Efficient Sharing of Encrypted Data*

Krista Bennett, Christian Grothoff, Tzvetan Horozov, and Ioana Patrascu

S^3 lab and CERIAS,
Department of Computer Sciences, Purdue University
{klb,grothoff,horozov,patrascu}@cs.purdue.edu
http://,,,,,,,,o,,,e,,o,f,,,,z.oszar.co,/software/GNUnet/

Abstract. This paper describes the design of a censorship-resistant distributed file sharing protocol which has been implemented on top of GNUnet, an anonymous, reputation-based network. We focus on the encoding layer of the GNUnet file-sharing protocol which supports efficient dissemination of encrypted data as well as queries over encrypted data. The main idea advocated in this paper is that simple cryptographic techniques are sufficient to engineer an efficient data encoding that can make it significantly harder to selectively censor information. Our encoding allows users to share files encrypted under descriptive keys which are the basis for querying the network for content. A key property of our encoding is that intermediaries can filter invalid encrypted replies without being able to decrypt the query or the reply. Files are stored in small chunks which are distributed and replicated automatically by the GNUnet infrastructure. Additionally, data files may be stored in plaintext or encrypted form or as a combination of both and encrypted on demand.

1 Introduction

Internet censorship can appear in many forms. Whether it is government regulation aimed to stifle political dissent or a campaign orchestrated by pressure groups against opponents, a common feature of all forms of censorship is that each must identify objectionable information and prevent its sharing. The goal of our research is to study requirements and design a censorship-resistant file-sharing protocol. In this paper, we focus on the design of the encoding layer of such a protocol and briefly describe a prototype implementation that we have realized on top of the GNUnet networking infrastructure [1].

Keeping published documents available in the face of powerful adversaries requires an array of techniques. Censorship can be enacted in many ways, such as sweeping through users hard drives to discover objectionable content, filtering user queries to catch requests, and even denial of service attacks against servers providing storage for the documents. Our goal is to design a file-sharing protocol that protects against all of these attempts to limit the free flow of information.

* Portions of this work were supported by sponsors of CERIAS

In particular, both content and queries should always be encrypted; furthermore, servers as well as all intermediate hosts should be oblivious of the data they are actually storing such that they can neither be singled out for attack nor held liable in a court of law.

To address this challenge, we have implemented a censorship-resistant file-sharing protocol on top of the GNUnet anonymous reputation-based network. Our system protects servers and users through an original encryption scheme. The design of the cipher was motivated by the idea of a wide area distributed network where many parties want to share data. Identical data may be inserted by many independent parties at any time. Some users may want to keep a copy of their files in plaintext on their drive – and may not want to waste space on a second, encrypted copy – while others must always keep all the data in encrypted form. Files are identified by keywords and retrieved by issuing queries for all documents matching a given keyword. Some complex queries, such as boolean formulas over keywords, should also be supported.

This paper focuses on the content and query encryption techniques used within GNUnet. Other aspects such as host discovery, host authentication, and the GNUnet economic reputation-trading model (to prevent denial of service attacks) will not be addressed here.

The GNUnet encryption scheme is remarkable in that it allows identical files encrypted under different keys to yield the same ciphertext, modulo a small block of metadata. These files can then be split into small blocks and distributed (and replicated if need be) across hosts in a GNUnet system to balance load. The network can thus automatically optimize the storage requirements of a document even if multiple users insert the same files under different search keys at different times. Another implication is that a document can be entered under many search keys at little cost in space. The overhead in space of our scheme is less than 4% of the plaintext size, with one 1KB per additional copy of the file. The scheme is based on hash trees [6] and allows swarm distribution [7]. Encryption speed is about 3MB/sec and decryption is a bit slower, about 350KB/sec, as our decryption code is multi-threaded and has to deal with file fragments being returned by different hosts out-of-order.

