An Encoding for Censorship-Resistant Sharing

Christian Grothoff[1], Krista Grothoff[2], Tzvetan Horozov[3], Jussi T. Lindgren[4]

[1] Department of Computer Science, University of Denver, USA
christian@grothoff.org
[2] CERIAS / Interdepartmental Program in Linguistics, Purdue University, USA
krista@grothoff.org
[3] Motorola Labs, USA
horozov@motorola.com
[4] Department of Computer Science, University of Helsinki, Finland
jtlindgr@cs.helsinki.fi

Abstract. This paper describes ECRS, a content encoding scheme for censorship-resistant peer-to-peer networking. The proposed encoding mechanism supports both efficient dissemination of encrypted data as well as encrypted queries over this data. Intermediaries can verify that an encrypted response matches an encrypted query without being able to decrypt either. Furthermore, ECRS allows users to share files encrypted under descriptive keys which are the basis for querying the network for content. With the proposed scheme, effective load balancing and swarm distribution are possible. The encoding mechanism handles data in small chunks, which can also be efficiently encoded on demand from the plaintext. The proposed encoding is conceptually simple and easy to implement.

1 Introduction

Internet censorship is a powerful weapon for oppressive governments, corporations and pressure groups in stifling dissent and suppressing dissemination of information. A variety of techniques can be used to censor online information; hard drives can be confiscated and searched, traffic can be filtered, and denial-of-service attacks can be performed against servers. In addition to these technical measures, individuals can be singled out and subjected to harassment. A common requirement for censorship in any form is the ability to *recognize* the objectionable information.

The goal of the encoding scheme presented in this paper is to make it difficult to identify information stored or transmitted in a open *decentralized* wide-area network where many *untrusted* peers want to share data. The proposed encoding scheme, called ECRS, allows both content and queries to be encrypted while being transmitted or stored on the network. This ensures that intermediaries remain oblivious to the data they are handling, reducing the chance that they can be singled out for an attack or held liable. It is important to note that the primary goal is *deniability* for the peers storing and transmitting the information. These peers must be able to claim that decrypting the information is infeasible

for them. ECRS does not attempt to protect against a censor who knows precisely which content to censor is able to match a request or response against that particular content. Using this kind of guessing attack a censor is able to invert one-way hash functions which are crucial for the security properties of the protocol. ECRS is supposed to protect intermediaries that merely transmit information and that need to be able to show that they are unable to recognize objectionable information for their own protection.[1]

The meaning of censorship resistance used in this work is a variant of the formal definition given by [23], in which censorship resistance is defined as the inability of a censor to restrict a targeted document while allowing at least one other document to be retrieved. In the context of [23], the adversary is granted extensive capabilities. In particular, the adversary is allowed to inspect every request or response, transcript the processing steps of all the peers and then decide whether to filter (drop) a given response. The primary difference in the adversary model for ECRS when compared to [23] is that the attacker is not allowed to be able to guess the exact content that is to be censored. This is an important restriction in practice. In the absence of such a restriction, [23] shows that censorship resistance requires a form of private information retrieval for which each operation has complexity $O(n)$, where n is the size of the datastore. In return for restricting the guessing capabilities of the adversary, ECRS can achieve $O(1)$ complexity for its operations. Considering that scalability is also an important goal, in particular since censorship-resistance for small collections would be of limited utility, we believe that this is a reasonable trade-off.

As a content encoding mechanism, ECRS could be integrated into almost any generic peer-to-peer infrastructure to enhance its resistance to censorship. Specific details on how peers communicate or route data – even though important for the availability of the service and the privacy of its users – are therefore orthogonal issues outside of the scope of this paper.

As an encoding scheme, ECRS has to address two important issues. First of all, users must be able to find content, preferably by searching with simple, natural language keywords. Metadata related to the contents of the file should be made available as part of the search result and prior to the download of the actual file. Metadata can include anything from user-provided content descriptions to thumbnails (for images) and automatically detected mime-types. Ensuring that the metadata is an accurate description of the file is outside of the scope of this paper; recent work on reputation systems [31] provides a partial solution to this problem. Secondly, the encoding scheme should support swarming – that is, the download of a file from multiple peers. Since it is possible that identical data could be inserted by many independent parties, the encoding scheme should

---

[1] If the data cannot be identified, editorial control is technically impossible. In *Stratton Oakmont v. Prodigy*, an Internet provider was found liable for hosting a bulletin board on which offensive messages had been posted. This finding resulted 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 potentially not have been held accountable [28].

always result in identical encodings for identical source files. This special form of encryption was probably first described in [5] and later termed *convergent encryption* [9]. With convergent encryption, the existence of multiple copies of the same content can be detected; this knowledge can then be used to increase fault-tolerance, performance and availability.

Unfortunately, it is not enough to just meet requirements related to efficient storage and retrieval of the content. A public filesharing network should be open to new participants without any restrictive form of access control. This opens the network to malicious participants. Thus, the encoding scheme should guarantee the *integrity* of the content, even in the presence of malicious participants. The encoding mechanism must ensure that invalid replies are detected and that peers can distinguish valuable content from maliciously generated noise. This is particularly important in three cases. First, if large files are assembled using blocks from multiple peers, some of these peers may be malicious and try to corrupt the file. These corruptions should be detected on a small granularity level and repaired with correct data from other peers. Second, for searches it should be difficult for peers to spam the result set; otherwise, the users may drown in useless information. Finally, intermediaries that are merely routing the content should be able to detect invalid replies. This enables the existence of local economies (such as those in [12]) and prevents the retransmission and replication of corrupted data by non-malicious peers.

This paper introduces the primitives that ECRS uses to achieve these goals: a specific altered variant of the Content Hash Key (CHK), a primitive used in [5], and *KBlocks*, a novel construction which adds functionality to the ECRS scheme while avoiding a proposed attack on the triple-hash primitive used in [2]. These primitives allow data to be inserted under multiple natural-language keywords at minimal cost in space, and queried using natural language boolean searches. Furthermore, the primitives give ECRS the means to identify, encrypt, decrypt and verify data, all while avoiding attacks by a non-guessing intermediate adversary. Verification of content by intermediate hosts is made possible without revealing the keys necessary to decrypt content, and the amount of data corruption an adversary can inject before being detected is limited to a small size. Additionally, the construction of these primitives makes ECRS quite scalable, and assists the encoding scheme in encouraging load-balancing, distribution and replication of content, and swarming. These primitives (and their functionality) are described in greater detail later in the paper.

