PicFS: The Privacy-enhancing Image-based Collaborative File System

Chris Sosa Blake C. Sutton
Department of Computer Science
University of Virginia
{sosa, bcs8d}@cs.virginia.edu

H. Howie Huang
Department of Electrical and Computer Engineering
George Washington University
howie@gwu.edu

***Abstract*—Cloud computing makes available a vast amount of computation and storage resources in the *pay-as-you-go* manner. However, the users of cloud storage have to trust the providers to ensure the data privacy and confidentiality. In this paper, we present the Privacy-enhancing Image-based Collaborative File System (PicFS), a network file system that steganographically encodes itself into images and provides anonymous uploads and downloads from a media sharing website. PicFS provides plausible deniability by preventing traffic and image analysis by any third party from revealing the existence of PicFS or compromising its data. Because all accesses are anonymized, users of PicFS are dissociated from their data, which protects users against being compelled to release their keys. For further security and ease of use, we develop a method for automatically generating a large set of non-suspicious images to serve as input to the system. Our prototype leverages a number of existing technologies, including the F5 algorithm for steganography, Quick-Flickr for Flickr API access, Tor for anonymization, and FUSE-J for user-level filesystem calls. We show that the PicFS is indeed practical as the prototype demonstrates satisfactory performance in the real-world environment.**

I. Introduction

Today online file storage (e.g., Google Docs and Flickr) allows easy backup, sharing, and collaboration on photos, documents, spreadsheets, presentations, etc. However, the users of online services for storing files currently have an all-or-nothing security choice: they can either compromise all of their data and its structure by using the service, or compromise nothing by opting out completely. Although encryption can protect the contents of data in this situation, it cannot hide the existence of hidden content - an equally important issue. For example, a host uninterested in user privacy could ban encrypted files above a certain size, or altogether. A host could choose to reveal encrypted data to a third party with the resources to break the encryption. Finally, an attacker with unauthorized access to the host could access the data. For these reasons, users of online storage need self-enforcing privacy - security for their files that protects not only the content and structure of data, but also its very existence.

In this paper, we present the Privacy-enhancing Image-based Collaborative File System (PicFS), a network filesystem that allows writes only by its owner and allows sharing to other designated individuals. PicFS achieves strong online privacy through hiding the existence of hidden data, that is, *plausible deniability* introduced in [1]. In PicFS, both inodes and data blocks are steganographically encoded into images and anonymously uploaded and downloaded from a media-sharing host. Because all accesses are anonymized, users of PicFS are dissociated from their data, which protects users against being compelled to release their keys. This is of special importance to allow users to share files and information without fear of censorship and retribution.

We develop these ideas by bringing plausible deniability to every step of an online steganographic filesystem. Specifically, PicFS targets four key issues. First, in order to mimic ”normal” access patterns to Flickr, we study a list of popular Flickr applications for comparison with PicFS's access behavior, as our system can use an open API to communicate with Flickr [2]. Second, in addition to normalizing access patterns to Flickr, we further enhance PicFS with anonymized accesses. We leverage Tor's onion routing technology to provide an additional layer of security for the user without sacrificing plausible deniability. Third, we address the logistics of finding a large number of believable images to use as input to the filesystem. This issue is crucial for plausible deniability and secure steganography - the system must not use a set of images that is very unlikely to appear on a Flickr account, and it cannot use images available elsewhere without the risk of image comparison revealing the use of steganography. Since the filesystem is also log-based, even small filesystems require a large pool of suitable images, as a new image is needed for every change. PicFS attacks this problem by automatically selecting ”interesting” frames from user-provided home videos to provide a never-ending source of believable images available only to the user. Finally, because online storage tends to cause undesirably long latencies, we explore the tradeoffs of an on-disk cache to mitigate network delays and overhead from steganography.

The idea of a steganographic filesystem is not new, although few prototypes exist. For example, StegFS is a steganographic filesystem [3] which uses the free memory blocks in a local filesystem to store information, but it has persistency problems as free memory blocks can be re-allocated at any time. In addition, this filesystem does not have the means to be shared covertly. Our system PicFS is most closely related to CovertFS, a Web-Based Covert File System [1]. However, there is yet to be a prototype or evaluation of CovertFS and so it is unclear whether such a system can achieve the original goal of strong privacy. In PicFS, we design and

implement a number of key components that are not only missing from CovertFS but crucial to the success of PicFS, e.g., automated image generation, anonymized access patterns, and filesystem optimizations. In this paper, we show that the PicFS is indeed practical as the prototype demonstrates satisfactory performance results in the real-world environment.

The rest of this paper is organized as follows. Section 2 explains the design of PicFS and describes algorithms used with video sequences in order to realize PicFS. Section 3 goes over the implementation of PicFS and the image generation. Section 4 presents an evaluation of the implementations of PicFS and the image generation. Section 5 presents the related work in the design of PicFS. Section 6 summarizes the work and discusses future research directions.