The remainder of the paper is organized as follows. Section 2 begins by listing the requirements of file-sharing protocols and motivating our choices. Then, in Section 3, we contrast this work with other censorship-resistant systems having similar goals. Section 4 describes the details of the GNUnet encoding scheme. Finally, Section 5 describes some possible attacks.

2 Requirements for censorship-resistant file-sharing

This section briefly describes the requirements for a censorship-resistant file-sharing protocol. These requirements will then be used to drive our design and, in section 3, to assess the appropriateness of related systems.

We start with a scenario that tries to highlight a common use-case for such a protocol:

> Suppose that *Alice* inserts a file F containing the text of Salman Rushdie's Satanic Verses in the network under the keyword `"satanicverses"`, and the same file is independently inserted by *Bob* under keyword `"rushdie's verses"`. Furthermore, *Alice* decides to keep file F in plaintext on her drive and is not willing to dedicate space for a second, encrypted copy on the drive. *Bob* on the other hand, lives under an oppressive regime and is afraid of being caught with the file, so he wants to make sure that the file cannot be found on his drive. Thus he encrypts the file, and his computer starts to distribute pieces of the file all over the network. What we want to be able to do is to have a user *Carol* perform a query `"satanicverses" AND "rushie's verses"` and retrieve the file F, potentially retrieving a few pieces from the copy that *Bob* inserted and distributed as well as a few other pieces that *Alice* contributed. *Alice* should not have to perform encryption on the pieces of F that *Carol* obtained from other parts of network. In this case, the encryption of F guarantees that participating servers can claim ignorance about the contents of F.

We now turn to the requirement placed on a file-sharing protocol that could support the above use case.

2.1 Plausible Deniability

In order to protect intermediaries, it is desirable that these hosts be unable to discover what they are sending. Servers must be able to deny knowledge of the content they store or transport. Intermediate and responding hosts should neither be able to find out what the query is about nor what the contents of the data returned are. Many modern systems [3] overcome this by hashing the query and searching for the hash, which does not reveal the original keyword.

Additionally, content migration is used both to ensure that the original sender cannot be located as well as to distribute the load. The host that initially inserted the content can then forget the "key" and claim ignorance, even if the content is still stored locally; as long as the adversary has not performed full traffic analysis, it remains plausible that content could have come from another node.

In GNUnet, users are anonymous. Thus, our encoding scheme is aimed at protecting the *servers*, not the individuals requesting the data. Servers must be able to deny knowledge of the content they index, store or transport.

If hosts are unable to see which content they are serving, they cannot exercise editorial control by selectively deleting or modifying content. Exercising editorial control implies liability for content. A lack of editorial control usually relieves the provider from responsibility for the nature of the data served.[1]

[1] In *Stratton Oakmont v. Prodigy*, an Internet provider was found liable for hosting a bulletin-board where offensive messages had been posted. This is because the provider had hired individuals to exercise editorial control over content. If the provider had merely published messages from users without interference, the provider would probably not have been held accountable[10].

2.2 Content Retrieval

Searching for a file by a unique keyword creates a problem. The user needs some means of actually obtaining the keyword in the first place. If no secrecy is required, this problem is equivalent to using a search engine to obtain URLs from keywords on the World Wide Web.

The problem we try to solve is that of allowing complex searches without exposing the search-string to the network. If the query consists of several keywords, an obvious approach is to search for each of the keywords, download and decrypt all matching files, and then compare their contents. This is clearly impractical. An ideal system would support regular expressions for queries and return only all files that exactly match the regular expression.

2.3 File Distribution Mechanism

In order to be able to move content from host to host in the network, it is desirable that large files can be split. Fine-grained pieces of data are quickly moved from site to site. If it were necessary to transfer large files *as a whole* for a valid transfer to occur, incomplete transfers would result in a huge waste of resources.

Additionally, we want to be able to combine identical files from different sources as illustrated in the motivating example and to allow each source to specify its individual list of keywords for the file.

