the steganographic file system must conceal the files it stores, so that the user can plausibly deny their very existence.

Several proposals in the literature provide plausible deniability to the user against attackers that take one or more snapshots of the raw storage. To the best of our knowledge, the proposal by Zhou *et al.* [7] is the only one that claims to resist attackers who can continuously monitor accesses to the storage. It relies on dummy updates and relocations of data that are supposed to conceal accesses to the hidden files.

Zhou *et al.* [7] present two separate mechanisms for reading and updating files; we present traffic analysis attacks which are effective against the file update mechanism. Our attacks succeed in revealing the existence and location of hidden files, depriving the user of plausible deniability. We describe the theory behind the attacks, and the impact of the system's parameters on their effectiveness. We have also simulated the attacks, and obtained empirical results that confirm our theoretical analysis.

T. Furon et al. (Eds.): IH 2007, LNCS 4567, pp. 220–236, 2007.

The rest of this paper is organized as follows: in Sect. 2, we summarize previous work on steganographic file systems. Section 3 describes the update algorithm of [7]. Section 4 explains theoretically how to attack the system. The empirical results of our implementation are presented in Sect. 5. We present our conclusions in Sect. 6, where we also suggest lines for future research. Finally, Appendix A shows the attack algorithms that have been used in our implementation.

2 Related Work

The concept of a steganographic file system was first proposed by Anderson, Needham and Shamir in [1] together with two implementations. The first approach consists of hiding the information in cover files such that it can be retrieved by XOR-ing a subset of them. In the second approach, the file system is filled with random data and the real files are hidden by writing the encrypted blocks in pseudo-random locations derived from the name of the file and a password.

Kuhn and McDonald proposed StegFS in [5]. They use a block allocation table (BAT) to have control over the file system contents. In this table, each entry is encrypted with the same key as the block it corresponds to. The file system is organized in levels in such a way that opening a level (decrypting all the entries in the BAT) opens also all the lower levels.

There is another proposal called StegFS by Zhou, Pang and Tan in [6]. This scheme tracks for each block of the file system whether it is free or it has been allocated. Each hidden file has a header placed in a pseudo-random free location derived from its name and a secret key. This header suffices to locate all the blocks of the file, as it contains links to their locations.

All of the previous systems are intended to run on a single machine on top of a standard file system. There are, in addition, two approaches for distributed steganographic file systems, Mnemosyne and Mojitos. The former has been proposed by Hand and Roscoe in [4]. In order to hide a file, they write it to a location chosen pseudo-randomly by hashing the file name, the block number and a secret key. The latter was proposed by Giefer and Letchner in [2], and it combines ideas from Mnemosyne and StegFS (by Kuhn and McDonald [5]).

3 Hiding Data Accesses in StegFS

The steganographic file systems presented in the previous section are secure towards an attacker who is able to get snapshots of the state of the file system (sufficiently spaced in time so it cannot be considered to be continuous surveillance). They are vulnerable, however, towards attackers who can continuously monitor the system.

Continuous attackers are able to continuously scan the contents of the file system and detect block updates. They can also observe the I/O operations on the storage, and perform traffic analysis on the accessed (read and written) block locations. Zhou, Pang and Tan proposed in [7] mechanisms to hide the data accesses in their StegFS [6] against this attack model. In the system model (Fig. 1) of [7], users send the file requests to a trusted agent over a secure channel. The agent translates these requests into I/O operations on the storage, and returns the results to the user. Whenever there is no user activity, the agent performs dummy I/O operations. The terminology used to distinguish the block types and update operations (also called "accesses") is:

- **Data blocks** are all the storage blocks that contain the user's data. We refer as **file blocks** to the data blocks of a particular file that is being updated by the user.
- **Dummy blocks** are empty (free) blocks that contain random data.
- When the user requests a **file update**, this triggers **data updates** on all the blocks that belong to the file.
- The system performs **dummy updates** (i.e., change the appearance of the block without changing its actual content) on both dummy and data blocks, in order to hide the data updates.

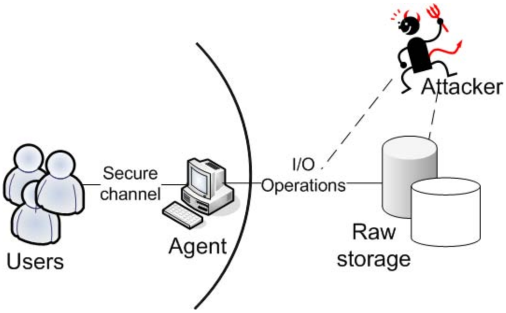


Fig. 1. System model in [7]

The authors of [7] give the following definition of security for hiding data accesses in a steganographic file system:

> "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."

