Tangler: A Censorship-Resistant Publishing System Based On Document Entanglements

Marc Waldman
Computer Science Department
New York University
waldman@cs.nyu.edu

David Mazières
Computer Science Department
New York University
dm@cs.nyu.edu

ABSTRACT

We describe the design of a censorship-resistant system that employs a unique document storage mechanism. Newly published documents are dependent on the blocks of previously published documents. We call this dependency an $entanglement$. Entanglement makes replication of previously published content an intrinsic part of the publication process. Groups of files, called collections, can be published together and named in a host-independent manner. Individual documents within a collection can be securely updated in such a way that future readers of the collection see and tamper-check the updates. The system employs a self-policing network of servers designed to eject non-compliant servers and prevent them from doing more harm than good.

1. INTRODUCTION

This paper makes the case for censorship-resistant publishing through document entanglement. The Internet is widely regarded as difficult to censor. Indeed, in a handful of well-known cases, such as the attempt to suppress DVD decoding software, the material being censored instead became widely replicated and more highly available. In addition, those responsible for publishing the software received free legal representation from non-profit organizations. However, DVD viewing is a particularly popular cause. In contrast, censorship-resistance is most important for those expressing unpopular views. In less high-profile cases, people often enjoy far less support for exposing corruption or criticizing schools, employers, and particularly litigious organizations.

In many cases, censoring documents on the Internet is fairly straight-forward. Almost any web page can be traced back to a specific server, and from there to an individual responsible for the material. Someone wishing to censor a web page can use the courts, threats, or other means of intimidation to compel the server administrator to remove the contents or reveal the author's identity. Even if these methods prove unsuccessful, various denial of service attacks can be launched against the server to make the page difficult or impossible to retrieve. Unless a web site's operator has a strong interest in preserving a particular web page, removing it is often the easiest course of action.

This research was supported in part by National Science Foundation Career award CCR–0093361.

CCS'01, November 5–8, 2001, Philadelphia, Pennsylvania, USA.

Several properties can make a publishing system far more difficult to censor than the web. First, to thwart censorship, documents should not be associated with any particular server. Data should periodically migrate from server to server, and multiple servers should store copies of a document and share responsibility for serving its contents.

In addition to diffusing the responsibility for serving documents, a censorship-resistant system should allow the source of a document to remain anonymous, so as to avoid real-world intimidation of the author. Unfortunately, attackers can themselves exploit anonymity in mounting attacks. For example, an attacker may attempt to exceed the capacity of a publishing system by anonymously publishing huge numbers of documents. Alternatively, the attacker may attempt to replace published documents with censored versions, preventing readers from obtaining the original.

Several past projects have explored censorship-resistant systems, with designs ranging from true peer-to-peer systems with thousands of nodes constantly coming and going to single servers that simply cannot delete one document without inflicting intolerable damage on others. In this paper, we propose a system called Tangler with a design that lies somewhere between current anonymous remailer networks and the Usenet news hierarchy.

As with anonymous remailers, we envision 10–30 Tangler servers, operated by volunteers around the world, with a general consensus of which servers are currently operational. Over time, servers will leave and new ones will appear. New servers, even if malicious, cannot seriously damage the system, leaving little incentive to exclude any volunteers. As with Usenet, servers have large storage capacities and good network connectivity, and each server stores an important fraction of the content of the entire system.

One of the key properties of Tangler is the lack of a one-to-one mapping from the stored blocks to published files. Rather, a stored block can be used to reconstruct several different files. Newly published documents are broken into blocks that must be combined with previously published blocks. The newly published documents are dependent on the previously published blocks. We call this dependency an *entanglement*. Entanglements not only break the one-to-one correspondence between blocks and files, but also provide a publisher with some incentive to replicate and store the blocks of other documents. Thus, replication becomes a fundamental part of publishing.

We have implemented entanglement, and believe the technique would apply to many existing censorship-resistant publishing projects. This paper also gives the design of a Tangler block storage network intended to accentuate the benefits of entanglements. The Tangler network resists many potential attacks to which anonymous publishing systems can fall prey and additionally makes auditing servers' behavior a fundamental part of publishing.