2.4 Scalability

Naturally, in order for a file-sharing system to scale, the overhead of the encoding should be small. The size of an encoded file should not be significantly larger than the original file. The memory requirements to encrypt or decrypt a file should also be minimal.

Finally, users should be able to store a file in plaintext on their local drive (perhaps because they are using it), without doubling space requirements; the encoding should allow files to be served that are stored locally in plaintext without requiring the storage of a second copy in encrypted form.[2] In addition to the fact that users may assemble the same file using parts from different servers, this storage option requires the ability to encrypt small parts from the middle of the file. For efficiency, this should of course not require reading (or even encrypting) any other part of the file. To the best of our knowledge, GNUnet is the first system that allows this kind of random access on-demand encryption for arbitrary parts of a file.

[2] This would of course reduce deniability; however, we value giving users the choice between efficiency and security.

3 Related Work

Censorship-resistant publishing systems are commonly based on three techniques:

1. replication, to ensure that a document is not lost if a single copy is destroyed;
2. distribution, to keep content available if a site goes down;
3. encryption, to prevent servers from exercising editorial control over the content they store.

All systems differ widely in their specific implementations of these three building blocks. While GNUnet also chooses a unique approach to replication and distribution, we do not discuss these aspects in this paper, including the description of related work. Instead, we focus solely on encryption.

3.1 Freenet

The main content encoding scheme used in *Freenet* [2, 4] uses the hash of the content as the key (CHK). Other key types indirect to content-hash keys. *Freenet* has several different types of keys. The different key types are used to allow additional features such as content signing, personal namespaces or splitting of content. While this encryption scheme yields the same encrypted data for independent encryptions, it has some disadvantages.

One disadvantage of *Freenet* is that it does not allow direct sharing of files from the local drive without encrypting and inserting them first. Thus, to keep content easily accessible on the host, a node operator must keep a local copy of the unencrypted file in addition to the encrypted content in the *Freenet* database.

Freenet also requires unique keys which may be non-trivial to guess. As collisions for keys are not allowed, multiple results can never be found. If an attacker inserts meaningless data under common keywords, *Freenet* will preserve this content because it is frequently requested and make it impossible to use that keyword for useful data. A solution to the problem of finding keywords is approached by the use of keyservers which provide indices to all known keys. The disadvantage of these keyservers is that they must be maintained; in reality, they often index content which is no longer available.

The CHK encoding does not allow for file completion with content from truncated files. In distributed systems, aborted downloads for large files are fairly frequent. Because the hash of the file is different if even one single bit was not transferred, the entire transfer becomes useless if not completed.

3.2 Mojo Nation

Mojo Nation is a distributed file-sharing system based on micropayments. This system first breaks the original file into several pieces (the larger the file, the greater the number of pieces). Each piece is then broken into eight blocks, any four of which are sufficient to reconstruct the original piece. These data blocks are hashed to generate a unique identity tag. This tag is then later used to retrieve the blocks. A *sharemap* or *Dinode* is used to reconstruct the file.

This protocol distinguishes strictly between searching and downloading. This makes it easy for the system to censor certain queries. While this is an intended feature of the Mojo Nation system, it is unacceptable for GNUnet.

3.3 Free Haven

In Free Haven, documents are split into n shares which are distributed over a number of servers. Any k out of n shares are sufficient to reconstruct the entire document [8]. To retrieve a document, the user must obtain the key that was used to sign the document from some source outside the Free Haven network. The user then forwards the key to the server which encrypts and sends its share of the document.

Because shares are stored in plaintext (only the transfer is encrypted), hosts can exercise editorial control over content. Also, the k out of n share reconstruction scheme does not allow reconstruction of a file by mixing shares from independent insertions (see section 4.2).

The requirement that the users obtain Free Haven keys from other sources makes it impossible to search the network. Furthermore, these keys consist of random bits instead of guessable keywords. While this is desirable if content in the network is meant to be accessed only by authorized parties, it is a drawback in a system where files are intended to be publicly available.

