Mnemosyne: Peer-to-Peer Steganographic Storage

Steven Hand* and Timothy Roscoe
Sprint Advanced Technology Lab
1 Adrian Court, Burlingame, CA 94010, USA
steven.hand@cl.cam.ac.uk
troscoe@sprintlabs.com

Abstract

We present the design of Mnemosyne[1], a peer-to-peer steganographic storage service. Mnemosyne provides a high level of privacy and plausible deniability by using a large amount of shared distributed storage to hide data. Blocks are dispersed by secure hashing, and loss codes used for resiliency. We discuss the design of the system, and the challenges posed by traffic analysis.

1 Introduction and Motivation

A steganographic file system, first presented in [2], has the property that it gives a user strong protection against being compelled to disclose (all) its contents. Attackers not in possession of the secret are unable to acquire the contents of files, and they cannot even gain information about whether a given file is present or not. In effect, the system allows an author to plausibly deny the existence of most files[2] in the system.

A distributed, peer-to-peer steganographic storage system like Mnemosyne has further interesting properties. Firstly, in common with systems like FreeNet [6], storage providers can offer a service without being able to know what is being stored. This property may be attractive to a service provider concerned about liability as it *de facto* confers something akin to common-carrier status on the provider.

Secondly, for a single user desiring to store files securely, a distributed steganographic storage system makes information less susceptible to machine failure or denial-of-service: a local storage medium can always be stolen, but a peer-to-peer system is harder to shut down.

Thirdly, such a system may also be used as a shared-memory communication medium with steganographic properties: this allows interpersonal messaging with a high degree of privacy.

A system with these properties is of great potential use to the modern business traveler.

Mnemosyne takes advantage of the widespread availability and low cost of network bandwidth and disk space. The system comprises servers that provide unreliable block storage, and clients which write and read blocks to and from the servers. A node can serve the function of server and client simultaneously. The servers collectively comprise a peer-to-peer system: a centralized organisation or authority is neither required nor desirable.

Before describing Mnemosyne itself, we present a description of our *local* steganographic file system. We do this for two reasons. Firstly, many of the principles of local steganographic systems carry over to the distributed case, and discussion of these helps establish context for describing Mnemosyne later. Secondly, our implementation of the local case differs from previous systems (most notably that described in [13]) in ways significant when extending the concept to a full peer-to-peer system.

2 A Local Steganographic File System

Anderson et. al. [2] describe two approaches to the steganographic storage of data. In the first, randomly-filled "cover files" are created, and user files are "written" by altering a subset of the cover files (determined by a passphrase) so that the user file is the XOR of that subset.

The second construction, followed here, assumes a disk which can store X blocks of data. To prepare this for use, we first write random data to every block. Then to store a file we simply encrypt each

---

*On leave from University of Cambridge Computer Laboratory, JJ Thompson Ave, Cambridge CB3 0FD, UK

[1] Pronounced *ne moz'nē*.

[2] At least some files must be revealed to justify the existence of the system itself.

block and write it to a pseudo-randomly chosen location (e.g. one determined by hashing the filename and block number with a secret key). With a sufficiently good cipher and key, the encrypted blocks will be indistinguishable from the random substrate, and so an attacker cannot even determine the existence of the file. On the other hand, someone privy to the filename and key can reconstruct the pseudo-random sequence, retrieve the encrypted blocks, and decrypt them.

This leads to the problem of *collisions*, where blocks are overwritten on the disk by subsequent files. The well-known "birthday paradox" makes this quite likely with even a small load factor (ratio of file blocks to total blocks on the disk), and so replication is used: each block is written to the disk at n independent locations.

We describe our implementation of this scheme (over Linux) by first describing the process for replicating a block on the disk, and then discussing file structures built over this facility.

Writing and Reading a Single Block

Writing a block to the local steganographic file system requires a user's key K, the block data itself, and two further pieces of information: an *initial hash value* h_0 for the block, and a *validity check* (a way of determining whether the block data has been corrupted or not). The initial hash value and validity check vary according to whether one is storing directory blocks, inodes, or file blocks (see below). To write (or overwrite) a block, the procedure is:

- The user computes a sequence of n hash values $h_0,\ h_1 = H(h_0),\ \ldots,\ h_{n-1} = H(h_{n-2})$
- Replica i ($0 \leq i < n$) is encrypted under the key $k_i = E_K(h_i)$ and stored at block number $b_i = h_i \bmod X$, where X is the number of blocks on the disk[3].