The mechanism proposed in [7] to hide data updates works as follows. Whenever there is no user activity, the user agent issues dummy updates on blocks selected uniformly at random in the storage volume. For a dummy update, the agent reads the selected block, decrypts it, changes the initialization vector (IV) which serves as initial value for the CBC mode, encrypts the block and writes it back in the same location of the file system.

When the user agent receives a request to update a file, it updates all its blocks. The update algorithm relocates file blocks each time they are updated, so that subsequent operations on the file do not result in accesses to the same storage locations. In order to update file block B1, the agent first chooses a block B2 uniformly at random from the storage space. If the selected block B2 is the same as the requested one (B1), the data update is performed in the same way as a dummy update. If B2 is a dummy block, the agent swaps it with the file block B1 and updates their contents. Finally, if B2 is a data block, the agent performs a dummy update on it and re-starts the selection process. Alg. 1 shows the update algorithm provided in [7].

Algorithm 1. Update algorithm of [7]

```
if there is a request to update block B1 then
   Re: randomly pick a block B2 from the storage space
   if B2=B1 then
      read in B1, decrypt it
      update B1 IV's 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 IV's and data field
      encrypt B2, write it back
   else
      read in B2,
      decrypt it, update B2's IV
      encrypt B2, write it back
      go to 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;
```

The authors of [7] claim that the update algorithm is perfectly secure on the basis of the following 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 previous definition of security, 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."

In this paper we show that, although all blocks have the same probability of being selected to hold the updated data, I/Os produced by file updates follow different patterns than dummy updates. Thus, the probability distributions of updates in the raw storage are different depending on whether there is user activity or not. Consequently, file updates can be distinguished by an adversary performing traffic analysis on the system.

For a data update, we note that there are two possible interpretations of the algorithm:

1. If we only look at the pseudo-code, when B2 is a dummy block, the updated content of B1 is stored in B2, but the original block location, B1, keeps its old value.
2. If we take in account the text of the paper, when B2 is a dummy block, B1 and B2 are *swapped*; i.e., the agent reads B1 and overwrites it with random data. Then, it reads B2 and overwrites it with the updated content of B1.

Implementing the update algorithm as in the first interpretation implies that the updated content of the file block B1 is transferred to B2, while B1 remains intact. This approach has an obvious problem: the information contained in B1 remains there until it is overwritten (when the block is chosen as B2 in a future data update), meaning that deleted file contents and old versions could still be recovered from the storage.

Moreover, from a traffic analysis perspective, dummy and data updates produce easily distinguishable access patterns: in dummy updates, the same block location is read and written; while in data updates, the read and written block locations are different. We present an example in Fig. 2, where a file located in blocks 1, 2 and 3 is updated and transferred to positions 34, 345 and 127. We can easily see where the file blocks were and where they have been transferred, eliminating the user's plausible deniability on the existence and location of the "hidden" file. Note that a series of data updates made very close to each other indicate that, with very high probability those blocks are part of the same file. Therefore, the dummy updates performed in between file block updates (blocks 479, 290 and 47 in the example) must have been on data blocks. This reveals the existence and location of additional data blocks, besides the ones that have been updated by the user.

In the next section, we show that the security proof given by Zhou, Pang and Tan is wrong even for what seems to be a more clever implementation. Assuming that users do not update their files simultaneously, a continuous attacker can distinguish between data and dummy updates and learn, as a result, the existence and location of hidden files.

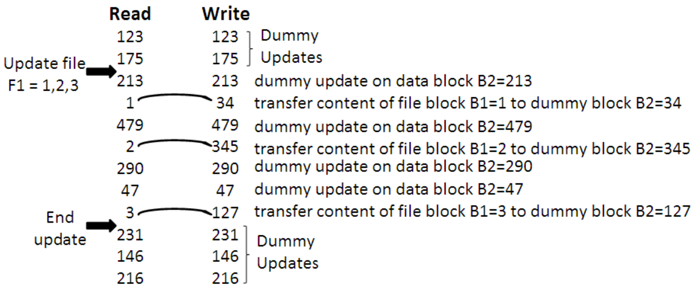

Read
Write
123
123
Dummy
Updates
Update file
F1 = 1,2,3
175
175
213
213
dummy update on data block B2=213
1
34
transfer content of file block B1=1 to dummy block B2=34
479
479
dummy update on data block B2=479
2
345
transfer content of file block B1=2 to dummy block B2=345
290
290
dummy update on data block B2=290
47
47
dummy update on data block B2=47
End
update
3
127
transfer content of file block B1=3 to dummy block B2=127
231
231
146
146
216
216
Dummy
Updates

Fig. 2. Update of a three-block file (locations 1,2,3) according to the first interpretation of the update algorithm of [7]

4 Traffic Analysis Attacks on StegFS

We recall the notation presented in the pseudo-code of Algorithm 3 of Sect. 3:

- B1 is the file block to be updated.
- B2 is the candidate block (selected uniformly at random) to hold the updated information of B1. B2 may be a data block or a dummy block. If B2 is a data block, a dummy update is performed on it, and a new B2 is selected. This process is repeated until B2 is a dummy block. Then, B1 is overwritten with random data and the updated content of B1 is stored in B2.
- B3 is the block selected uniformly at random for a dummy update while there are no user requests.