II. Design

The design of PicFS can be broken into two distinct parts: image generation and filesystem design.

A. Image Generation From Video

It is important for plausible deniability that the input images to PicFS be unique and not arouse suspicion. One approach to this problem is to ask the user to provide a group of pictures - however, the log-based filesystem requires a very large image set. To solve this problem, PicFS uses user-provided home videos to generate the large number of images it needs for steganography. Since many modern digital cameras have video capture features, original footage is a perfect and never-ending source of plausible, unique images. Home videos of all quality levels and subjects are very common. In addition, each sequence of video is unique to the user and cannot be found elsewhere for comparison against the encoded version.

Since we want to extract a large number of plausible images from a variety of videos, we mix and match from previous video shot detection algorithms to create the most general, domain-independent method that suits our purpose. For each input video, the difference values using three different metrics are computed between consecutive frames. We use the L1 distance, the Chi-squared difference in the intensity histograms, and the number of edges detected with the Sobel edge filter [4] as our metrics. Each measure is sensitive to some type of noise present in many videos, so the other metrics are chosen to compensate for this effect. For example, the L1 distance is very sensitive to motion while the histogram difference is not, and the histogram difference is sensitive to lighting changes, while the L1 distance and the edge detector are more robust to this type of noise.

To process a home video, we first compute the differences between consecutive frames for each metric. Once these difference vectors are computed, we average the results and select interesting frames from the resulting distribution using an adaptive threshold. Based on the difference statistics in a window of surrounding frames, the threshold varies across each video sequence and each video, requiring no manual parameter setting. The size of the window affects the number of frames selected from the video.

B. The Filesystem

Our design is based on the Second Extended Filesystem (Ext2) [5]. We chose Ext2 over Ext3 and Ext4 since many other filesystems including Ext3 are based on the structure of Ext2. Since we are mostly concerned with the structure and not additional services such as the Log introduced in Ext3, Ext2 is a good choice.

We define the following Attacker-Centric Threat Model. The attacker we are concerned with is the one who has access to all images on the media-sharing service and also can snoop on the network and detect what and to whom a client sends a message. We address this attacker in our evaluation section. We assume that the attacker does not have access to a client's machine.

In order to support our goal of plausible deniability and address our threat model we must provide:

1) An efficient mapping between filesystem blocks and images;
2) File access traffic that does not draw more attention than other applications using the API of a media sharing site.

The first point is intrinsic to filesystems - it must be mountable and provide a way to find files. The second point, normalizing access patterns, supports plausible deniability and addresses our threat model. The following sections show our design and how it addresses these issues.

1) The Superblock and the Allocation Table: In Ext2, the first portion of memory on disk is known as the superblock. From the superblock, a filesystem knows where to find free blocks, how to find inodes that correspond to the head of a file and how to find data blocks. However, for PicFS most of these are unnecessary. First, since PicFS draws free memory blocks from generated images, PicFS cannot track free blocks. Instead, PicFS generates images as needed from a pool. PicFS further combines the other functions of the superblock into an allocation table - a data structure that links inode numbers to paths within the mounted filesystem and their corresponding image names. The allocation table also links the data block numbers to image names. Thus, given an inode number, we can find which path it refers to on a mounted filesystem and the image that corresponds to the inode block on the media sharing service. We can also find the image names of the corresponding data blocks. Figure 1 is an example of an allocation table. Note that keeping the filesystem paths within the table improves performance as it avoids having to search a directory hierarchy to locate the image file for an inode. To scale up the design to a large number of files, we shall investigate new techniques (e.g., hash, embedded DB) as part of future work to improve the search performance.

2) Mapping Blocks to Images: To structure inodes, we use the Ext2 design, which includes meta-information such as the file size, ownership, and data block locations. All filesystem information including that of the allocation table is stored within the media sharing service. We have a one-to-one mapping between blocks in the filesystem and images stored on the media sharing service.

Allocation Table

Inode number	PICFS Path	Image Name
1	/	Pic010.jpg
2	/dir1	Pic005.jpg
3	/dir2	Pic003.jpg
4	/file1.txt	Pic006.jpg
5	/dir1/file2.txt	Pic021.jpg

Data Block Number	Data Image Name
1	Pic007.jpg
2	Pic008.jpg
3	Pic020.jpg
4	Pic124.jpg
5	Pic023.jpg
6	Pic127.jpg
7	Pic643.jpg
8	Pic693.jpg
9	Pic234.jpg
10	Pic232.jpg
11	Pic999.jpg
12	Pic245.jpg

Fig. 1. An allocation table populated with inode and data information.