To read a block given the key K and an initial hash value h_0, we read and decrypt each replica in turn from block b_i until we have a block which passes the validity check. If no blocks pass the check, the block is deemed lost. The use of a per-replica key k_i ensures that replicas are not identical on disk. It also means that K alone is not sufficient to determine the validity of a given block.

In our implementation we use SHA256 as the hash function H and AES as the block cipher for encrypting blocks, choosing a key size of 256 bits to match the size of hash values.

Directories, Inodes and Files

We build a file system over this basic block facility using *directories*, *inodes*, and *file blocks*.

In Mnemosyne directories are used to aggregate files which share a common key K. A directory block contains a known textual name for the directory itself, and a list of textual file names. The validity check for a directory block is the presence of the name of the directory in the block. The initial hash value used for writing a directory block is obtained by hashing the directory name and XORing the result with the key, K. Using K in this way prevents different users from overwriting each others' blocks deterministically when they choose identical directory names.

Each file is represented in the file system by an inode block. The inode block is stored using an initial hash value obtained by concatenating the directory name and file name to produce a pathname, hashing this pathname, and then XORing the result with the key K as before; this is the reason directory blocks need only store filenames. The filename is also stored in the inode block, acting as the validity check. Note that in this scheme directories themselves are completely optional, serving simply as a mnemonic device for a set of file names. Directory *names*, on the other hand, are necessary components of path names.

In addition to this file name, the inode block for a file consists of a list of zero or more $\{$*initval*, *checkval*$\}$ pairs, one for each block in the file. These pairs of 256-bit values are analogous to the block pointers in a conventional file system. *initval*, chosen at random, is the initial hash value for locating the file block replicas. *checkval* is a secure hash of the file block and is used as the validity check for file blocks since, unlike directories and inodes, no redundant information is stored within file blocks.

Discussion

As discussed in [2], the choice of n (the number of

[3] We believe that using subkeys $k_i = E_K(h_i)$ improves over $k_i = K \oplus h_i$, used in an earlier version of this paper.

replicas) is critical. Intuitively, there is a tension between increasing n to make an individual replica set more resilient and decreasing n to reduce the overall number of blocks written (and hence potentially overwritten). Analytical solutions are difficult to obtain, but initial experiments (see §5) suggest overall replication factors of 2 to 8.

This results in a significant cost in disk space, but the factor is constant (while large) over a conventional file system and so we consider it acceptable since what is offered is a specialised service for certain types of information. The key point is that the service scales well in disk size, not how much disk space is required for a given load.

The systems in [2] and [13] present a hierarchical security model, which can be generalised to a matrix controlling access by a fixed number of users (or principals) to a fixed number of security "levels". We eschew such an approach in favor of a simpler, flat key space: if a user possesses a key and the name of a directory, he or she is able to read and write files in that directory. This has two advantages. Firstly, the indefinite number of keys makes it less likely that all the keys can be extracted from a user under duress. Secondly, and more importantly, when we extend the system to a distributed, peer-to-peer scenario, we cannot know in advance how many users, files, or available blocks there will be. The matrix model implies an authority that at least allocates rows of the matrix to users; the flat key space model is more appropriate for a federated, peer-to-peer world.

Note also that even in this local implementation, users don't have to trust the block store, as long as most of the time it doesn't throw away blocks, and the load factor isn't so great that too many blocks have all their replicas overwritten. This feature is significant when we extend the system to the peer-to-peer case.

3 Distributing the Block Store

We first present here the obvious extension of the local system to the distributed case, and then discuss refinements and modifications of this in §4.

Assume there exists a set of M nodes each of which wishes to contribute N blocks of storage to the collective. We can logically treat this as an array of MN blocks, and proceed to store and retrieve files and directories as described in the previous section. Rather than storing the block replica i at block number (h_i mod X), we need to derive both a node identifier and a block number on that node from the 256-bit hash value.