We have developed two attack strategies. The first one applies to multi-block files, and is based on exploiting file block correlations, as explained in Sect. 4.1. The second strategy, explained in Sect. 4.2, applies to one-block files, and is based on the assumption that a file block is updated with higher frequency than a dummy block.

4.1 Attack on Multi-block Files

Identifying file update patterns. Each of the data updates follows a pattern with: first, as many dummy updates on data blocks as data data blocks B2 are chosen in the updating algorithm; second, an update on the file block (B1); and finally, an update on the dummy block B2 to which the data is transferred. In Fig. 3 we can see the same example as in the previous section, where a file located in blocks 1, 2 and 3 is updated and moved to blocks 34, 345 and 127.

The updates on blocks belonging to the same file are separated by a number of dummy updates on data blocks B2. As there are more empty blocks in the storage, it is easier to randomly pick a free block, and therefore file blocks B1 are accessed closer in time to each other (together with their updated locations B2).

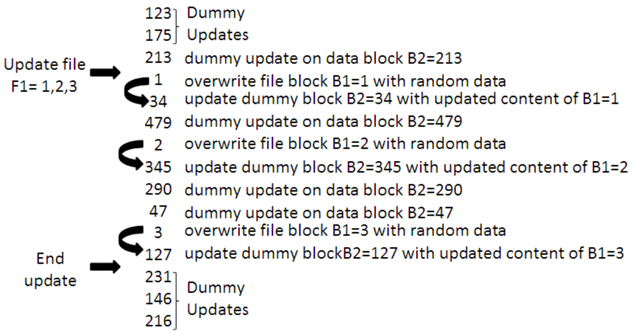

123
175
Dummy
Updates
Update file
F1= 1,2,3
213 dummy update on data block B2=213
1 overwrite file block B1=1 with random data
34 update dummy block B2=34 with updated content of B1=1
479 dummy update on data block B2=479
2 overwrite file block B1=2 with random data
345 update dummy block B2=345 with updated content of B1=2
290 dummy update on data block B2=290
47 dummy update on data block B2=47
3 overwrite file block B1=3 with random data
End
update
127 update dummy blockB2=127 with updated content of B1=3
231
146
216
Dummy
Updates

Fig. 3. Update of a three-block file (locations 1,2,3) according to the second interpretation of the update algorithm of [7]

Let R be the occupation rate of the file system. The maximum number of block updates (including data and dummy updates) a file update is expected to need is given by a negative binomial distribution, where the size b of the file is the number of successes and $1 - R$ the success probability. With probability greater than $1 - \epsilon$, a file of b blocks will be updated in at most $A = b + k$ blocks, where k can be derived from the probability mass function of the negative binomial: $\binom{k+b-1}{k} \cdot R^k \cdot (1 - R)^r < \epsilon$. For example, for an occupation rate $R = 0.5$, a complete update on a file of $b = 4$ blocks will be contained in a maximum of $A = 29$ updates with probability greater than $1 - \epsilon = 1 - 10^{-5}$.

As shown in Fig. 4, we can find similar patterns each time the file is updated (i.e., a set of locations updated closely together at two different points in time). By identifying these patterns, we can tell when a file has been updated and where exactly the file blocks are in the storage space. Once we find the most recent pattern, we know that the file will be in the location updated just after the repeated locations that have created the pattern. In the example shown in Fig. 4, the file will be in the positions 12, 60 and 125 of the storage space.

Probability of false positives. We must not forget that these patterns could have been produced by dummy updates (the attacker would think he has found a file update, but actually the access pattern has been randomly generated). We call the probability of this event as *probability of false positive*, and denote it P_{f+}. We now explain how P_{f+} can be computed.