Typical photo sizes on media sharing services range from 40KB to 300KB and current steganographic allow embedding of about 10% of data [1]. This allows for a minimum block size of up to 4 KB. Since this filesystem is based on Ext2, it relies on a single fixed value for the block size. Our implementation uses an even safer value of 2 KB; however, with the image generation technique described in the previous section, a user will have the ability to increase this block size. Note that with this feature, we can use at most 10% of the total storage capacity provided by the media sharing service. While we realize that this may be seen as a large waste of space, Flickr and other service providers have accounts with relatively unlimited bandwidth and storage for a small service fee. Thus, this is not a problem for the determined user.

3) Locating and Reading Files: With the allocation table, PicFS can look up a path for a desired file. This path gives PicFS both the inode number and the image name of the image encoded with the inode of that file. Once PicFS has decoded that image, it proceeds to look up the individual data blocks for the file by using the information in the inode and the allocation table. Figure 2 shows the different method that PicFS uses versus CovertFS's method. The data for this example is taken from the allocation table in Figure 1. As you can see, for even a simple lookup of a file in the root directory, PicFS requires 40% less accesses than CovertFS.

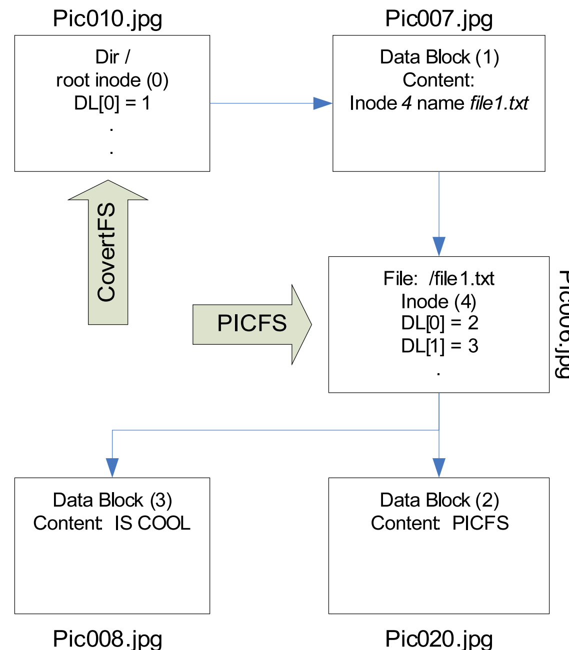


Fig. 2. How to access a file in PicFS vs. CovertFS using the allocation table from Figure 1. Both filesystems are performing a lookup on file1.txt. The arrows indicate the starting point of the lookup process for PicFS and CovertFS, respectively.

However, one question remains - how does PicFS get the allocation table? We cannot treat it as a regular file because we cannot yet reference an allocation table to do so. This is a classic chicken and egg problem. To avoid this issue, we add an additional attribute to the inode of the allocation table that references a chain of images that contains the consecutive data blocks for the allocation table. Each data block in turn also has an attribute that contains the image name of the next image for the chain.

4) Mounting the Filesystem: In order to mount PicFS, a user must have a valid account with the media sharing service. In addition, the filesystem owner must also know the image name of the allocation table's inode. Sharing users must also know this image name - it is a shared secret that can be transferred from filesystem owner to authorized shares in the same manner as a password might be. To finish mounting, a user must download the image file associated with the allocation table, the full allocation table contents, and finally the root directory of the filesystem.

5) Filesystem Writes and the On-Disk Cache: To keep file accesses from looking too suspicious, PicFS must use log-structured writes. We know of no application or user that routinely modifies images and re-uploads them. It is not intuitive for an individual to do so with the amount of storage available - even when users modify images after upload, it is usually easier to upload the modification as a new image.

Instead, when a file is modified and closed, new images are produced for the data blocks that have been changed and the inode block. The inode block requires an update because on modification at least the modification time must change. The allocation table is then modified to reflect these changes. PicFS keeps track of modifications to the allocation table by chaining the images. In other words, the original allocation table takes advantage of the a priori knowledge of possible image names and reserves another one at creation for the location of the next allocation table inode block. If this file exists on the media sharing service, then a new allocation table is available. This chain is only necessary for the allocation table. If the filesystem were to support writes from other individuals besides the owner it would be necessary to chain all files. Instead, users are reading the filesystem are guaranteed a filesystem that is as up-to-date as when they mounted it. This avoids issues of cache coherency and consistency that can be solved with a complex locking protocol. Note that PicFS only needs to find the most up-to-date version of the allocation table when the filesystem is mounted. Therefore, the expensive operation of following a chain image by image only has to be performed once.

In order to further facilitate this filesystem, we add an *Image-based On-disk Cache*. This optimization is essential in reducing traffic without giving up plausible deniability. Basically, the on-disk cache mimics the behavior of the media sharing service by handling all reads and writes for the owner until dismount. The on-disk cache can be implemented as a directory on the local filesystem which contains the steganographically encoded images produced by PicFS and our prototype takes this approach. A compromise of this directory does not affect the privacy of PicFS, because an intruder would at most have an out-of-date image directory from the user's media sharing service, but no unencoded files. Many tools on Flickr behave in this way (see Evaluation) and thus this optimization does not reduce plausible deniability. This also reduces the number of images that need to be uploaded to the media sharing service and can markedly improve performance. It is also possible to flush the cache at a specific interval to balance the performance and the freshness of online data.