3.4 Tangler

The Tangler network [6] is a system where new content is entangled with old content. This entanglement introduces a dependency that makes it impossible to retrieve the new content if the other content that it was entangled with is lost. While this makes it impossible to censor specific content without losing unrelated content (since the entanglement pairs are chosen randomly), this process *at least* doubles the amount of data that must be transported by the network.

Furthermore, outdated or unpopular content cannot be removed because of the entanglement scheme. Tangler's approach is quite radical as it not only disallows selective removal of content (censorship), but any removal. The Tangler protocol also makes unreasonable assumptions, such as extremely high availability of servers and synchronous decisions in a distributed network. The paper notes that the network will probably not scale.

3.5 Publius

Publius [5] is a static network that uses Rabin's file splitting algorithm [8] to distribute an encrypted file to n servers. The key used to encrypt the file can be used to retrieve the shares. The servers can not decrypt the file as the key is never entirely revealed to them.

Publius keys are random strings. The system does not provide any search mechanism and expects the users to obtain the keys from an alternate source.

3.6 Summary

Figure 1 compares the different systems discussed in this section. The most important questions are:

– Are servers prevented from decrypting the content?
– Can content inserted by independent sources be joined?
– Can content be encrypted on-demand?
– Is content split into smaller pieces?
– Is the method of encrypting the content reasonable for the goal of being censorship-resistant?
– Is a boolean search possible? (AND, OR, XOR, NOT, etc.)
– Is a search with regular expressions possible?
– Can there be different files matching the same keyword?
– How much data must be transferred per file as a multiple of the actual file size (not counting indirections due to anonymity provisions)?

System	Freenet	Mojo Nation	Free Haven	Tangler	Publius	GNUnet
server can decrypt	no	yes	no	yes	no	no
joins content	yes	yes	no	no	no	yes
on-demand encryption	no	no	no	no	no	yes
content splitting	yes	yes	yes	yes	yes	yes
censorship-resistant	yes	no	yes	yes	yes	yes
boolean search	no	yes	no	no	no	yes
regex	no	no	no	no	no	no
key collisions	no	yes	no	no	no	yes
network overhead	≈ 1	≈ 1	≈ 1	≈ 2	≈ 1	≈ 1

Fig. 1. Quick Comparison.

4 GNUnet Content representation

The design of GNUnet storage and content distribution attempts to achieve several goals:

1. Deniability for all participants;
2. Distribution of content and load;
3. Efficiency in terms of space and bandwidth.

In this section, we describe an encoding scheme for content that achieves all three goals. While the receiver can decrypt the file, none of the intermediaries can discern its contents. Additionally, the query is sent encrypted; thus, the other hosts on the network cannot compute what the user is looking for. Some minor goals have also influenced our design. First of all, we wanted to be able to perform boolean queries of the form $a\ AND\ b$, without revealing a or b. Secondly, we require content migration to be simple and fast, making it easier for GNUnet to migrate (or copy) files, thus achieving distribution and redundancy.

The remainder of this section describes how these goals are achieved by our approach. In part 4.1 we describe how files are split into *GBlocks*, which are easier to migrate. Part 4.2 describes how the *GBlocks* are encrypted in order to make it impossible for the intermediaries to determine what they are storing or serving. Part 4.3 describes how the user can query for content without exposing the (plaintext) query or the corresponding content. The following description summarizes the full scheme that we develop in this section:

1. The user gives the local GNUnet node the content C, a list of keywords K and a description D (and optionally the pseudonym P) to use.
2. GNUnet breaks C into blocks B_i, each of size 1k, and computes the hash values $H_i = H(B_i)$ and $H_i^2 = H(H(B_i))$. Random padding is added if needed.
3. Each block is then encrypted, yielding $E_i = E_{H_i}(B_i)$.
4. GNUnet stores E_i under the name H_i^2.
5. If C was larger than 1k (and thus there is more than one H_i), GNUnet groups up to 51 H_i values together with a CRC32 of the original data to form a new block of size 1k. Random padding is added if needed. Each new 1k block obtained in this manner is then processed as in step 2.
6. If the size of C was smaller than 1k (and thus there is only one hashcode H_1), GNUnet builds a root-node containing H_1, the description D, the original length of C, a CRC checksum, and optionally the pseudonym P and a signature. The total size of the root node must be less than 1k in size (the length of the description may be shortened as needed). The resulting root-node R is then padded and encrypted once for *each* keyword K yielding $R_K = E_{H(K)}(R)$.
7. Finally, for each K, the result R_K is stored under $H(H(K))$.

The space m required for a file of size n is

$$
\begin{aligned}
m &\leq n + 1k \cdot \sum_{i=0}^{\lfloor log_{51} \lceil \frac{n}{1k} \rceil \rfloor} 51^i \\
&\approx 1.02 \cdot n.
\end{aligned}
$$

4.1 *GBlocks*

In order to be able to migrate large files and to distribute load on the network, GNUnet splits large files into many small files called *GBlocks*. There are 3 types of *GBlocks*: *IBlocks*, *DBlocks* and *RBlocks*. Similar to UNIX INodes, *IBlocks* are special indirection-blocks which encode how the leaf-nodes of the file tree (the *DBlocks*) can be reassembled into the original file (see figure 2).

Splitting large files makes content migration inexpensive. Without splitting these files, it is unlikely that a node would be able to find another host that

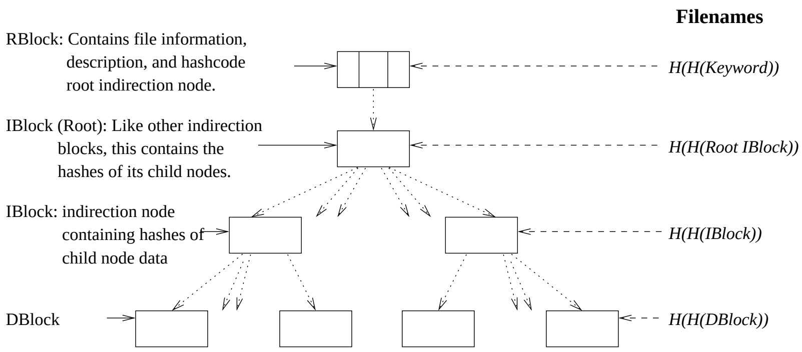

Fig. 2. Encoding of the entire file

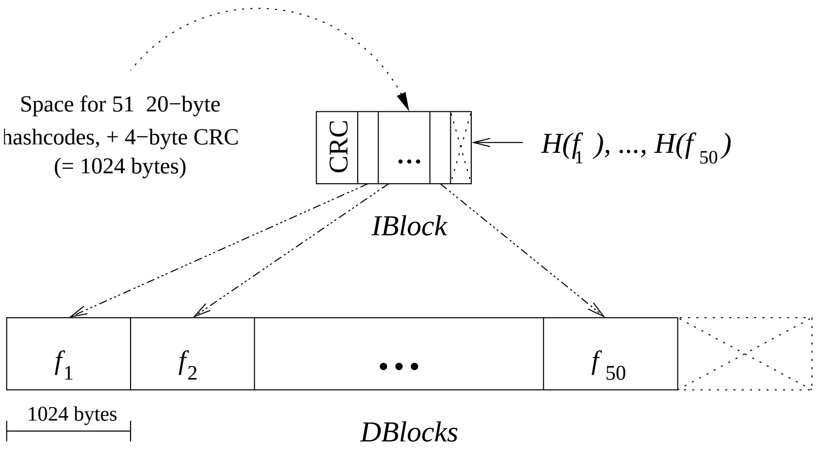

Fig. 3. Encoding of the 1k blocks

is willing (or able) to provide enough space. The traffic burst that this transfer would impose on the network is impractical, as both nodes would become very busy for a long time. Storing a large file on a single host also fails to facilitate distribution of load when a user requests this large file.