The rest of the paper is organized as follows. Section 2 contrasts the current work with mechanisms used in other censorship-resistant systems. Section 3 describes the ECRS encoding scheme in detail. Section 4 lists some extensions to the basic scheme. Finally, Section 5 discusses attacks on ECRS and contrasts it with previous work.

This work builds on our earlier paper [2] which was first published at ACISP 2001. Since then, the encoding scheme has been revised to address additional requirements an to improve the security in various aspects. This paper reflects these changes.

2   Related Work

A large number of censorship-resistant (or at least fault-tolerant) publishing systems have been proposed in recent years [1,4,5,8,10,15,29,30]. The proposed protocols differ in various aspects unrelated to the specifics of the encoding mechanism, such as content replication and the use of anonymous or steganographic communication. Characteristic of all of these systems is the use of replication and the need to ensure data integrity. For an overview of content replication strategies see [6]. The censorship-resistant systems all use some form of content encoding to hide the specifics of the content from the servers.

2.1   Freenet

The main content encoding scheme used in Freenet [5] uses CHKs: the hash of the content is used as the key. Freenet has several other types of keys which are used to refer the user to content-hash keys. These different key types are used to allow additional features such as content signing, personal namespaces and splitting of content. Freenet's CHK yields the same encrypted data for independent encryptions. However, it has several disadvantages.

One drawback of Freenet's CHK is that it does not allow direct sharing of files from the local drive without encrypting and inserting them first. Thus, if a node operator wishes to keep content directly accessible on the host, a local copy of the plaintext must be kept in addition to the encrypted content in the Freenet database. Also, Freenet does not support swarming. One reason for this is that Freenet's CHK encoding can only verify a file in its entirety and not pin-point corruptions on a smaller scale.

One way to obtain the CHK key of a file in Freenet is to search a private subspace. A subspace contains a set of CHK keys that have been cryptographically signed by the owner of the subspace and placed into the subspace under a certain keyword. These signed references are called subspace-signed keys (SSK). Given the subspace-prefix and the keyword, users can obtain the CHK key. The form of the keywords can be freely chosen, allowing for conventions such as dates in the keyword to be used for updates (in Freenet's terminology, these are called *date based redirects*).

In addition to CHK and SSK keys, Freenet supports KSK keys. A KSK key is essentially a CHK key stored in plaintext (!) under a keyword. The protocol attempts to ensure that there cannot be two different CHK keys available in the network under the same keyword. While collisions for KSK keys are supposedly not allowed, the protocol cannot prevent them from occurring. If an attacker inserts meaningless data under a common keyword, Freenet will try to preserve that content because it is frequently requested. Subsequently, that keyword can no longer be used for any useful data since the peers attempt to enforce the no-collisions rule. Note that the no-collision rule cannot be enforced for malicious peers. For these reasons, the Freenet developers consider the use of KSK keys insecure and recommend against using them.

As a result, Freenet effectively does not support any type of keyword-based search. The CHK keys must be found by using keyservers or ad-hoc applications built upon the basic solution by using incremental numbers as parts of keywords in signed namespaces. The disadvantage of keyservers is that they must be maintained if the indexed keys are to be kept up-to-date. Freenet does not seem to use any encryption for the data stored in KSK or SKS keys and therefore fails to prevent intermediaries from learning the CHK keys, making censorship by intermediaries easier.

2.2 Free Haven

In Free Haven [8], 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 [24]. 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 design of Free Haven itself does not discuss any search functionality. The user then forwards the key to the server which encrypts and sends its share of the document.

Shares are encoded with an information dispersal algorithm (IDA). Hosts are still able to exercise editorial control over the content since the IDA does not prevent hosts from decoding parts of the plaintext. Also, the k out of n share reconstruction scheme does not allow reconstruction of a file by mixing shares from independent insertions with different values for k or n.

2.3 Mnet and Mojo Nation

Mnet is the successor of Mojo Nation [32], a peer-to-peer network that was the first to focus on using economic incentives when trading content. As the descendant of a commercial product, Mnet focuses mostly on efficiency and reliability. Nevertheless, Mnet encrypts files with AES using a user-supplied key to allow for privacy. The file is then broken into shares using an erasure code. Multiple shares are identified with an SHA-1 hash and stored in an inode. The file can be retrieved using the hash of the inode and the symmetric encryption key.

The use of whole-file AES encryption and the use of an erasure code forces Mnet, as with Freenet, to store a second copy of the entire file in the Mnet data store even if the user retains a copy of the file in plaintext. The encoding makes it infeasible to encode parts of the file on request. The free selection of the AES key allows independent users to create completely different shares for identical files. Since the encryption of the content is a perfectly independent pass over the file, other encodings such as ECRS and Freenet's encoding could be easily extended to include this behavior if desired – without the need to mandate a certain style of AES encryption.

2.4 Publius

Publius [30] is a static network that uses Shamir's algorithm [26] to split the key required to restore stored files between the available servers. The encrypted file

is then stored on all servers. The servers cannot decrypt the file as the key is never 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. Publius never discards any content and thus assumes that disk space is never a limitation.

2.5   Tangler

The Tangler network [29] is a system where new content is entangled with old content. This entanglement introduces a dependency which makes it impossible to retrieve a piece of 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 *doubles* the amount of data that must be transported by the network. This scheme fails to address the fact that an adversary may not care about losing other data when attempting to censor content. Also, nothing prevents the adversary from first making a copy of important documents and re-inserting them into the system after the censorship operation.

Tangler has some similarities with ECRS. In particular, it uses hash trees for integrity checks on fixed-size blocks of data. However, unlike ECRS, the Tangler encoding uses Shamir's secret sharing [26] to entangle the block with other, pre-existing blocks, preferably from other documents. Both Tangler and ECRS (as well as Freenet) use cryptographically signed data, which Tangler calls collections. ECRS has analogous constructions to Tangler's collections, referred to in this paper as *directories* and *namespaces*; however, unlike ECRS, a Tangler collection has a versioned root which explicitly lists all of the contents in the collection, resulting in one of Tangler's global synchronization problems. Placing a document in a Freenet subspace or ECRS namespace only requires that it be adequately signed.