2. CENSORSHIP RESISTANCE AND DESIGN GOALS

The main goal of a censorship-resistant system is to keep published documents available in the face of attempts to censor them. Therefore a censorship-resistant system is in large part shaped by the threats it faces.

In this section we examine the types of threats that a publishing system may face and briefly describe ways to cope with these threats. At the end of this section we list the design goals of our censorship-resistant system called Tangler.

2.1 Attacks on Storage

A censorship-resistant system must replicate published documents. A sole replica presents a single point of failure that can be exploited by an adversary or, for example, made unavailable by an act of nature.

In order to achieve a high degree of replication many censorship-resistant systems rely on volunteers to donate disk space. This donated disk space allows the censorship-resistant system to be used like a distributed file store. However, even with a large number of participating servers, a publishing system only has a finite capacity. Once this storage has been filled, new documents cannot be published until old documents expire or are deleted. By filling the system with random files, an attacker can exhaust available disk space and therefore make the system unusable to other publishers.

This sort of block flooding attack is a form of denial of service, as it prevents future publishers from using the system. The usual way of combating this type of attack is to charge the publisher for disk space. This charge can take the form of anonymous e-cash or a CPU-based payment system that forces the publisher to perform some sort of work—possibly even of use to the censorship-resistant system.

2.2 Document Deletion

The most obvious way to censor a published document is to delete it from all hosting servers. An attack with the same end result is to simply force the hosting servers off the network so that potential readers cannot contact the servers.

An adversary can use threats or the legal system to force individual server administrators to delete certain documents. In addition, the adversary can attempt to remove the servers from the network by threatening the server's network provider. These attacks clearly show how the power of the adversary can affect the censorship resistance properties of a system. While a single individual or company might not be able to successfully remove a document from all participating servers, a government certainly might possess such power.

The main way of dealing with these types of attacks is to highly replicate the published documents. Ideally, the replicated documents would be stored on servers in many different countries and judicial domains. This clearly makes these sorts of adversarial attacks harder to execute successfully.

2.3 Document Tampering

A less obvious form of censorship is the modification of previously published documents. If an adversary can arbitrarily change the content of a document then he has succeeded in censoring it. This form of censorship is especially easy for server volunteers to perform. Each server volunteer completely controls the disk space that he has donated to the censorship-resistant service. Therefore he can arbitrarily modify any of the stored files.

The most effective way of combating this form of censorship is to provide a tamper-check mechanism for retrieved documents. This is usually done by embedding a cryptographic hash of the published document in its name. Thus, anyone who possess the document's name can verify the integrity of a copy retrieved from a server.

2.4 Rubber-Hose Cryptanalysis

Some censorship-resistant systems allow publishers to update or delete previously published content. These features make the system more usable as opposed to censorship-resistant. A system that supports document updates allows the publisher to change the published document and republish it in such a way that a reader will always view the latest version of the document.

The delete operation allows publishers to delete content that has been accidentally published or is simply no longer relevant. However, both of the update and delete operation can be exploited by an adversary. If an adversary finds the individual responsible for publishing a particular document, the adversary can use threats, torture, blackmail, etc. to force the publisher to delete or update the offending document.

Many censorship-resistant systems provide some sort of anonymity service that allows an individual to publish anonymously. However, the degree of anonymity usually depends on the adversarial model. For example, most anonymizing systems assume that an adversary can only control a certain number of servers or that he can view only a portion of the network traffic.

2.5 Goals

Below we list the design goals that were important in shaping Tangler.

Dynamic Server Participation. New servers should be allowed to join and participating servers allowed to leave. This means that possible adversaries could join the system and attempt to corrupt other servers, learn the identity of a publisher or try to subvert the rules of the system.

Previous Document Replication. The replication of previously published material should be an integral part of the publication process. This increases the number of replicas of previously published documents and therefore makes the censor's work a bit harder.

Publisher and Reader Anonymity. The system should provide a degree of anonymity to both document readers and publishers.