Lemma 1. *Let B be the number of blocks in the storage, let b be the file size and A the expected length of the windows we analyze (i.e., we expect a file of b blocks to be updated in A updates). Let P_A be the probability that in A random accesses all the b blocks of the file are accessed. Then*

$$P_A \leq \left[\frac{e}{\frac{B}{A}}\right]^{\frac{B-A}{A}\cdot\frac{bA}{B}} \approx \left(\frac{eA}{B}\right)^b \quad .$$

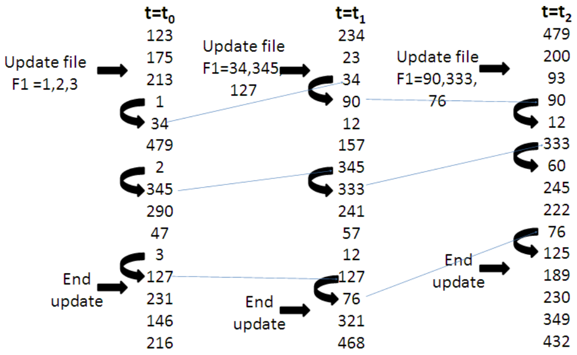


Fig. 4. Three updates on a three-block file according to the second interpretation of the algorithm of [7]

Proof. Let I_i $(i = 1, \ldots, b)$ be an indicator variable whether block i of the file has been accessed. It is easy to see that $\Pr[I_i] = 1 - (1 - \frac{1}{B})^A \leq \frac{A}{B}$. We note that the set of events $\{I_i\}$ is not independent, but as these events are negatively correlated and as we are interested in an upper bound, by assuming that they are independent we obtain a bound which is not tight.

Denote $X = \sum_{i=1}^{b} I_i$, then there all the blocks of the file are accessed if $X = b$. If we treat the events as independent, we can apply the Chernoff inequality which states that $\Pr[X > (\delta + 1)\mu] \leq (\frac{e}{\delta+1})^{\delta\mu}$, where $\mu = E[X]$. As $E[X] = \sum_{i=1}^{b} E[I_i] = \sum_{i=1}^{b} \Pr[I_i] \leq \frac{bA}{B}$, we obtain that

$$P_A = \Pr[X = b] = Pr[X > \underbrace{(\delta + 1)\frac{bA}{B}}_{b}] \leq \left[\frac{e}{\frac{B}{A}}\right]^{\frac{B-A}{A} \cdot \frac{bA}{B}} \approx \left(\frac{eA}{B}\right)^{b} .$$

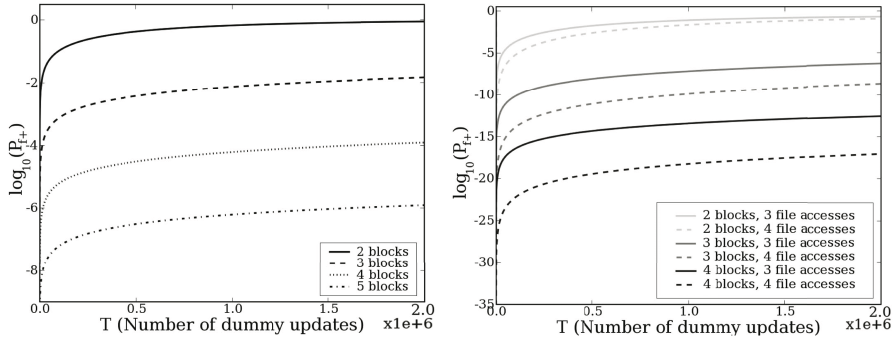
Lemma 2. *Let T be the number of dummy accesses, and let $C = T/A$ be the number of subsets of A consecutive accesses each. Under the assumption that these subsets are independent of each other.*[1] *The probability P_{f+} of having a false positive in at least one of the C subsets is: $P_{f+} \leq 1 - (1 - P_A)^C$.*

Proof. The probability that a false positive happens for one of the subsets is P_A. Thus, the probability that a false positive have not occurred for a given A accesses is $1 - P_A$. All C subsets do not produce any false positive with probability $(1 - P_A)^C$, and the result follows from that immediately.

The probability P_{f+} of having a false positive decreases both with the size b of the file and the number B of blocks in the storage. P_{f+} increases with the number T of dummy updates taken into account, and the occupation rate R.

[1] This is an approximation supported by our experimental results. The adaptive algorithm used in our implementation (see Appendix A) deals with accesses straddling two subsets of length A.

We denote $P_{f+}(i)$ as the probability of false positive, given $i+1$. Figure 5(a) shows the probability, $P_{f+}(1)$, of false positive (logarithmic scale) for two file updates. We can see that, as the file size increases (from five blocks on), the probability of false positive becomes negligible even when there are only two updates on the file. $P_{f+}(i)$ also decreases with i (see Fig. 5(b)), as it becomes less likely that more repetitions of patterns happen just by coincidence. Note that we have considered in the figures up to two million accesses, meaning that on average each block in the storage is read 200 times. We can see that even with these large numbers of accesses, real operations are detected with low false positive rate. This is because our detection algorithm takes into account the correlations between accessed locations, and not the amount of times they have been selected.