2.6   Gnutella and FastTrack

The FastTrack network[2] and modern variants of the Gnutella protocol use the SHA-1 hash code of the shared files to uniquely identify files. The SHA-1 hash codes are used to identify identical files stored on different peers. The peers can then use this information to download the file from multiple sources (swarming). The problem with this approach is that the SHA-1 hash codes can only be verified once the entire file has been retrieved. To ruin a download, it suffices for a malicious peer to contribute just one single invalid block. The presence of an invalid block could only be detected after the entire file was retrieved. Worse, the

---

[2] Protocol description from the giFT-FastTrack project is available at

http://gnunet.org/papers/FAST-TRACK-PROTOCOL.

simplistic SHA-1 approach does not give peers any way to identify the location of the invalid block or the malicious peer.

Neither FastTrack nor Gnutella use file encryption. Consequently, they do not provide any protection from censorship beyond decentralized routing of requests.

2.7   Distributed File Systems

Various distributed file systems based on a peer-to-peer infrastructure have been built in recent years [7,18,19,22,25]. These systems typically do not focus on censorship resistance and consequently do not encrypt the contents. Modern systems like [7,25] route content using distributed hash tables [17,27] to achieve $O(\log n)$ scalability. Writable distributed file systems typically require users to authenticate themselves before being granted access to the system by one of the servers [18]. ECRS focuses on encrypting shared data while allowing peers to verify the integrity of the encrypted data that they are routing. In SUNDR [19] a similar problem is addressed; here users can detect tampering by the (untrusted) server that is storing the data in a setting where multiple users are updating the same data. SUNDR achieves what is called *fork consistency*, a guarantee that if the server passes a version that lacks the changes of one client to another client at one point in time, it can never again make the changes of either client visible to the others. This is a weaker guarantee than the optimal situation where any misbehavior is detected; however, it is the strongest possible property that can be achieved given the security model used. ECRS does not allow users with distinct pseudonyms to update the same content without essentially creating two files. This allows ECRS to guarantee that peers can never return an *invalid* result since every copy can be uniquely identified at all times.

3   ECRS encoding

ECRS uses two fundamental primitives in the overall encoding scheme. The majority of the data is encrypted with a variant of Freenet's CHK encoding [5]. The fundamental idea behind CHK is that content is encrypted with a key that is derived from a cryptographic hash of the plaintext. The encrypted content is hashed again to obtain the query hash which is used to request the content. This key-query pair is what the receiver needs to download and decrypt the content.

The primary difference between the use of CHK in ECRS and that used in [5] is that ECRS does not subject entire files to the CHK encoding. Instead, files are split into fixed-size blocks which could then be individually distributed (and replicated) across peers in the network; this is a natural way to balance the load and to enable swarming. These fixed-sized blocks can be individually identified, encrypted, decrypted and verified. Each block is identified by one or two hash codes, and the blocks are encrypted using a key that is independent of the hashes used for identification. Verification only requires the encrypted block and the query; it does not reveal the key required to decrypt the block. ECRS guarantees that it is hard to forge a response that passes verification for a given
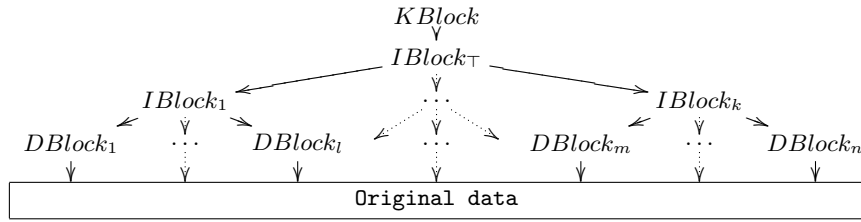
Fig. 1. Simple encoding of a file using a tree of CHK blocks (*DBlock*s and *IBlock*s are CHK encoded) with one *KBlock* to allow for a keyword-based search.

query, and splitting files into blocks allows ECRS to detect data corruption on the block level. Malicious peers cannot transmit more than the selected block size of invalid data before being detected.

The second primitive used by ECRS, the *KBlock*, is completely novel. *KBlock*s allow users to securely obtain the key-query pair by entering a natural language keyword selected by the content provider. The primitive used to achieve this in [2], the triple hash, is improved on in this work in that *KBlock*s *cannot* be attacked successfully by a non-guessing intermediary. The triple hash scheme was vulnerable to possibile substitution attacks by intermediaries. *KBlock*s defeat this attack by signing content advertisements using deterministically generated public keys. Additionally, by using *KBlock*s, documents can be inserted under multiple keys at a minimal cost in terms of space. Users can perform boolean searches of the form "a AND b" which then return the documents which were inserted under a matching set of keywords. Before the download, the user can refine the search results using the metadata associated with the file.

The rest of this Section gives a bottom-up description of ECRS. After a brief overview, Part 3.2 shows how individual data blocks are encrypted in order to make it impossible for the intermediaries to determine what they are storing or transmitting. Part 3.3 then describes how files are split into those data blocks. Parts 3.4 through 3.6 motivate and explain how ECRS enables users to search for content without exposing the plaintext of the keyword or the corresponding content and how intermediaries can verify that a given encrypted reply matches the (encrypted) query. Finally, the core of ECRS is summarized in the form of the encoding algorithm and a protocol example.

3.1 Overview

In ECRS, the encoded representation of a file is a collection of blocks. There are four types of blocks: *DBlocks*, *IBlocks*, *SBlocks* and *KBlocks*. The maximum size of all blocks in ECRS is supposed to be fixed as some small number (e.g. 32Kb). The blocks are used to create a tree structure of the encoded file (see Figure 1). The actual file data is encoded in *DBlocks* which are organized by a tree of

IBlocks. The *IBlocks*, which are similar to UNIX inodes [20], are indirection blocks which encode how the leaf nodes of the tree can be reassembled into the original file. Every *IBlock* provides the data required to request, decrypt and verify the contents of its direct children. Verification is done using hash codes of the children, similar to the way a Merkle tree [21] works. Given the size of the file, the top *IBlock* uniquely identifies the contents of the entire file. *KBlocks* and *SBlocks* facilitate obtaining the key hash (to decrypt) and query hash (to request) for the top *IBlock*.