6) Dismounting the Filesystem: At dismount, PicFS writes the allocation table to the on-disk cache by dividing the table into blocks and steganographically encoding the blocks into images. The changes are then flushed from the on-disk cache to the online media service. These changes can be detected by checking against the manifest of the media sharing service before uploading. As a final step, the on-disk cache is erased.

III. Implementation

A. The Filesystem

Our implementation of the PicFS filesystem is an installable filesystem for Linux. We used FUSE to allow us to build PicFS as a user-level filesystem to simplify the amount of work involved [6]. FUSE is composed of a kernel-driver that communicates with a user-level library that an installable filesystem can interact with. Specifically, we used FUSE-J, a Java wrapper around FUSE that uses the Java Native Interface (JNI) to make calls between the FUSE user-level library and PicFS [7]. Java was our language of choice because serialization of data structures into bytes is much easier in Java than in C. As a communication layer we configured Tor to anonymize our traffic. For steganography, we used F5, a popular high-capacity steganographic method that hides information in the least significant bit of image pixels [8]. However, note that the method of steganography used is modular and tied to PicFS. There is a definite tradeoff between the robustness of the steganographic method and its performance and F5 is a good compromise. In addition, F5 also allows users to encrypt data with a special password before encoding to protect the confidentiality of the data even if steganalysis decodes the image.

The design of the filesystem left a few choices. The most important of these choices was to either bulk download all the image files that are used with a given allocation table at mount time, or download images on-demand. We chose to implement both of these methods.

While the filesystem is a complete prototype that handles all commands passed by FUSE, there are a few items from the design left unimplemented. First, we have not yet implemented chaining of the allocation table. Second, we have only implemented direct blocks in inodes. This allows the maximum file size to be only twelve times the block size (which in our case is 24KB). Finally, we implemented file opens to decode all data in a file before passing back an open handle. This step is unnecessary and in terms of performance, inefficient.

B. The Media Sharing Service

We began to implement PicFS using Flickr as the media sharing service. Flickr was an ideal choice because of its strong presence on the web and its open API [2]. We used Quickr-Flickr to access the Flickr API from Java [9]. However, halfway through the integration with Flickr we found two key problems. First, it would be difficult to evaluate the anonymity and access patterns given that the filesystem was accessing Flickr through Tor. Second, Flickr's free accounts do not allow other users to download original images from the API.

Although we considered working around the system with HTTP requests to mimic a web user, we anticipated much lower performance from this approach. Instead, we implemented our own media server on the web which supports upload, download, search and delete commands. All evaluations were done on our media server to model the service provider's knowledge of incoming traffic and account information. The media server is for evaluation purposes only and we modularized PicFS so that it would be relatively easy to use a different media server in the future.

Figure 3 show our implementation of PicFS and how the individual components interact.

C. Traffic Anonymization with Tor

In a public image store, a normal access pattern is for users to upload images or share with their families, friends,

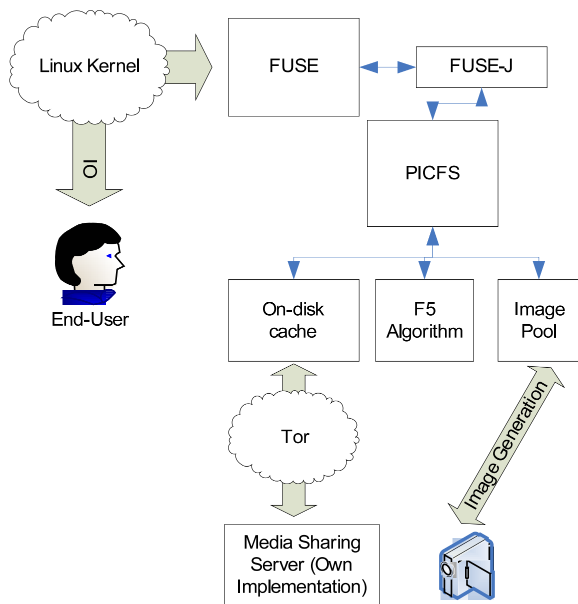


Fig. 3. Block diagram of the different subsystems of PicFS interacting. The thin arrows refer to module communication in the system while the thicker arrows refer to external communication such as network communication or through the I/O subsystem of the operating system.

or anyone with an internet connection around the world. Once uploaded, many images are mostly viewed by others, rather than the owners. However, a PicFS client frequently needs to retrieve a set of his/her own images from the web site in order to read the hidden contents. To avoid this abnormal access pattern, we utilize Tor network [10] [11], in particular its encrypted and periodically updated data channels, where the participating routers in each hop have no knowledge of data source or destination beyond the current hop. The main benefit that Tor brings to PicFS is that it helps to blend PicFS accesses into the main stream of image browsing traffic. As a result, when a PicFS client reads a file, the traffic to Flickr will no longer stands out as the owner suspiciously views his/her own pictures over and over again. On the contrary, it will look like just another user on the internet becomes interested in these images and starts to view them.