(a) Probability of false positive with two updates per file (logarithmic scale)

(b) Probability of false positive with more than two updates per file (logarithmic scale)

Fig. 5. Probability of false positive for multi-block files ($B = 10000$; $R = 0.5$; $\epsilon = 5 \cdot 10^{-4}$)

4.2 Attack on One-Block Files

Now, we assume that the user updates one-block files with a higher frequency than dummy updates occur on random blocks. Instead of searching for a file update pattern, we analyze the distance between two accesses to the same block (note that each time the file is updated it is transferred to a different location). In the example of Fig. 6 we show three updates on file F1. The distance between the first and second updates (file in block 479) is three, and the distance between the second and third updates (file in block 231) is seven.

Let B be the total number of blocks in the file system. When there are only dummy updates, the probability $P_D(i)$ of having a distance i in between two updates of the same block, follows a geometric distribution:

$$P_D(i) = \left(1 - \frac{1}{B}\right)^{i-1} \cdot \left(\frac{1}{B}\right) \quad . \tag{1}$$

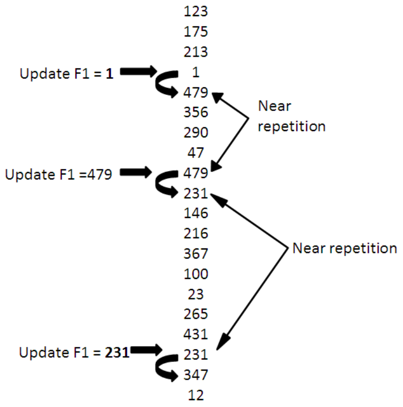


Fig. 6. Example of 3 updates on a one-block file

This equation provides a lower bound on the distance between dummy updates on the same block. The distance between random block updates increases as the user requests data updates more frequently, because the user requests particular data blocks, therefore reducing the frequency with which random blocks are selected.

Let f $(0 < f < 1)$ be the access frequency to a one-block file. The probability $P_F(i)$ of having a distance i between two updates of the file is given by:

$$P_F(i) = (1-f)^{i-1} \cdot f \quad . \tag{2}$$

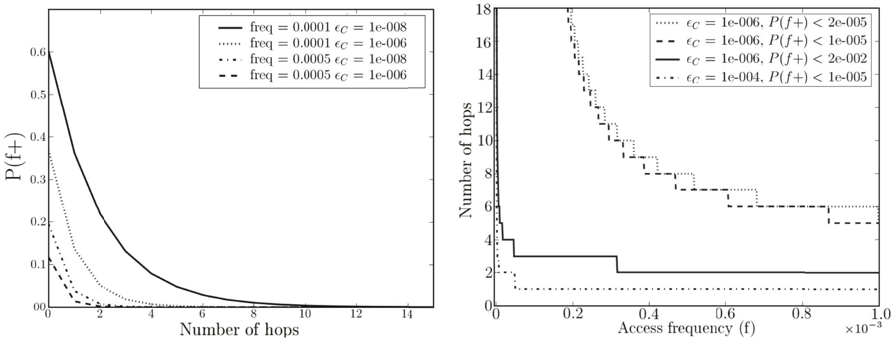
As long as the file update frequency f is significantly higher than $1/B$, distance analysis can be used to distinguish user updates on one-block files from dummy updates on random blocks.

Although the distance between accesses is, on average, much smaller for the file blocks than for the dummy blocks, it is also possible that two dummy updates on a block happen to be close to each other. If we look just for two accesses (as we do in the multi-block case), we have a high probability of false positive. Therefore, we need to find more than one *near* access in order to statistically prove that a block really contains a file. *Near* means that two data updates could be separated this distance with non-negligible probability.

Let D_C, such that $P_F(D_C) < \epsilon_C$ (see (2)), denote the maximum distance we consider *near*. The probability of false positive (i.e., consider random updates as produced by user requests) for one near access is given by $P_{f+}(1) = \sum_{i=0}^{D_C} P_D(i)$ (see (1)). As the number h of hops (consecutive near accesses) increases, the probability of a false positive decreases: $P_{f+}(h) = P_{f+}(1)^h$. On the other hand, the probability of a false negative (i.e., considering that a file update has been a dummy update) in one near access happens when a file update occurs further than expected: $P_{f-}(1) = \sum_{i=D_C}^{\infty} P_F(i)$. If we consider h near accesses as bound to consider a chain caused by user actions, we will miss a file if any of the h hops happens further than expected: $P_{f-}(h) = \sum_{i=1}^{h} \binom{h}{i} P_{f-}(1)$. We show in

Fig. 7(a) the probability of having a false positive with respect to the number of hops, for different access frequencies and values of ϵ_C.

In Fig. 7(b), we show the number of hops needed to ensure that a series of near updates belongs to a file, as a function of the access frequency. We note that a small number of hops is sufficient, even for relatively low access frequencies (in the figure we show a maximum frequency of 10^{-3}).