3.2   *DBlocks*

Every *DBlock* corresponds to 32Kb of the original file (the last block may be padded with zeros if the file end does not fall on a 32Kb boundary). *DBlock*s are encrypted using a symmetric cipher. For the *DBlock* that corresponds to a plaintext block B_i, the key for the cipher is $K_i := H(B_i)$. This guarantees that the same block will always be encrypted with the same key. Also, the integrity of the decrypted block can be verified by checking that $K_i = H(B_i)$. The encrypted block $E_{K_i}(B_i)$ can be uniquely identified by its hash $Q_i := H(E_{K_i}(B_i))$. A peer can query for $E_{K_i}(B_i)$ using the query hash Q_i without revealing K_i. Consequently, the pair (K_i, Q_i) can be used to retrieve and decrypt B_i. Intermediaries can verify that a response $E_{K_i}(B_i)$ matches a query hash Q_i without knowing K_i. Freenet [5] uses the same scheme to encrypt entire files (not just blocks) and the authors call the pair (K_i, Q_i) a content hash key (CHK).

3.3   *IBlocks*

For files larger than the blocksize, multiple CHK keys are required to retrieve the content. For example, with a blocksize of 32Kb and 512-bit hash codes, a 32Mb file would be split into 1024 *DBlocks*, $B_1, \ldots, B_{1024}$, of length 32Kb each. Groups of up to 256 CHKs are assembled into one 32Kb *IBlock*, resulting in four *IBlock*s for the 32Mb file. The resulting *IBlock*s are CHK encrypted in the exact same manner as the *DBlocks* and their CHK-keys are again grouped into *IBlock*s until only one CHK key is left. For the 32Mb file, the CHKs for the four *IBlock*s are thus collected in one final top *IBlock*. The additional space requirements for encoding the file are typically less than 1% of the original file size.

3.4   Finding files

The CHK encoding described so far is useful for transmitting content that is encoded and yet verifiable for intermediaries. However, a practical file-sharing scheme needs to provide functionality such that the users can obtain the query hash and the key that are now necessary in order to retrieve and decrypt the encoded content. While out-of-band communication of those CHK keys is certainly feasible, an integrated solution is clearly desirable. Related work on secure indexing [11] allows servers to retrieve documents by keyword without learning

about the rest of the document. The scheme that will be described in Section 3.6 improves on this by keeping the keyword itself private. Additionally, it still allows intermediaries to verify the correct operation of the server. It also avoids the need for the publisher to encrypt the document with the public key of a *specific* receiver, thereby theoretically permitting arbitrary participants to access same encrypted content (as long as they are able to obtain or guess the necessary keyword).

The key problem for the search is that simply publishing CHK keys in plaintext under certain keywords (as done in Freenet with KSK keys) has various drawbacks. First, intermediaries would learn about the CHK keys that they are storing or transmitting, which could allow them to easily censor search results. Second, the keywords are directly exposed to the intermediaries, which again would facilitate censorship. Finally, peers would be able to send invalid replies that refer to inappropriate content. This may be just a nuisance if malicious peers use this for advertising (a kind of behavior that can be observed on gnutella from time to time). Possibly worse, if a network wants to economically reward peers for returning search results, the possibility of returning arbitrary nonsense might make such incentive schemes impractical.

But how can a system prevent peers from returning invalid responses to a keyword-based search? Since it is inherent in the nature of such a search that the initiator does not know precisely what content is desired, it is impossible for the query to completely determine the reply. For this reason, schemes like CHK cannot be applied to the keyword-search problem: in CHK, the query hash completely determines the reply. As a result, a keyword-based search mechanism where a query should possibly have multiple valid results cannot be implemented based on CHK. Since for this kind of search there cannot be any completely deterministic link between the keyword and the content, ECRS must trust the user who is uploading the content to provide a valid link. That user must specify appropriate keywords (and metadata) that properly describes the content. Clearly this trust may be misplaced. The uploader may specify keywords that have no semantic relation with the content. One solution to this problem is the use of namespaces, which attack the issue by enabling users to learn which uploaders are trustworthy.

3.5   Namespaces

One way to avoid searching in an untrustworthy global keyword space that may contain mostly inappropriately-labeled content is to restrict the search to namespaces. A namespace is a keyword space that contains content that is cryptographically signed with the same private key. The concept of namespaces is also present in Freenet [5], but with Freenet it is not possible to obtain multiple results by searching a namespace. Fortunately, it is easy to add *searchable* namespaces to ECRS.

In ECRS, a namespace is identified by a public-private key pair (also referred to as the pseudonym). The private key is used to sign the content that is placed into the namespace, and the public key (or more precisely, the hash of the public

key) is used as a public handle that is used to refer to the namespace. Any user can create any number of pseudonyms. Once the user accesses files from a particular namespace it is possible to form an opinion about a particular pseudonym, which can then be used to target the search.

Adding namespaces to the design requires a new type of block, called an *SBlock*. An *SBlock* contains the key and query hash (CHK) of the top *IBlock* together with the metadata belonging to the content, thereby allowing users that receive an *SBlock* to download the file. However, this data is encrypted such that intermediaries do not learn anything about the referenced file. The encryption key is the hash of an identifier that the inserter chose for the content. The identifier does not have to be unique and can, for example, be a keyword (to allow searching the namespace) or a password (for private content). The encrypted portion of the *SBlock* is followed by a hash code which is the *query* identifier for the *SBlock*. The query identifier is the hash of the key (which makes it the hash of the hash of the identifier). When a user searches a namespace, the request contains the hash of the public key of the pseudonym together with the query identifier. By using the hash of the key as the query identifier, the correct *SBlock* can be identified by intermediaries but not decrypted. At the same time, only the identifier and the hash of the public key of the pseudonym are required to construct both the query identifier needed to search the namespace and the key needed to decrypt the *SBlock*. Finally, *SBlock*s contain digital signature of the other contents. This allows intermediaries to verify that a given *SBlock* was created by the owner of the namespace.