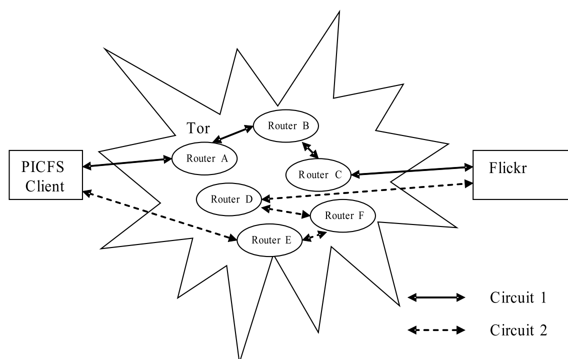


Fig. 4. Example of image retrieval with the Tor network.

Figure 4 shows how a PicFS client retrieves images via a Tor network. In this simple example, a PicFS client first creates Circuit 1 on three onion routers A, B, and C. In this case, Flickr sees that a user from router C sends requests to a number of images. It is not aware of the fact that the access is originated by the PicFS client. Nor does Router C. Also, router A knows the source of the request but not the destination, and outer B knows neither the source nor the destination. The client uses this circuit for ten minutes. Then a new circuit, Circuit 2, is created. This way, the accesses originated from one PicFS client are disguised as ”normal” photo sharing activities of two independent users.

IV. Evaluation

The evaluation of PicFS is decomposed into two. The first component of the evaluation evaluates the effectiveness of the image generation technique. The second component evaluates the filesystem itself. The filesystem is evaluated on its support for plausible deniability and practicality.

A. *Image Generation from Video*

Evaluating the effectiveness of the image selection algorithm for our purposes is somewhat difficult. Although it is relatively simple to test if actual shot changes in a video sequence are detected, there are probably images besides shot changes in the video sequence that are different enough to be included in our image set. One study of Flickr users described a new class of photo-takers, who uploaded images more than twice per week, took many ”interesting” and ”artsy” pictures, and were unconcerned with photo privacy [12]. As a result, we restricted our evaluation to a qualitative analysis of the images selected from distinct video sequences, in the context of the distribution of differences.

We used three datasets from different genres to evaluate the image selection: a short home video of a cat playing with a tortoise, part of an animation episode, and part of a popular television show. Each video sequence had different features which made some of the metrics more suitable than others. The animation sequence had frequent small movements and many shot changes, but clearly defined edges. The home video had many lighting changes, jerky camera movement, and relatively low quality frames. Finally, the television sequence had more regular shot patterns, but had dramatic lighting changes and longer periods of small movements, like talking. Interestingly, our method yielded a greater number of very different images for the home video sequence, making it ideal for the purpose of using home videos to generate images for the filesystem.

Figure 5 illustrates the difference distributions calculated from the three metrics used for our first dataset, a home video of a cat playing with a tortoise. The distribution for each metric is very different, which is expected since each metric was chosen for its sensitivity to distinct video features. In addition, the scale of differences detected varies greatly between metrics, emphasizing the importance of an adaptive thresholding approach. Figure 6 shows the frame differences for each difference metric plotted against the frame number in the sequence.

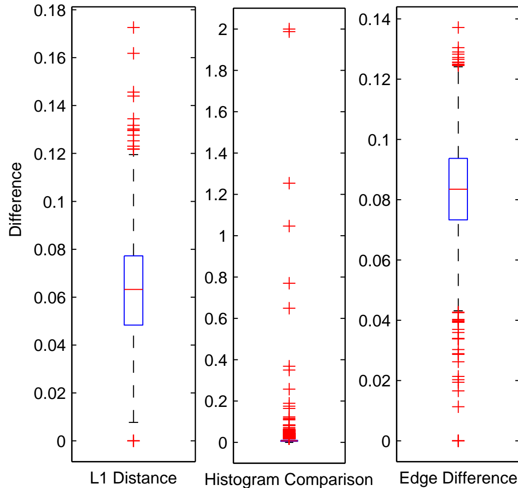


Fig. 5. Difference distributions, home video.