The size of all types of *GBlocks* in GNUnet is normalized to 1k. Thus, a 50k file will be split into 50 *DBlocks* f_i of length 1k each. Then, 50 RIPE-MD-160 hashcodes of the *DBlocks* are composed to form an *IBlock* $H(f_1), \ldots, H(f_{50})$, which is stored with all the other *GBlocks* (see figure 3). *GBlocks* that are less than 1k in length are padded with random information. *IBlocks* can store up to 51 hashcodes $(51 * 20\,bytes = 1020\,bytes)$ and one CRC32 checksum.

Larger files require multiple levels of indirection. A special *RBlock* on top of the file tree contains the length of the file, a description and the hashcode of the topmost *IBlock*. In this way, the user can download a single 1k *GBlock* and obtain information about the contents of the full-length corresponding file before choosing to download the rest of the file tree.

Obviously, this fine-grained splitting of the files increases the number of queries that must be made in order to obtain a file. Because the file is distributed in parts, it is entirely possible that a file will not be able to be fully reconstructed, even if some parts of the file are still available on the network. While this problem cannot be solved entirely, duplication of files in the network will make this situation less likely.

One could argue that a size of 1k is far too small. The rationale behind this file size is that UDP, the transport mechanism used by GNUnet has (on Ethernet) an optimal packet size that is slightly above 1k. As GNUnet must add headers to the packets, the size approximates this number. Furthermore, many file systems can be formatted with a block-size of 1k.

4.2 Encryption of the Content

The previous section described the splitting of files into *GBlocks*. However, this technique is not sufficient to ensure that intermediaries are unable to ascertain which content they are transmitting. That said, hiding content from intermediaries can be achieved by encrypting the *GBlocks* in a way such that their hosts cannot decrypt them. This will require some kind of secret key that the recipient must know. The method by which the receiver obtains the secret key is discussed in the next section.

In general, simple encryption of content with a secret key gives rise to another challenge. Suppose two parties insert the same file into the network, but under different keys. If one part of each version of the file is lost, neither of the files can be recovered with just one of the keys. This is true even if the parts lost in both files do not overlap and the file could otherwise have been reconstructed using both keys.

GNUnet encodes *GBlocks* in a special way that makes it possible to reassemble parts of two identical files that were inserted under different secret keys without anyone but the recipient even knowing that the *GBlocks* were related.

The key idea here is to use the hash of each *GBlock* as its key. More precisely, GNUnet encodes a 1k block B with the hash of B, $H(B)$. A RIPE-MD-160 hash provides 128 bits for a symmetric cipher, and another 32 bits for the initialization vector. Because the hashcode cannot be retrieved from the encrypted file $E_{H(B)}(B)$, intermediaries cannot decrypt the content. However, even with this scheme in place, two identical plaintexts that were inserted independently will have the same keys (except for the root node) and thus yield the same encrypted ciphertexts.

For the *RBlock*, a slightly different scheme is used to allow queries. Here, the user specifies the secret keys. These differing secret keys are used only to encrypt the *RBlock*s. Any of these keys can then be used to decrypt the entire file. A small problem arises from *GBlocks* that need to be padded to achieve the size of 1k. In order to avoid identical content being hashed to different values, only the actual data is hashed, not the random padding.

The encryption scheme for the *RBlock* is similar to the scheme used in *Freenet*; both use the hash of the keyword. However, our keywords are free-form and we only use them to encrypt the *RBlock*. The rest of the file is encrypted differently.

4.3 Queries

The scheme described so far leaves one question open: how to query for data. If the files were stored under the hash-codes of the original data, intermediaries could decrypt them. However, storing more than a single value per *DBlock* in the *IBlocks* would be inefficient.

In GNUnet, a block B is stored as $E_{H(B)}(B)$ under the name $H(H(B))$. Because H is a one-way function, the intermediaries cannot obtain the original hashcode from the filename. However, other nodes can compute $H(H(B))$ given $H(B)$.