Since *SBlock*s are signed, it is possible to allow updates. Without a digital signature, a secure update mechanism is impossible, since anyone could publish a potentially illegitimate update. If the update requires a matching digital signature, forged updates are impossible. In order to allow content updates, the metadata of an *SBlock* can contain information about when and under which identifiers updates may be published. If such forward pointers are present, the owner of the namespace can decide to publish updates to a particular file at a later time. Since the private key used for signing is owned by a single entity, consistency issues from parallel updates do not arise. ECRS does not prevent the owner of the namespace from publishing conflicting updates.

It should be recognized that the same file can be part of multiple namespaces under multiple identifiers without the need for storing the file multiple times in the network. The *IBlocks* and *DBlocks* of the file do not need to be changed when the namespace or namespace identifier changes.

Namespaces cannot completely solve the search problem. Most importantly, users would still have to find the namespaces themselves, and while it can be assumed that users can guess keywords, they are unlikely to be able to guess public keys for a namespace search. ECRS alleviates this problem by also offering a pure keyword-based search in a *global* keyword space where any user can advertise content. Here, the disadvantage is clearly that it is possible for malicious users to pollute the global keyword space with advertisements and other inappropriate data. However, it is assumed that in that case users will use

keywords that are more difficult to guess or only use the global keyword space
to discover trustworthy namespaces, thus avoiding spam most of the time.

3.6   Keyword search

This section describes how information about namespaces or the CHK keys
that are necessary to retrieve a file can be obtained by searching using natural
language *keywords* – without revealing the keywords to intermediaries. Searching
requires the content provider to supply metadata about the file; in particular,
the content provider must supply a list of keywords. While the metadata is
published as part of the *KBlock* and made available to the recipient the keywords
are only used to compute the *KBlock*s and not made available to other users
directly. It is assumed that the user searching for data is able to *guess* at least
one of these keywords. The nature of the search implies that there is no direct,
cryptographically verifiable relation between the keyword and the content. Many
files can match a given keyword. Different users may sometimes have a different
perception of what is a correct semantic match. Thus, a keyword search always
has the potential of resulting in search results that a user may find inappropriate.
It is also not possible to prevent a user from supplying keywords that have no
semantic relation with the content and thereby increasing the chance of other
users finding undesirable results for those keywords.

Despite these fundamental drawbacks, ECRS should (and can) give some
unusual *guarantees* for keyword-based searches. First, the keyword used for a
search is not exposed to intermediaries in a way that would allow them to eas-
ily exercise editorial control (i.e. to censor queries). Note again that a guessing
attack in which the intermediary guesses a keyword, computes the respective
query hash, and attempts to match the query hash to the incoming query, is
considered acceptable; plausible deniability for the intermediaries is unaffected
by this attack. Secondly, only peers that have content available under the given
keyword should be able to produce a valid response. In other words, the ulti-
mate responder must have succeeded in guessing the keyword. This is important
since it prevents peers from making up replies for arbitrary requests on-the-fly. If
peers were able to respond to any keyword query without having to at least guess
the keyword, they could use this to spam every possible search, rendering the
keyword search mostly useless. Forcing responders to guess keywords does not
eliminate the problem completely; however, it does limit the attack to the com-
mon, easily-guessed keywords. Finally, the verification that a response is valid
should be possible not only for the final recipient, but also for intermediaries.

ECRS achieves all of this with a *new cryptographic primitive* called K-
deterministic keys which is a key pair $(Pub_{H(K)}, Prv_{H(K)})$ generated from a
keyword K. The hash $H(K)$ of the keyword is used as the seed and the only
input to the pseudo-random number generator that generates the public-private
key pair. The authors are not aware of any previous use of K-deterministic keys.
Given such a key pair, the inserting peer generates *KBlock*s by signing (with
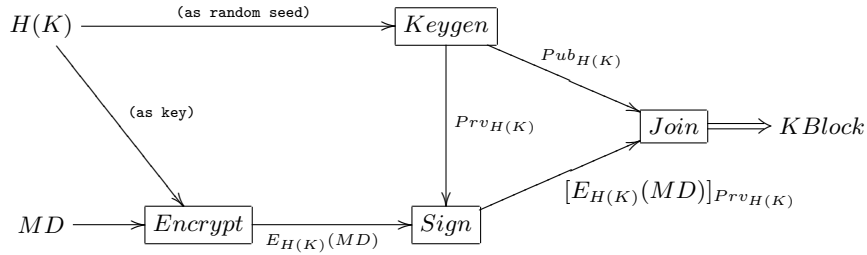$Prv_{H(K)}$) the metadata necessary to download the file. *KBlock*s are essentially

Fig. 2. The construction of a *KBlock*. The hash $H(K)$ of the keyword K is used to encrypt the metadata (MD) and to seed the key generator. The obtained private key $Prv_{H(K)}$ signs the encrypted metadata ($E_{H(K)}(MD)$), but is not otherwise used. Finally, the public key $Pub_{H(K)}$ is appended to the signed data to form the *KBlock*.

like *SBlock*s with the difference that the K-determinate key is used instead of the pseudonym. The construction of a *KBlock* is shown in Figure 2.

A peer searching for K also computes $Pub_{H(K)}$ using the same deterministic key-generation algorithm ($Prv_{H(K)}$ is also computed but not used). The peer then uses $H(Pub_{H(K)})$ as the query hash. Intermediaries can verify that the response is valid by checking the *KBlock*s signature against its public key, and checking that the public key hashes to the query hash. Yet intermediaries are unable to decrypt the response or learn the keyword K that was used to generate the public key without guessing.

In conclusion, using *KBlock*s it is possible to allow intermediaries to verify the integrity of replies without learning the plaintext of the keyword or the response. *KBlock*s require a keyword guess, making it it harder to reply to arbitrary queries with random data; they cannot prevent users from inserting such data under popular keywords. The receiver of the *KBlock* can retrieve the rest of the file tree using CHK queries in the same way as it is done for *SBlock*s.

3.7   Summary

The ECRS encoding algorithm is summarized in Figure 3. Figure 4 shows the sequence of messages exchanged by peers to download a 64Kb file starting with a keyword.