We can do this by leveraging existing work on peer-to-peer object location and routing schemes. We use Tapestry [21], although any of [15, 18, 19] could serve. All we require is routing of messages tagged with arbitrary n-bit identifiers to nodes.

In Mnemosyne, even in the local case, blocks read from the disk need not be correct. Instead, the validity of blocks is explicitly checked after they have been retrieved. This allows us to build a distributed block store in which there is little reliance on the integrity of any single node. The only operations a node need implement are:

- **putBlock**(*blockid*, *data*)
- **getBlock**(*blockid*) → *data*

The semantics of these are weak: **putBlock** simply requests that the node store the block *data* in such a way that it may be subsequently retrieved by **getBlock** using an identical *blockid*. However, the node is not required (and may not even be able) to ensure this — that is, the **putBlock** operation has at-most-once semantics.

getBlock requests that the node return whatever data it has associated with the given *blockid*. However the node may ignore the request, or return any block of data it chooses. The client will determine if the information is valid after it has been received.

Using this service we construct a first attempt at a distributed steganographic storage system. We assume a set of Tapestry nodes, each of which exports the same amount of storage space (e.g. 1GB arranged as 2^{20} blocks of 1KB each).

To store a block, we follow the block replication algorithm described in §2, except that we choose the leading 160 bits of h_i as the Tapestry node identifier N_i, and the next (e.g.) 20 bits as the blockid b_i on that node.

To retrieve a block, the client requests blockids b_i from nodes N_i. We note that these requests may proceed in parallel. The client then tries to decrypt and verify each block until a valid one is found. If none is found, the block is deemed lost.

We can build directories and files over this basic system as in the local case. Note that it is not necessary for an individual node to respond "correctly" or even at all. All that the client requires is that at least one of the replicas for a block is still available. This makes it difficult for an attacker without a key to destroy any particular piece of information.

We note that with lookup services having a notion of unique "successor" for a node (such as Chord), a new node joining the system can initialize by duplicating the entire block store of its successor; neither the new nor the existing node need be aware of which blocks are "valid". This duplication means that the new node will immediately respond correctly to any **getBlock** requests made of it. With Plaxton-based systems like Tapestry, there are several nodes analogous to a Chord successor (roughly 4 in Tapestry), but we can still usefully copy fractions of the stores of these nodes.

Discussion

This system has the following useful properties:

Firstly, given the obvious implementation for a "cooperative" node (viz. to reserve 1GB of space and then store and retrieve blocks as requested), the owner of the node can plausibly deny knowledge of any of the contents. Indeed, they will in general be unaware even of which blocks are in use.

Secondly, a node can choose to use a smaller amount of storage by mapping the 20-bit block identifiers down to $k < 20$ bits. This produces a less resilient but still valid store.

Finally, a node can provide more than 2^{20} blocks simply by obtaining more than one node identifier (e.g. as with "virtual servers" in CFS [7]).

In summary, Mnemosyne provides information hiding at two levels: first, data is striped widely across different nodes each of which is unaware of the other nodes holding parts of the file. Second, each individual node embeds encrypted blocks in a random substrate, thus making them indistinguishable from one another (without a valid key).

4 Enhancements

Our first enhancement to this basic scheme is to replace simple replication with the information dispersal algorithm (IDA) [14]. Using this, an author chooses two numbers $m \geq n$ and encodes information to be published into m blocks such that any n of these are sufficient to reassemble the original data. Using the IDA gives us much better resilience for a given "redundancy factor" (m/n).

The IDA requires that we replace our simple redundancy-based validity checks with a cryptographic authenticity check on each dispersed block; our current implementation uses the AES in the new OCB mode [17] to get both privacy and authentication in one pass, although CBC-MAC, XCBC, or IACBC [11] would also suffice.

Readers now independently retrieve m' of the m blocks where $m' \geq n$ is chosen by each user so as to obtain a "reasonable" expectation that at least n blocks will be valid. The publisher chooses m so that $\binom{m}{m'}$ is large enough for likely values of m'. Concurrently, readers retrieve r other blocks chosen at random and discard them on receipt.

This allows us to more efficiently address the problem of traffic analysis whereby an adversary who can snoop packet transfers can infer the existence (and possibly location) of a file. If desired some of the r blocks could represent a known piece of content to provide "deniable encryption" [3].