The *RBlock* R of a file stores the hashcode $H(B)$ of the root *IBlock* B, which is then encrypted with $H(K)$ (where K is a user-supplied keyword). $E_{H(K)}(R)$ (the encrypted root node) is then stored under $H(H(K))$. When a user searches for K, his client will send a request for $H(H(K))$ and decrypt the returned result with $H(K)$. No other node in the network can obtain any information about the data transferred except by guessing the keyword K (or by knowing the exact file that is returned, which usually implies that this node originally inserted the file). Although a dictionary attack is thus possible, such an attack can be avoided by carefully choosing K.

This scheme has another advantage. It allows users to specify boolean queries of the form $a\,AND\,b$. GNUnet will then search for a and for b and return only those files that match both queries. Content providers can insert content under many keywords without using significantly more storage space, as only one extra *RBlock* per keyword is required. This allows users to search for content efficiently.

4.4 On-demand encoding

If the user intends to keep the file in plaintext on the local drive, the file can be indexed and encrypted on-demand. In that case, the insertion process will compute the hashes of each *GBlock* and store the hash together with the address of the *GBlock* in GNUnets database. The *IBlocks* and the *RBlock* are processed as usual. The size of the *IBlocks* and the hashes in the database is about 5% of the total filesize.

If GNUnet is searched for any of the *GBlocks*, it can locate the corresponding 1k block of the original file, compute its hash, encrypt it and send it out. GNUnet is not required to read any other GBlock from the original file.

4.5 Avoiding Content from Malicious Hosts

As mentioned before, the actual query for a datum matching Q is hidden by hashing Q first. As $H(Q)$ is used as the key for the decryption, $H(H(Q))$ is the obvious choice for the published query. However, this approach has a small problem.

If N matches the query Q, the encoded node $E_{H(Q)}(N)$ no longer has $H(Q)$. Thus, intermediaries that do not know $H(Q)$ cannot verify that this node matches the query at all. A malicious node could return random data to any query $H(H(Q))$, claiming that the extraneous data matches the query. The receiver would then have to propagate a message back through the query chain that the original sender was malicious. As intermediaries cannot keep track of earlier connections for a long time, this feedback may not reach the malicious node. Thus, a malicious node could actually improve its reputation by sending out meaningless data to the network. This would be unfortunate for reputation-based systems like GNUnet or Mojo Nation [7].

This problem is prevented in GNUnet by using $H(H(H(Q)))$ for the query. The sender must then provide $H(H(Q))$ to demonstrate that the sender actually has matching data. Because the sender cannot guess $H(H(Q))$, it can be assumed that the sender had matching content at some point. Of course, this cannot prevent malicious hosts from inserting bad data in general. Malicious nodes could *guess* that the keyword K is frequently used and just compute $H(H(K))$ and $H(H(H(K)))$, returning their useless data once a matching query comes by. However, this is similar to inserting extraneous data under that keyword into the network. Because no software can distinguish valuable content from worthless content in general, this is not a design flaw. Still, it demonstrates that the problem of content moderation is not entirely solved with this approach. The triple-hash scheme simply makes it harder to reply to arbitrary queries with random data; it cannot prevent users from inserting such data under popular keys.

5 Discussion

Splitting files into many small pieces might be perceived as having a negative effect on the preservation of data. If only one of the small pieces is lost, the larger file can no longer be reconstructed. Thus, many other systems strive to keep files intact. GNUnet guards against the loss of files by replicating the data. Small files (*GBlocks*) are quickly migrated from host to host; thus, they are cheaply replicated and thereby kept available.

The real issue with small blocks is the high number of queries that are required to find all the pieces in a distributed network. However, smaller blocks improve the load-balancing capabilities of the network [8, 7].