(a) Probability of false positives vs number of hops

(b) Required number of hops depending on access frequency

Fig. 7. Analysis of the number of hops ($B = 100\,000$)

5 Results

In order to test the effectiveness of the attacks, we have implemented a simulation of the update algorithms and the attacks in Python.[2] In this section, we summarize the empirical results.

5.1 Implementation of the Attack on Multi-block Files

We have tested the attack using favorable parameters for the user, with few files, a very low update frequency and only two updates per file. The parameters of the simulation can be seen in Table 1. As the attack becomes more efficient for larger files, we present results for two sets of files. The first set of files have sizes between 2 and 3 blocks, and the second considers files of sizes between 4 and 10.

Even though we are in a bad case as attackers, the attack has a high success probability. We summarize the results in Table 2. In both tests we found more than 99% of the files hidden in the system, although the guessed file size differs from the real one in some cases (2% for files of 2-3 blocks, and 1% for files of 4 blocks or bigger). Found files may appear slightly larger than they are because sometimes we assign “extra blocks” to the file (when there is a dummy update that fits the pattern next to the end or the beginning of the real file blocks). With

[2] The code is available upon request to the authors.

Table 1. Parameters of attack on multi-block files

Size of files (blocks)	Number of files per size	File update frequency	Size of storage space
2-3	10	3%	10000
4-10	10	3%	10000

probability ϵ, the separation between file block updates is larger than expected. This results in either some block of the file being lost (when the block with extra separation is the one accessed at the beginning or the end of the file update); or in a file being found split and considered as two smaller files (when the extra separation happens in the middle of a file). While searching for 2 and 3 block files, we find some false positives, as this occurs with non-negligible probability for very small file sizes (see Fig. 5(a)).

Table 2. Results of the attack on multi-block files

Size of files	Files found	Wrong size	False positives
2-3	$> 99\%$	$< 2\%$	$< 2\%$
4-10	$> 99\%$	$< 1\%$	0%

5.2 Implementation of the Attack on One-Block Files

We have considered a file system with $B = 100\,000$ blocks, where 10 one-block files are accessed 12 times each. We have tested the efficiency of the attack for several access frequencies. In the implementation of the attack, we consider we have found a file when we find a chain of 10 or more near accesses. We have set the probability of false negatives by fixing $\epsilon_C = 10^{-12}$.

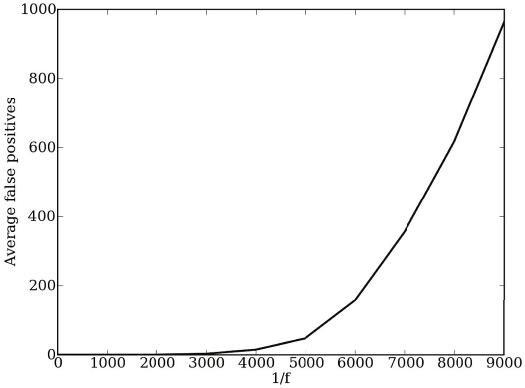


Fig. 8. False positives vs. $1/f$ ($B = 100\,000$; $\epsilon_C = 10^{-12}$)

As expected, the performance of the attack heavily depends on the access frequency f to one-block files. We show this dependence in Fig. 8, where we represent the number of false positives as a function of the access frequency. For frequencies greater than $4 \cdot 10^{-4}$ (in Fig. 8 this corresponds to $1/f = 2500$), we can find all one-block files without obtaining any false positives. As the access frequency decreases, the performance of the attack degrades, and more false positives appear. We note that the number of false positives depends on the access frequency, but not on the number of files in the system.

6 Conclusions and Future Work

In this paper we have analyzed the update algorithm proposed by Zhou, Pang and Tan in [7], and found that it is vulnerable to traffic analysis attacks. We have described the theory behind the attacks, explained how the system's parameters influence their performance, and presented empirical results on their effectiveness.

Our results show that the security claims in [7] are unsubstantiated. Their algorithms do not produce the same patterns for file and dummy updates, therefore, the probability distribution of updated locations in the storage is different whether there is user activity in the system or not. Our attacks exploit these changes and successfully distinguish file and dummy updates, finding the locations of hidden files. Multi-block files are very easy to find (two updates are sufficient to reveal their existence and location), while several file updates are needed in order to find one-block files. Our (non-optimized) implementation successfully finds most of the files hidden in the storage, and more efficient implementations could further increase the accuracy of the attacks.

The two key weaknesses in the update algorithm proposed in [7] are:

- Blocks are rarely relocated, and when they are, their new location appears next to the old one in the history of accessed locations. This greatly reduces the uncertainty on the possible locations to which block contents may have been moved.
- While the "dummy updates" select block locations uniformly at random, multi-block file updates generate correlations between accessed locations that could not have been plausibly generated at random.