4   Extensions

The vanilla ECRS encoding scheme still leaves some practical problems that should be addressed to facilitate real-life usability. For example, the manual effort of assigning or guessing keywords should be minimized. Also, the overall system should be efficient in practice. In particular, the system requires peers

1. Input: Content C, a list of keywords $\{K_1, ...\}$ and metadata D.
2. Split content C into blocks B_i, each of size 32Kb. Call these *DBlocks*.
3. Compute hash values $H_i^{key} = H(B_i)$, $E_i = E_{H_{key}}(B_i)$ and $H_i^{query} = H(E_i)$. Form CHK_i from (H_i^{query}, H_i^{key}). Pad content (with zeros) if needed.
4. Store E_i under the name H_i^{query}.
5. If there is more than one block, group the blocks into disjoint sets I_j of up to 256 members. For each I_j, create *Iblock$_j$* containing the *CHK*s of the blocks in I_j. Add zero padding if needed. Process each new 32Kb block obtained as in step 3.
6. If the size of C was smaller than 32Kb (and thus there is only one pair of hashcodes CHK_0), create *SBlock*s and *KBlock*s containing CHK_0, the metadata D about the content and the original length of C using keywords K_i according to Figure 2. Encrypt and sign the resulting *SBlock*s and *KBlock*s.

Fig. 3. The ECRS encoding algorithm.

B shares file which is encoded as B_1, B_2 and one *IBlock* under keyword "test".

$A \rightarrow B$: Query: $H(Pub_{H(\text{"test"})})$
$B \rightarrow A$: Response: $([E_{H(\text{"test"})}(Q_0, K_0, size)]_{Prv_{H(\text{"test"})}}, Pub_{H(\text{"test"})})$ (one *KBlock*)
$A \rightarrow B$: Query: Q_0 (CHK-query)
$B \rightarrow A$: Response: $E_{K_0}(Q_1, K_1, Q_2, K_2)$ (encrypted *IBlock*)
$A \rightarrow B$: Query: Q_1 and Q_2 (CHK-query)
$B \rightarrow A$: Response: $E_{K_1}(B_1)$ and $E_{K_2}(B_2)$ (encrypted *DBlock*s)

Fig. 4. Example protocol for an ECRS download. (Q_i, K_i) is a CHK pair to request and decrypt block i.

to frequently handle small, per-block queries, many of which may not have an answer that is locally available. The following subsections present some ideas on how these issues can be addressed.

4.1   Keyword inference

One problem with ECRS is that the content must often be associated with keywords. Although this is vastly better than forcing the user to rely on "meaningless" bitstrings as the only identifier, the keys must still be chosen so that the target audience can guess them. The simplest non-technical solution for this problem is for a group of users to agree on an informal standard for keywords. If content providers want to make it easy for others to guess a keyword, it is typically useful to follow informal conventions for keyword selection. For example, a reasonable convention may be to only use lower-case letters.

A larger problem for a keyword-based search is that users may be reluctant to put in the effort to provide good keywords for the content that they provide. A solution to this problem is the `libextractor` library [13]. `libextractor` is a
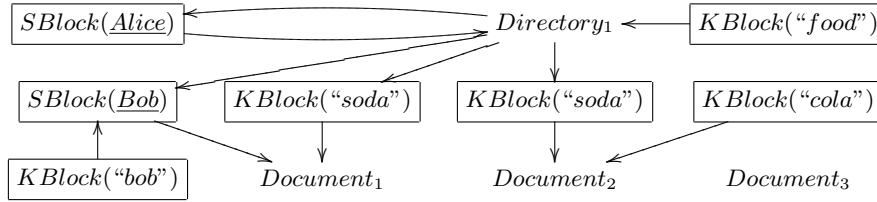
Fig. 5. ECRS allows the formation of complicated network structures. In this example, the *SBlock* of pseudonym *Alice* eventually allows to retrieve two of the documents encoded in the system, as does *Kblock* for keyword "food". Boxes denote simple blocks that serve as entry-points. The directory and the documents consist of trees of possibly many CHK-encoded blocks. Searching for keyword "soda" would return two results pointing to different resources. Note that all the encoded resources, including the one document with no pointers to it, can be retrieved by knowing the CHK keys of their root *IBlock*s.

library for metadata extraction that is used to automatically extract keywords and other metadata, such as descriptions, comments and mime-types from files of various formats.

4.2   Directories

Another approach to avoiding searches is to group files into directories. Directories can help with the search problem since they reduce the need for searching. Also, they can be used to bundle similar content and thus help users avoid inappropriately labeled data. If several files of such a directory have appropriate descriptions, it is likely that the other files will also have been provided by a non-malicious party.

Integrating directories into the ECRS scheme is fairly straightforward. A directory is simply a file that contains any number of *SBlock*s or *KBlock*s in plaintext (!). Like files, directories are immutable and are subjected to the same encoding for sharing.

Note that it is possible for users to assemble a directory that refers to any combination of files: files the user inserted personally, files that the user downloaded, files that the user encountered as search results but never downloaded, or even invalid data (which would be interpreted as references that correspond to non-existent files). Directories can also refer to other directories, allowing users to build navigable structures similar to file systems or the World Wide Web. In particular it is possible to refer to contents of other users and even to files or directories that are supposed to be published in the future. Figure 5 illustrates some of the possible reference relationships between the various constructs.

4.3   Efficient lookups with bloom filters

Given that a query only requires the transmission of a hash code, the number of queries that any given peer may receive is quite high. The bandwidth required for the response can often be discounted, since a peer may only be able to respond to a fraction of the received queries. Assuming that queries dominate the traffic, over 50 queries per second could be transmitted over a slow modem line. Many peers might not be able to perform disk-based database lookups at such rates. Worse, if peers are sharing gigabytes of content, simply keeping the index information in memory is also often no longer feasible.

In order to dramatically reduce the necessary number of database accesses, ECRS can be supplemented with a bloom filter [3], which is a probabilistic data structure that uses little memory and can be used to filter out queries that definitely cannot be answered by the local peer. The bloom filter is guaranteed to allow all queries that can be locally answered through the filter, and will reject all but a statistically insignificant number of queries that cannot be answered locally.