We also use the flexible dispersal of the IDA to address the problem that any reader of a file can replace or destroy its contents. To combat hijacking we can simply allow authors to use pseudonymous digital signatures, much as in [8]. To prevent destruction of file content we introduce explicit *location keys*: randomly chosen values which are XORed with a (directory or file) name's hash in order to choose the set of m storage locations. An author can now choose any l different location keys and write lm blocks (assuming no collisions).

Each reader is now provided with the name, the encryption key, a location key, and m. This prevents a single reader from destroying more than a fraction of the total replicas. Furthermore, if l is never disclosed, an author under duress can claim to delete all copies but later recover the information, as in the Eternity Service [1].

Writing of data under Mnemosyne also holds interesting challenges. A per-node rate limiter protects against brute-force denial-of-service attacks, as an alternative to the Hash-Cash scheme in [20]. We note that Mnemosyne is less susceptible to such attacks due to its sparse use of storage space.

Nonetheless, over time more and more of a document $\mathcal{D}$'s replicas will be overwritten until at some point it is no longer accessible. To avoid this we need to periodically refresh $\mathcal{D}$. Choosing a good refresh interval in the absence of global knowledge is difficult, and so we expect users to err on the side of caution (i.e. to rewrite rather frequently).

The refresh of files provides us with another traffic analysis problem. We could attempt to resolve this as before: i.e. arrange for additional writes to occur so that the "real" ones may be concealed. Unfortunately this would result in a large number of additional writes, and hence collisions.

A better scheme is to require that all messages to block stores are encrypted and of the same size. A single bit in a request is used to specify if the accompanying payload is to be written. In all cases, a block of data is returned. This makes it impossible for an eavesdropper to distinguish between reads and writes, making traffic analysis more difficult. If bandwidth is cheap, an obvious extension is for all users to issue an isochronous stream of requests in which "real" requests are occasionally embedded.

5 Simulation

Two of the key parameters in the system are the choices of m and n for a given file since there is a tension between maximizing the capacity of the store, and increasing the resilience of each file. This is further complicated in the decentralized case since users are free to choose m and n independently, and no-one knows how many users there are, or how much traffic they are generating. Nevertheless, to give some idea of the trade-offs involved, we present here some initial simulation results for fixed-size files and uniform coding schemes.

The simulation repeatedly adds files to a store of 4 million blocks and keeps track of how many files are still retrievable: i.e. files for which n blocks have not been overwritten in the store. Starting with an empty store, this number converges to a limit for each m as files are added, and we call this limit the capacity of the store. Figure 1 shows how the capacity changes with choice of m. For low values, the birthday paradox comes into play and capacity is limited. As m increases, capacity increases until the large number of writes per file reduce it again.

Of more importance to actual users of the system is the expected lifetime of a file: how long a file lasts before it becomes inaccessible. Figure 2 shows cumulative distributions of file lifetimes (measured as the number of subsequent file writes) for the same coding parameters as before. Of interest to users is where these curves intersect some low probability of file loss, thus giving an idea of how often a file needs to be refreshed.

6 Implementation

We have built a working implementation of Mnemosyne. The client is implemented in C and makes use of freely available implementations of SHA256 and the AES; it provides a command-line interface with operations for creating directories and copying files between Mnemosyne and the Unix filing system.

We use the IDA with polynomials over $GF(2^{16})$ for dispersal, and OCB-AES to provide combined encryption and authenticity. Local performance is plausible: we can copy in at around 64KB/s, and out at circa 375KB/s (for $n = 32$, $m = 96$).

The distributed block storage functionality is implemented in Java over Tapestry [21]. The client uses a simple UDP-based protocol to communicate with a randomly picked Tapestry node. Read and write requests are then routed through Tapestry to the appropriate block store. Responses are returned to the client via the original Tapestry node. In early tests using 3 co-located nodes we can copy in files at around 80KB/s, and copy them out at 160KB/s.

We intend to make the code for Mnemosyne available in the near future.

7 Relation to Existing Work

Some recent systems have used distribution and self-organisation to provide robustness and availability [1, 7, 9, 10, 12]. Other systems use their decentralised nature to provide anonymity of access and prevent censorship [4, 6, 8, 20].