The traffic analysis strategies presented in this paper show that introducing "a bit of randomness" is not sufficient to effectively conceal user accesses to files in a steganographic file system. More sophisticated mechanisms are required in order to design a traffic analysis resistant steganographic file system; developing such mechanisms is left as an open problem.

The authors of [7] also propose a method to conceal read accesses to files, based on a multi-level oblivious RAM [3]. A line of future research could analyze whether or not this mechanism resists traffic analysis attacks.

Acknowledgments

We would like to thank the anonymous “Reviewer 1”, whose comments have substantially improved the quality of this paper. This work was supported by the IWT SBO ADAPID project (Advanced Applications for e-ID cards in Flanders), GOA Ambiorics and IUAP p6/26 BCRYPT. Carmela Troncoso is funded by a grant of the Foundation Barrie De la Maza from Spain.

References

1. Anderson, R.J., Needham, R.M., Shamir, A.: The steganographic file system. In: Aucsmith, D. (ed.) IH 1998. LNCS, vol. 1525, pp. 73–82. Springer, Heidelberg (1998)
2. Giefer, C., Letchner, J.: Mojitos: A distributed steganographic file system. Technical report, University of Washington (2004)
3. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM 43(3), 431–473 (1996)
4. Hand, S., Roscoe, T.: Mnemosyne: Peer-to-peer steganographic storage. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 130–140. Springer, Heidelberg (2002)
5. McDonald, A.D., Kuhn, M.G.: StegFS: A steganographic file system for linux. In: Pfitzmann, A. (ed.) IH 1999. LNCS, vol. 1768, pp. 462–477. Springer, Heidelberg (2000)
6. Pang, H., Tan, K.-L., Zhou, X.: StegFS: A steganographic file system. In: Proceedings of the 19th International Conference on Data Engineering, pp. 657–667. IEEE Computer Society Press, Los Alamitos (2003)
7. Zhou, X., Pang, H., Tan, K.-L.: Hiding data accesses in steganographic file system. In: Proceedings of the 20th International Conference on Data Engineering, pp. 572–583. IEEE Computer Society Press, Los Alamitos (2004)

A Algorithms for Detecting Patterns

We show first the algorithm used to uncover multi-block files, and afterwards the one to reveal one-block files. We note that these algorithms are not optimized. An attacker could run them several times, and use the information gained in the first runs to refine the results in later ones.

A.1 Multi-block Files

Worst Case Scenario. We checked the effectiveness of the algorithm in a worst case scenario. We made two simulations with a file system of $B = 10000$ blocks, with files with sizes of $2 - 3$ blocks and $4 - 10$ blocks, respectively, and only 10 files of each size. The file access frequency has been set to 3%, so that accesses to files are on average far away from each other. Finally, we considered only 2 updates per file.

The algorithm (Alg. 2). The algorithm used to find multi-block files works as follows. For b-block files we first calculate the expected number A of blocks its update is expected to occupy. Denote by $G_F(A)$ the most recent chunk of A consecutive accesses. We compare $G_F(A)$ (as shown in Fig. 9(a)) with the previous chunk of A accesses $G_M(A)$. If there is more than one element in the intersection (i.e., locations that have been accessed in both chunks), we increase the size of the chunks and compare them again, in order to detect file updates that do not fit exactly inside the chunks. Finally, if the intersection contains b (or more) elements, we conclude these are due to a file update. If not, we take a new $G_M(A)$ and compare it again with the $G_F(A)$. Once $G_F(A)$ has been compared with all prior $G_M(A)$'s, we choose a new $G_F(A)$ and restart the process.

Fig 9(a) shows an example of this algorithm, where a 3-block file (1, 2 and 3) is detected in positions 34, 345 and 127. In Step 1, we define a first chunk $G_F(A)$, and a first chunk $G_M(A)$, we can easily see that the intersection of the fixed chunk with the rest of the accesses is void. Then, in Step 2, new $G_F(A)$ and $G_M(A)$ are chosen. We can see that, after a couple of comparisons (Step 3), $G_F(A) \bigcap G_M(A) > 1$. Then, in Step 4, we increase their sizes to check if the intersection grows. From this result we can derive that a 3-block file is located in positions 34, 345 and 127.

A.2 One-Block Files

Worst case scenario. For one-block files, the simulation was made with a file system of $B = 100000$ blocks, only 10 one-block files accessed 12 times each, and frequencies higher than 10^{-4}. We consider we have found a file when at least $h = 10$ near accesses are chained.

The algorithm (Alg. 4). In order to find one-block files, we start with the most recent access and search near repetitions of it, if there are not, we move to the next position and so on. Once we find a first near access, we build a tree of near accesses (given that there can be more than one near access to a position) and, if one of its branches has more than h elements, we conclude that we have found a one-block file.