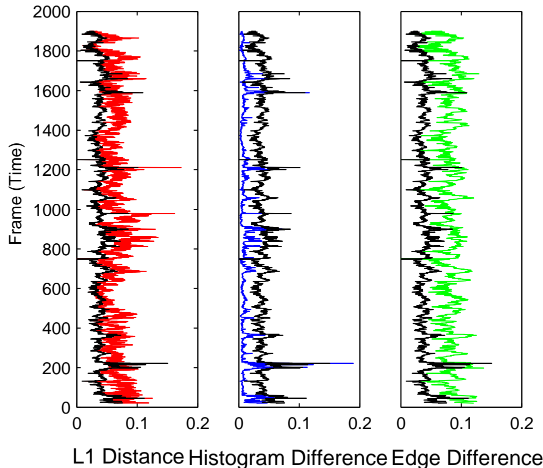


Fig. 6. Difference metrics and randomized distribution, home video.

Finally, Figure 7 shows a plot of the frames that our algorithm automatically selected from the weighted difference distribution. As expected from the adaptive thresholding approach, the frames selected all lie at the peaks of the weighted difference distribution. We have manually annotated the graph with descriptions of the action in the video at key points in the distribution. See Appendix A for the actual selected images corresponding to the figure annotations.

For brevity, we omit the plots from our other datasets. Although the distributions for the other datasets are similar, there are significantly more false positives (overly similar frames) selected in the television and animated sequence datasets. The algorithm seems particularly prone to false positives resulting from professional camera effects (subtle fades, cross dissolves, camera spinning around the scene), and long scenes with small amounts of motion, like talking sequences. We believe this is because the difference metrics do not provide a good enough

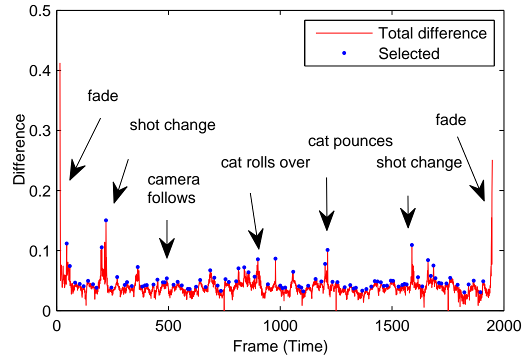


Fig. 7. Annotated plot of automatically selected frames, home video.

estimate of the similarity between frames for this type of data. The problem of computing perceived similarity between two input images remains an open problem in computer graphics. In the future, we hope to apply the latest research in video shot detection to help alleviate these issues. Fortunately, most home videos will not make use of such professional effects.

Dataset	Length	Frames	Selected	Percentage
Home Video	2:15	1950	96	4.9
Television	3:17	14932	258	5.2
Animated	15:00	14298	885	6.2

TABLE I
IMAGE GENERATION RESULTS.

Table I shows the number of images selected from each video sequence and the percentage yield. Although the percentage may seem relatively low, the focus of the approach is to select images unlikely to arouse suspicion from an outside observer. Thus, the percentage of frames selected from the video should be very low unless the video itself features choppy editing and covers many different kinds of scenes.

B. The Filesystem

We have evaluated the implementation of the PicFS such that the on-disk cache downloads all images that are on the filesystem on mount. The only other interaction between the media server and the filesystem is the bulk upload of images that are in the allocation table when dismounting and that were not downloaded on mount. This ensures that PicFS minimizes the number of photos it must upload and download.

1) Plausible Deniability: To show that the types of access of this filesystem are consistent with plausible deniability, we must show that the file access patterns are consistent with a few API tools of a media sharing service. We use Flickr as the media sharing service to evaluate against since it is a popular photo-sharing service and has many popular tools that take advantage of its API [2].

Among the various applications that use the API there are at least two applications that exhibit similar behavior to the PicFS prototype. These two are FlickrFS [13] and Gnicker [14]. These applications both exhibit bulk download on startup

(request most images on Flickr) and bulk upload on exit (synchronize on exit).

These are exactly the semantics of PicFS. When PicFS is mounted, the on-disk cache fetches many images in bulk when it reads in the allocation map and all files in the allocation map. Once the filesystem is mounted, there is no interaction between the filesystem and the media-sharing service until dismount. On dismount a large number of files are uploaded. The files that are uploaded are different from the ones downloaded. This can be seen as a synching operation. As a result, PicFS can mimic other applications and preserves plausible deniability.

2) Practicality: Our target system for the evaluation of our prototype system is on a virtualized Ubuntu operating system running on VMWare Server. The hardware of the system is an Intel Core Duo running at 2.4 GHz with 2GB of RAM. The block size for our filesystem was kept at 2KB. Finally, since indirect blocks have not yet been implemented in PicFS, the file sizes are limited to 24 KB (with twelve) direct blocks.

At this time, our filesystem decodes all the blocks of a file on open. This, unfortunately, precluded the use of many popular benchmarking software since they do not take into account the amount of time it takes to open a file. This is a good point for future work since there is a definite trade-off between decoding blocks all at once or on-demand decoding. Depending on the usage of a file, one may be much better than the other. Figure 8 shown below shows the performance taking into account both opens and closes for reads and writes.