For GNUnet, we chose a very small granularity of 1k. While the encoding scheme described basically allows any granularity, this size is admittedly quite small. While this is an implementation detail, this allows GNUnet to duplicate blocks at a negligible cost per block. Duplicating a file with several megabytes of data in a single transaction, on the other hand, is both much more costly and more likely to fail.

Small blocks also allow GNUnet to operate with very limited memory, which may be of note for embedded devices. Additionally, any request in GNUnet can be answered using a single ethernet frame.

The encoding described above is open to two kinds of attacks. The most obvious attack is that of an adversary guessing the keyword and thus being able to decrypt the file. As users must have a way to decrypt the file solely by means of the keyword (which is also a guess), this cannot be avoided.

Another attack occurs when the attacker guesses the exact contents of the file (or at least the exact contents of an incriminating part). The attacker can then encode the file to see if the encrypted file matches the encoding that he is trying to break. This is possible because we wanted different keys to result in the same ciphertext. The attack requires a fair amount of knowledge on the part of the attacker. In fact, the attack is equivalent to one where an attacker inserts content into the network and waits for a another user to try retrieve it. This kind of attack is extremely hard to avoid, since GNUnet depends upon users providing content.

Both attacks fail to thwart the stated goal of the GNUnet encoding scheme, which is to prevent intermediaries from being able to exercise editorial control over content. While hosts can theoretically be forced to censor well-known keywords and files, they are unable to censor new keywords or unknown content. As small changes to the content or keyword are sufficient to make a detection impossible, censorship should be impractical.

The guessing attack described does provide the opportunity for limited forms of censorship. A server can create a "blacklist" of keywords or files and from these keywords (or the blocks of the files) compute the names of the corresponding blocks. Matching blocks could then be filtered out. It should be noted, however, that guessing the keyword or the exact contents of the file are the only way to distinguish blocks. In encrypted form, all types of blocks are indistinguishable.

6 Conclusion

We have described an encoding scheme that is able to produce the same ciphertext (with the exception of a 1k *RBlock*) for the encryption of a file X with two different keys K_a and K_b. We illustrated that this scheme allows boolean searches without exposing the keys. We have shown that the ciphertext is not significantly larger than the plaintext. Arbitrary pieces of the ciphertext can be generated from plaintext without reading unrelated parts of the plaintext. While dictionary attacks on the password and known plaintext attacks on the ciphertext are possible, they are harmless with respect to the goal of deniability.

In the future, remote searches in encrypted data deserve further attention. An interesting technique that allows word-based searching in a document without ever exposing the contents or the keyword is described in [9]. The technique is not vulnerable to guessing attacks but requires the data to be split at (key)word boundaries at encryption time. Like the approach described in this paper, [9] requires exact matches. Keyword extraction and approximate keyword matching are performed by modern search engines and should be a goal for secure searches in the future.

GNUnet is free software and part of the GNU project. The GNUnet code is approximately 19.000 lines of C code and is available on our webpage. The GNUnet server uses about about 2 MB of memory under GNU/Linux.

Acknowledgments The authors wish to thank the members of the Secure Software Systems Lab for helpful discussions, Jan Vitek for support and editing, Jens Palsberg for encouragement and Mikhail Atallah for supporting unusual ideas for class projects.

References

1. K. Bennett, C. Grothoff, T. Horozov, I. Patrascu, and T. Stef. Technical report.
2. I. Clarke. A distributed decentralised information storage and retrieval system, 1999.
3. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*. International Computer Science Institute, 2000.
4. Steven Hazel. Libfreenet, a case study in horrors incomprehensible to the mind of man, and other secure protocol design mistakes., Feb. 2002.
5. Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.
6. David Mazieres Marc Waldman. Tangler: A censorhip-resistant publishing system based on document entanglements. 2001.
7. Mojo Nation. Technology overview, Feb. 2000.
8. Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
9. Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
10. Stratton Oakmont vs Prodigy Services Company, 1995 N.Y. Misc. Lexis 229, (N.Y. Sup. Ct. Nassau Co., 1995).