Mnemosyne is more aligned with the latter class of system. However it provides in addition plausible deniability for clients, and is more suited to private storage and messaging applications than to the wide-scale publishing of data. Mnemosyne also shares some common ground with private information retrieval systems [5, 16].

REFERENCES

[1] ANDERSON, R. The Eternity Service. In *Proc. of the 1st*

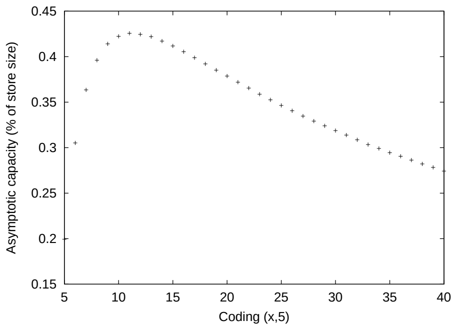

Figure 1: Capacity of a simulated 4Mblock store

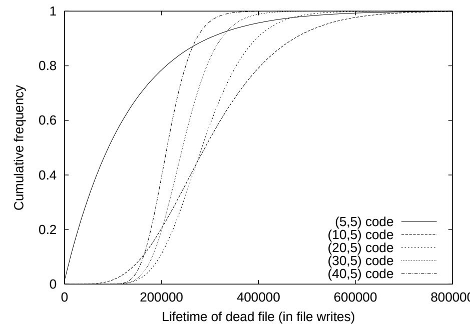

Figure 2: File lifetimes in a simulated 4Mblock store

International Conference on the Theory and Applications of Cryptology (PRAGOCRYPT'96) (1996).

[2] ANDERSON, R., NEEDHAM, R., AND SHAMIR, A. The Steganographic File System. In *IWIH: International Workshop on Information Hiding* (1998).

[3] CANETTI, R., DWORK, C., NAOR, M., AND OSTROVSKY, R. Deniable encryption. *Lecture Notes in Computer Science 1294* (1997), 90–104.

[4] CHAUM, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (February 1981), 84–88.

[5] CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., AND SUDAN, M. Private Information Retrieval. In *IEEE Symposium on Foundations of Computer Science* (1995), pp. 41–50.

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability* (July 2000), pp. 46–66.

[7] DABEK, F., KAASHOEK, M., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. SOSP '01, Banff, Canada.* (October 2001).

[8] DINGLEDINE, R., FREEDMAN, M. J., AND MOLNAR, D. The Free Haven Project: Distributed Anonymous Storage Service. In *Workshop on Design Issues in Anonymity and Unobservability* (July 2000), pp. 67–95.

[9] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS-VIII, Schloss Elmau, Germany* (May 2001).

[10] IYENGAR, A., CAHN, R., GARAY, J. A., AND JUTLA, C. Design and implementation of a secure distributed data repository. In *Proc. of the 14th IFIP International Information Security Conference (SEC 98)* (1998).

[11] JUTLA, C. S. Encryption modes with almost free message integrity. Cryptology ePrint Archive, Report 2000/039, 2000. `http://eprint.iacr.org/`.

[12] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO., B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. ASPLOS 2000* (November 2000).

[13] MCDONALD, A. D., AND KUHN, M. G. StegFS: A Steganographic File System for Linux. In *Information Hiding* (1999), no. 1768 in LNCS, pp. 462–477.

[14] RABIN, M. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (April 1989), 335–348.

[15] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM 2001, San Diego, California, USA.* (August 2001).

[16] REITER, M. K., AND RUBIN, A. D. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security 1*, 1 (1998), 66–92.

[17] ROGAWAY, P., BELLARE, M., BLACK, J., AND KROVETZ, T. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)* (August 2001), ACM Press.

[18] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM Internation Conference on Distributed Systems Platforms (Middleware 2001)* (November 2001).

[19] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM 2001, San Diego, California, USA.* (August 2001).

[20] WALDMAN, M., RUBIN, A. D., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. of the 9th USENIX Security Symposium* (August 2000), pp. 59–72.

[21] ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Tech. Rep. UCB//CSD-01-1141, U. C. Berkeley, April 2000.