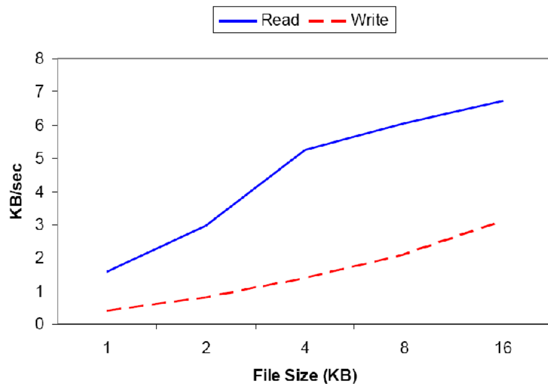


Fig. 8. Performance of reads and writes on a file of differing sizes. These reads and writes involve reading and writing to the entirety of the file.

It is important to note that this performance has been measured in KB/sec. As a reference, the host filesystem (Ext3) measured using IOzone [15], a popular benchmarking suite, measured writes on the order of 100 MB/sec and reads on the order of 1 GB/sec! Clearly, there is a large price to privacy and the practicality of the prototype for normal use filesystem is insufficient. However, as the filesystem being used with files of small sizes (order of KB or small order of MB), is sufficient. There are also remain many possibilities in optimizing the performance by using better caching and encoding/decoding mechanisms. At present time, the largest overhead is in the steganographic method.

V. Related Work

There is a rich body of work concerning steganographic techniques for media, although many methods use steganography for watermarking, rather than information hiding. A useful class of techniques is Quantization Index Modulation (QIM), which maps content values to a set of possible encoded values, which are then embedded invisibly into an image [16]. QIM, originally a watermarking technique, is more robust to image modification than many other algorithms and can be made less detectable with statistical restoration, a process which attempts to restore the statistical properties of the cover image [17]. Due to the higher amount of information that can be encoded in each image, PicFS uses the F5 algorithm that encodes information in the lowest significant bit of image pixels [8].

User anonymization to dissociate IP addresses from web traffic has been well studied in the context of privacy. The developers of FreeNet employ a distributed filesystem built over anonymous networks. However, this approach does not address full privacy as it requires some trust of all other users of the system [18]. Goldschlag et al. presented the onion router, which allows users to achieve anonymous communication on the Internet by hiding among a group of other users [19]. This approach was used in APFS (Anonymous Peer-to-peer File Sharing), which aims to preserve privacy of file sharers in a P2P network [20]. PicFS uses Tor, an implementation of second generation onion routing [10] [11]. Tor utilizes an overlay network of onion routers to create random circuits from source to destination, replacing traditional direct routes. Anonymity is then achieved through disguising the identities of both the sender and receiver.

Finally, although the subject of finding plausible images for steganography has not been directly studied, numerous techniques exist in computer graphics for constructing novel images from some initial seed set. Methods range from traditional morphing techniques to advanced image-based rendering, which uses a base set of images to interpolate or extrapolate realistic novel views from the proper combinations of the input images [21] [22]. However, these methods are insufficient for automatically generating images as they require too much knowledge of relationships between a large set of seed pictures to use effectively. Instead, we rely on a large body of work on detecting distinct shots in video sequences [23] [24].

VI. Conclusion

In this paper we present PicFS, a filesystem that provides self-enforcing privacy while using an untrusted media sharing services for storage. PicFS represents a realizable design and implementation as well as a sound way to generate thousands of plausible, unique images from home video sequences. With this filesystem, users will be able to share their information with chosen individuals using the vast resources of media sharing services - without sacrificing privacy to the services.

Although we discuss some optimizations, performance is not a focus. This stems mostly from the fact that we are building a filesystem to provide plausible deniability before

everything else. We provide an Image-Based On-Disk cache to improve performance, but there are many other possible enhancements. The most promising is an in-memory cache, although it could negatively impact plausible deniability. Other possible enhancements include pre-fetching and pre-decoding.

PicFS is a network filesystem implementation of a filesystem that supports plausible deniability. However, it would be more versatile as a distributed filesystem incorporating technologies like CFS [25], Chord [26], or Symphony [27]. By moving to a distributed model, PicFS could potentially provide better protections against suspicious hot spots as well as improve performance with a distributed cache infrastructure.

Images are a useful steganography medium, but it is possible that methods for video steganography might offer greater capacity for encoding information. Video sharing sites today are prominent and the great variety of videos currently uploaded to sites such as YouTube could provide excellent opportunities for hiding data, although the upload formats are typically restrictive. We would also like to explore techniques to make our image selection algorithm sensitive to qualities of images that make them more robust against steganalysis. Some possible directions for this approach include preferring images with high amounts of texture and certain kinds of noise, or even filtering images slightly to improve statistics.