We illustrate in Fig 9(b) how a one-block file is found. Assuming that 123 has been the latest access to the storage and that we are looking for chains of $h = 5$ hops, the first candidate we find is 479, which has two possible *newCandidates*, 231 and 431. Following the algorithm, we arrive at the tree showed in the figure, and can conclude that there is a file in location 1, that comes from position 222 after having passed by locations 278, 347, 231 and 479.

Algorithm 2. Algorithm to search multi-block files patterns

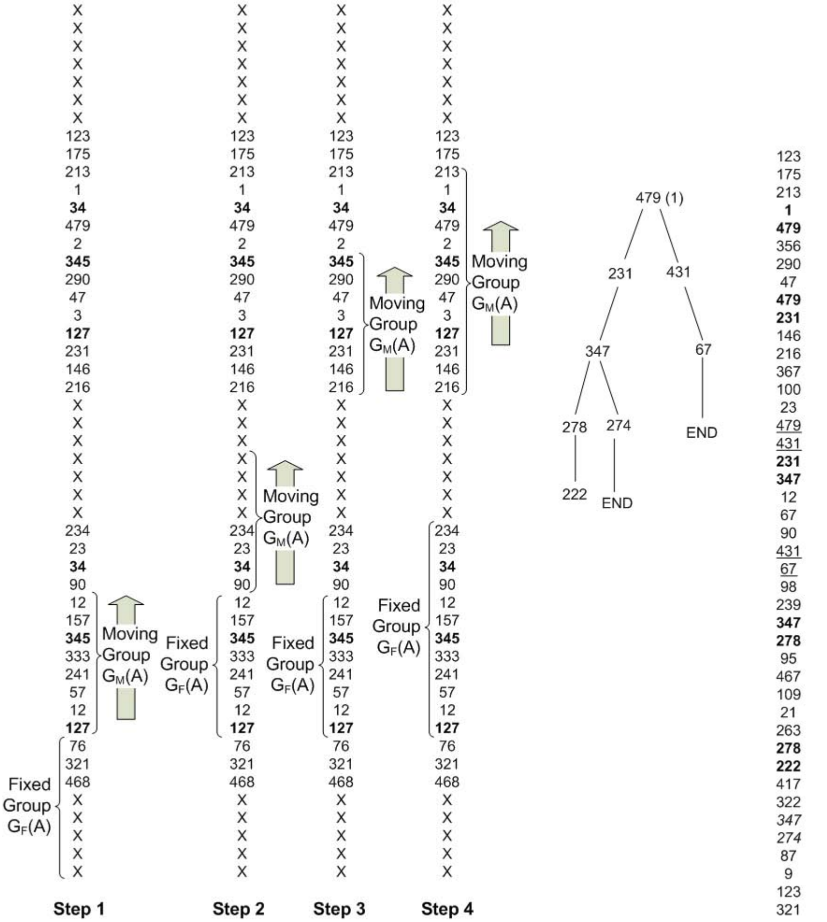
```
$FixPointer$ = last access to the system
while there are accesses to make a new fixed chunk do
    we choose $G_F(A)$, a chunk of $A$ accesses from $FixPointer$ to $FixPointer - A$
    $MovPointer = FixPointer - A + 1$
    while there are accesses to compare with $G_F(A)$ do
        we choose $G_M(A)$, a chunk of $A$ accesses from $MovPointer$ to $MovPointer - A$
        $MovPointer = MovPointer - A - 1$
        if $Int = G_F(A) \bigcap G_M(A) > 1$ then
            repeat
                $IntOld = Int$
                we increase the size of the chunk in x blocks $A = A + x$
                $Int = G_F(A) \bigcap G_M(A)$
            until $size(IntOld) = size(Int)$
        if $size(Int) >= b$ then
            there is a file in the locations belonging to the intersection
        else
            false alarm, continue searching
        $MovPointer += A$
    $FixPointer += A$
```

Algorithm 3. searchCandidates(*Location*, *Tree*) (near accesses to a given location)

```
$candidates$ = list of "near" repeated accesses to $Location$
if $size(candidates) >= 1$ then
    for $candidate$ in $candidates$ do
        $newCandidate$ = location accessed immediately before $candidate$
        append $newCandidate$ to $Tree$
        $Tree = searchCandidates(newCandidate, Tree)$
RETURN: Tree
```

Algorithm 4. Search one-block pattern Algorithm

```
$Location$ = last access to the system
$tree = NULL$
while there are accesses in the list do
    $treeCandidates = searchCandidates(Location, tree)$
    if there is a branch with more than $h$ elements then
        the block right before the root of the tree has a file
    else
        $tree = NULL$
```



(a) Finding a multi-block file pattern

(b) Finding a one-block file pattern

Fig. 9. Algorithms used to detect patterns in the accessed locations