The size of the bloom filter in memory that is required to reduce the number of false-positives to approximately 3% is about one byte (8 bit with 5 bloom tests, $2^{-5} \approx 3\%$) per entry. This reduces the memory requirements for the index by a factor of 16-64, depending on the size of the hash. Since a node occasionally needs to remove content from the local database, additional bit counters for the bloom filter should be stored on the disk. The assumption is that changes to the filter are rare compared to the frequency of queries. Queries that pass through the bloom filter are forwarded to the database layer which then attempts to locate the content.

5   Discussion

The fact that each *DBlock* can be encoded independently from any other *DBlock* results in an important feature of ECRS called *on-demand encoding*. Given a plaintext file and an index that maps a query hash to an offset in the file, the resulting *DBlock* can be encoded on-the-fly, without processing any other part of the file. This can be an important benefit if the user intends to keep the file in plaintext on the local drive. In this case, the system can index the *DBlocks* without replicating the actual data completely, storing only the resulting *IBlocks* (and *SBlocks* and *KBlocks*) and then later producing the *DBlocks* only using the index and the original file. This can result in rather dramatic savings in space (no replication of encoded content).

Splitting files into blocks might be perceived as having a negative effect on the preservation of data. If only one of the blocks is lost, the larger file can no longer be reconstructed. Many other systems [8,32] attempt to address this problem by either attempting to keep files intact, or by using erasure codes [24] to patch missing pieces. While erasure codes could be used together with ECRS, they would conflict with the performance benefits of on-demand encoding. Consequently, the primary mechanism to guard against the loss of files encoded with

ECRS should be data replication. Since ECRS splits files into blocks it should be possible for the network to duplicate blocks at a negligible cost per block. Duplicating a complete file of several gigabytes would be much more costly and more likely to fail. For large files, it may be difficult for a peer to find individual hosts that are willing (or able) to provide enough space to replicate the complete file in its entirety. Also, the traffic burst that a large transfer would impose on the network might keep the participating nodes busy for too long. Blocks, on the other hand, can migrate quickly from host to host. Storing a large file on a single host also fails to facilitate distribution of load for downloading purposes.

5.1   Attacks

ECRS is open to two types of attacks from censors. The most obvious attack is to guess keywords and use those to compute queries for specific *KBlock*s. The adversary can then try to censor these queries. Since the same content can be available under many keywords, this technique is not necessary successful. However, this guessing attack does provide the opportunity for limited forms of censorship. An adversary could create a "blacklist" of keywords and attempt to legally force operators to filter matching queries or replies.

   Instead of predicting keywords, the attacker can also try to obtain the exact contents of the file (or at least the exact contents of an incriminating part). This maybe achieved by searching the network for certain keywords and then downloading the resulting content. This way the attacker can again compute the query hash for the *DBlock*s and *IBlock*s and also obtain the encrypted *DBlock*s and *IBlock*s themselves. Again, the adversary can then try to censor this data. Note that this attack requires the adversary to obtain a copy of the content and then exercise a certain amount of control over the peers in the network. As users must have a way to decrypt the file solely by means of guessing the keyword, this attack cannot be avoided. A fault-tolerant peer-to-peer routing protocol may be able to route the content around the censoring peers.

   If such an active censorship attack is known to occur on a large scale, a possible defense would be to publish the content with slight alterations for each keyword. A single space inserted at the beginning of a text-document is sufficient to change *all* blocks of the ECRS encoding.[3] While this would defeat the use of convergent encryption and increase the storage and bandwidth requirements, this would make it infeasible for the censor to effectively censor all of the variants. Since such a powerful active attacker is likely to be uncommon, the *default* in ECRS is to use convergent encryption.

   Both attacks fail to enable intermediaries to exercise editorial control over content in general. While peers 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 detection impossible, these types of censorship should be impractical.

---

[3] Note that only operations causing global shifts in the data have this property. One single-character replacement changes only one root-to-leaf path in the encoding tree.

Another serious type of attack is spamming the global keyword space with useless content. Popular keywords are an easy target since they are by nature easily guessed. Namespaces can theoretically eliminate this problem, however they create a new difficulty in assessing which pseudonyms represent users that provide valuable content. A secure reputation system, like Advogato [16], may be a step towards a technical solution for this problem. A possibly more important challenge for future work is making such complex solutions easily accessible for the average user.

5.2   *KBlocks* or triple-hash

The triple-hash that was used instead of *KBlock*s in our earlier work [2] allows for an attack by an active, participating adversary. The following paragraphs first briefly present the original scheme and discuss the attack, and then compare it with the *KBlock* approach taken in this paper.

Let an *RBlock* be a new kind of block that also contains the metadata of a file (including the top CHK data). The only difference between *RBlock*s and *KBlock*s is that *RBlock*s are encrypted and requested using the following triple-hash scheme. For each keyword k, the *RBlock* R is encrypted with $H(k)$ and stored as the pair $\{H(H(k)), E_{H(k)}(R)\}$. Using the keyword as the source for the key allows the receiver of the file to obtain the key by guessing.

Querying for an *RBlock* is **not** straightforward. The problem is that the intuitive choice for a query hash, $H(H(k))$ does not give intermediaries a way to verify that the (encrypted) response is correct. Instead of using $H(H(k))$, the initiator sends the so-called *triple-hash* query $H(H(H(k)))$. A peer that has stored $E_{H(k)}(R)$ under $H(H(k))$ can match the query hash with the response and send back the pair $\{H(H(k)), E_{H(k)}(R)\}$. Intermediaries can verify that the proof of authenticity, $H(H(k))$, hashes to the query $H(H(H(k)))$. Malicious peers cannot construct the proof $H(H(k))$ without guessing k or inverting the hash function.