VII. Acknowledgments

The authors thank Dave Evans and Duane Merrill for their help in contributing ideas, and anonymous reviewers for the comments that helped improving this paper.

References

[1] A. Baliga, J. Kilian, and L. Iftode, "A web based covert file system," in *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[2] "Flickr api." [Online]. Available: http://www.flickr.com/services/api/

[3] A. D. McDonald and M. G. Kuhn, "Stegfs: A steganographic file system for linux," in *IH '99: Proceedings of the Third International Workshop on Information Hiding*. London, UK: Springer-Verlag, 2000, pp. 462–477.

[4] K. Engel, *Real-time volume graphics*. AK Peters Ltd, 2006.

[5] P. Bhagwat, D. Gupta, and R. Moona, "Design and implementation of a file system with on-the-fly data compression for GNU/linux," *Software - Practice and Experience*, vol. 29, no. 10, pp. 863–874, 1999.

[6] M. Szeredi, "Fuse: filesystem in user-space for linux," 2005. [Online]. Available: http://fuse.sourceforge.net/

[7] "Fuse-j java bindings for fuse (filesystem in userspace)." [Online]. Available: http://sourceforge.net/projects/fuse-j

[8] A. Westfeld, "F5-a steganographic algorithm," in *IHW '01: Proceedings of the 4th International Workshop on Information Hiding*. London, UK: Springer-Verlag, 2001, pp. 289–302.

[9] "Quickr flickr: java access to the flickr api." [Online]. Available: https://quickr-flickr.dev.java.net/

[10] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[11] "Tor project." [Online]. Available: http://www.torproject.org

[12] A. D. Miller and W. K. Edwards, "Give and take: a study of consumer photo-sharing culture and practice," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 2007, pp. 347–356.

[13] "Flickrfs." [Online]. Available: http://sourceforge.net/projects/flickrfs

[14] "Gnicker." [Online]. Available: http://gnicker.sourceforge.net

[15] W. Norcutt, "The iozone filesystem benchmark." [Online]. Available: http://www.iozone.org/

[16] B. Chen and G. W. Wornell, "Quantization index modulation methods for digital watermarking and information embedding of multimedia," *J. VLSI Signal Process. Syst.*, vol. 27, no. 1-2, pp. 7–33, 2001.

[17] K. Solanki, K. Sullivan, U. Madhow, B. Manjunath, and S. Chandrasekaran, "Provably secure steganography: Achieving zero k-l divergence using statistical restoration," *Image Processing, 2006 IEEE International Conference on*, pp. 125–128, 8-11 Oct. 2006.

[18] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: a distributed anonymous information storage and retrieval system," in *International workshop on Designing privacy enhancing technologies*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 46–66.

[19] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing for anonymous and private internet connections," *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[20] C. Shields, "Responder anonymity and anonymous peer-to-peer file sharing," in *ICNP '01: Proceedings of the Ninth International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 2001, p. 272.

[21] S. M. Seitz and C. R. Dyer, "View morphing," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1996, pp. 21–30.

[22] C. Buehler, M. Bosse, L. McMillan, S. Gortler, and M. Cohen, "Unstructured lumigraph rendering," in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 2001, pp. 425–432.

[23] M. K. Mandal, F. M. Idris, and S. Panchanathan, "A critical evaluation of image and video indexing techniques in the compressed domain." *Image Vision Comput.*, vol. 17, no. 7, pp. 513–529, 1999. [Online]. Available: http://dblp.uni-trier.de/db/journals/ivc/ivc17.html#MandalIP99

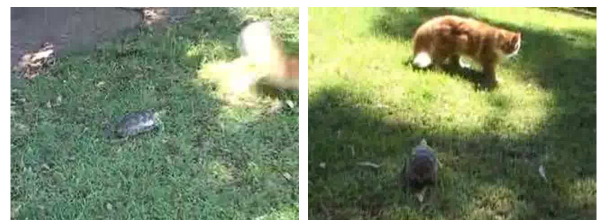
[24] S. L. M.-C. Lee, "Effective detection of various wipe transitions," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 17, no. 6, pp. 663–673, June 2007.

[25] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 202–215, 2001.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 149–160.

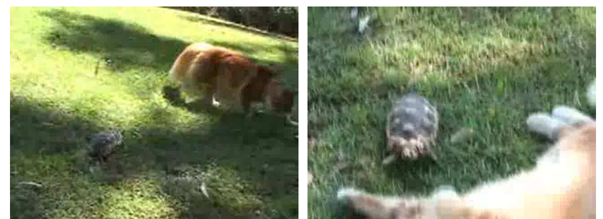
[27] G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: distributed hashing in a small world," in *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 2003, pp. 10–10.

Appendix A
A case study: images taken from a home video



(a) Frame 245: shot change from cat to turtle

(b) Frame 493: camera follows cat



(c) Frame 523: camera moves to follow action

(d) Frame 858: cat rolls over