The problem with the triple-hash is that an adversary that is routing a response to a triple-hash request can substitute the encrypted content in the $\{H(H(k)), E_{H(k)}(CHK, size)\}$ response while maintaining the $H(H(k))$ token. The resulting reply $\{H(H(k)), X\}$ still looks like a valid response to the triple-hash query $H(H(H(k)))$. The triple-hash thus does not protect against such a substitution attack by an intermediary. The impact of the attack is such that the network would propagate and possibly cache the invalid replies. The attack will not prevent the spread of valid replies through other routes in the system since multiple results to the same query are possible. Furthermore, any user searching for K will not see the invalid results. The reason is that the attempt to decrypt X with $H(K)$ will (with high probability) result in a malformed *RBlock*. Thus, the end-user is still protected from this type of spam. Nevertheless, the adversary can abuse network resources in the form of transmission and caching of invalid data.

Replacing the triple-hash with *KBlock*s for ECRS comes at a relatively high price. Where the triple-hash only requires a simple hash operation to verify

replies, *KBlock*s need to perform a public key operation. Worse, when publishing content or searching, the triple-hash scheme again only requires a few quick hash operations to compute the query or the content. With *KBlock*s, the cost increases to additionally generating a public-private key pair in addition to a public key operation and hash operations.

Both *KBlocks* and the triple-hash are vulnerable to the guessing attack. Guessing is slightly easier for the triple-hash since the computation for each guess is much cheaper. Overall, *KBlocks* work better than the triple-hash against adversaries that indiscriminately attempt to abuse network resources, but at the expense of higher CPU utilization even in the absence of an attack. Thus, using the triple-hash scheme may still be reasonable in networks with limited CPU power or less stringent security requirements.

6   Conclusion

The encoding for censorship-resistant sharing (ECRS) provides plausible deniability by hiding the nature of the queries and responses from intermediaries through encryption. The encoding uses convergent encryption and produces a ciphertext that is not significantly larger than the plaintext. Despite the fact that queries and responses are encrypted, intermediaries can *verify* that the responses are valid with respect to a particular query. Thus, the encoding mechanism provides robustness against tampering and false replies. Additional robustness is gained in ECRS by splitting files into blocks that can be *independently* verified, stored and distributed across the network. This makes it easy for protocols using ECRS to incorporate features such as swarm distribution [14], load balancing and microeconomics [12].

Since all handled blocks are small, nodes can participate in the network activity even if their own resources are limited. Also, storing small, individually unintelligible pieces of potentially objectionable content is likely to further support plausible deniability. While dictionary attacks on the keywords or known plaintext attacks on the content are still possible, they are harmless with respect to the goal of deniability for intermediaries. Using K-deterministic public-private keys for signing content it is possible to verify search results in a global keyword space.

ECRS is implemented as a part of GNUnet, a secure platform for peer-to-peer networking. In GNUnet, ECRS works towards implementing anonymous, censorship-resistant file-sharing. GNUnet is free software and part of the GNU project. The GNUnet code is approximately 100,000 lines of C code. ECRS itself is implemented in about 8,000 LOC. The code is freely available from the GNUnet webpage[4].

---
[4] http://gnunet.org/

Acknowledgements

References

1. Ross Anderson. The Eternity Service. In *Proceedings of Pragocrypt 1996*, 1996.
2. Krista Bennett, Christian Grothoff, Tzvetan Horozov, and Ioana Patrascu. Efficient Sharing of Encrypted Data. In *ASCIP*, volume 2384 of *Lecture Notes in Computer Science*, pages 107–120. Springer-Verlag Inc., 2002.
3. Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
4. Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.
5. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodorew W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Hannes Federrath, editor, *Designing privacy enhancing technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 25–26, 2000: proceedings*, volume 2009 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., 2001.
6. Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *The ACM SIGCOMM'02 Conference*, August 2002.
7. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.
8. Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In Hannes Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, July 2000.
9. John R. Douceur, Atul Adya, Wiliam J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical report, Microsoft Research, 2002.
10. Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing web censorship and surveillance. In *11th USENIX Security Symposium*, 2002.
11. Eu-Jin Goh. Secure indexes. In *Cryptology ePrint Archive*, Oct 2003.
12. Christian Grothoff. An Excess-Based Economic Model for Resource Allocation in Peer-to-Peer Networks. *Wirtschaftsinformatik*, 3-2003, June 2003.
13. Christian Grothoff. Reading File Metadata with extract and libextractor. *LinuxJournal*, 6-2005, June 2005.
14. John H. Hartman, Ian Murdock, and Tammo Spalink. The swarm scalable storage system. In *International Conference on Distributed Computing Systems*, pages 74–81, 1999.
15. Maurice Herlihy and J. D. Tygar. How to make replicated data secure. In *CRYPTO*, pages 379–391, 1987.
16. Raph Levien. Attack resistant trust metrics. Draft available at http://www.levien.com/thesis/compact.pdf, 2003.

17. Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge*, March 2002.
18. David Mazières. *Self-certifying file system.* PhD thesis, MIT, 2000.
19. David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.
20. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
21. Ralph C. Merkle. A certified digital signature. In *G. Brassard, editor, Advances in Cryptology—CRYPTO '89*, pages 218–238. Springer-Verlag, 1990. volume 435 of Lecture Notes in Computer Science.
22. Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.
23. Ginger Perng, Michael K. Reiter, and Chenxi Wang. Censorship resistance revisited. In Jordi Herrera-Joancomarti, editor, *Pre-Proceedings of the 7th International Workshop on Information Hiding*, pages 279–293, 2005.
24. Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
25. Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.
26. Adi Shamir. How to share a secret. In *Communications of the ACM*, volume 22, pages 612–613. ACM, 1979.
27. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.
28. Stratton Oakmont vs Prodigy Services Company, 1995 N.Y. Misc. Lexis 229, (N.Y. Sup. Ct. Nassau Co., 1995).
29. Marc Waldman and David Mazières. Tangler: A censorhip-resistant publishing system based on document entanglements. In *ACM Conference on Computer and Communications Security*, pages 126–135, 2001.
30. Marc Waldman, Aviel D. Rubin, 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.
31. Kevin Walsh and Emin Gün Sirer. Fighting peer-to-peer spam and decoys with object reputation. In *P2PECON '05: Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 138–143, New York, NY, USA, 2005. ACM Press.
32. Bryce Wilcox-O'Hearn. Experiences Deploying a Large-Scale Emergent Network. In *Peer-to-Peer Systems: First International Workshop, ITPTS 2002*, pages 104–110. Springer-Verlag Heidelberg, January 2002.