Secure Update. A publisher can securely update previously published material.

Publisher caching incentive. The publisher of a document has some incentive to cache the blocks belonging to previously published documents. This leads to greater replication.

Publishing limit. A publisher can publish no more than a certain fraction of what he is willing to store. This is intended to limit the damage done by a malicious publishers trying to fill up all available space—a denial of service attack.

Location-independent naming. The name of a document should not be tied to a specific network address. This helps prevent adversarial attacks against specific network locations that are holding the published material. It also allows published material to be relocated.

Self-policing. As long as a majority of the participating servers are honest, misbehaving nodes can be identified and temporarily ejected from the system. The reason for the "temporarily" qualifier is that misbehaving nodes can always reappear under a new name (IP address and public key), and can therefore rejoin the system.

All servers perform useful work. Before being allowed to join the system a server must perform some useful work for the system. This work may include redundantly storing or indexing documents. This ensures that a server bent on adversarial behavior performs useful work before being allotted full access to the system.

Document links. Similar to the world wide web's hyperlinks there should be a method of linking to previously published documents. These links should point to the latest version of the particular document and contain an embedded tamper-check mechanism. This mechanism is used to tamper-check the retrieved document.

3. RELATED WORK

In this section we briefly describe the relevant characteristics of other censorship-resistant systems. In addition, we describe some peer-to-peer systems that were not designed to be censorship-resistant but could conceivably be used as building blocks for such a system.

3.1 Censorship-Resistant Systems

Current censorship-resistant publishing systems use a variety of techniques for document distribution and storage. Roughly speaking, current systems distribute documents in one of two ways. The first is to redundantly store the document on a large collection of servers participating in the publishing system. One of the advantages of this approach is that only one server needs to be available in order to successfully retrieve the document. Systems that fit into this category are Freenet [4] and Publius [21]. In Freenet, documents are named by the cryptographic hash of the title or description of the document (e.g. the hash of the phrase "The Declaration of Independence"). In Publius, documents are encrypted and named by a special URL that specifies the various hosts that are storing the document. In addition, possession of the URL allows one to perform a tamper check on the retrieved document. Publius redundantly splits the document's decryption key and stores the pieces on various servers. Therefore, at least some of the servers that hold a piece of the decryption key must be available in order to read a Publius document.

The second category, into which Tangler falls, consists of systems that break the published document into a number of smaller blocks. Each of these blocks is treated independently and stored on a subset of the participating servers. In order to reconstruct the document, requests for the component blocks are sent to a subset of participating servers. The servers that possess these blocks send them back to the requester. Frequently the component blocks are redundantly stored such that a document can be reconstructed even if a portion of the participating servers are unavailable. Systems that fall into this category are Free Haven [6], Intermemory [9] and Mojonation [12]. For each published document a structure similar to an inode needs to be created to store the name or address of the document's component blocks. Systems in both categories may encrypt or otherwise obscure the contents of the document so that the servers cannot readily identify the content they are hosting and therefore have less of an incentive to censor it.

In each of these block based systems, an individual file block belongs to exactly one document. This usually means that a server administrator has no plausible excuse for retaining a particular file block when he is being pressured or threatened to delete it. The document inode itself becomes a very attractive adversarial target as the document cannot be retrieved without it.

Stubblefield and Wallach in [20] describe a publication method that is somewhat similar to our entanglement. They use the term "intertwine" to describe the XORing of newly published documents with those of previously published documents. Newly published documents are "intertwined" with previously published ones. However, in order to read a published document one must retrieve all of the documents that were intertwined with it. This is not the case with entanglements.

Some of the previously mentioned censorship-resistant systems suffer from a flat name space which can lead to to file name collisions and so-called "name squatting." In this context, "name squatting" refers to the adversarial practice of publishing an empty or meaningless file with a specific file name in order to prevent others from using that same name.

An issue somewhat related to naming is that of pseudonymous publishing. A pseudonymous publishing system allows groups of documents to be be linked to one publisher while the publisher himself may remain anonymous. This allows the publisher to gain some sort of reputation over time and to update previously published documents.

Many of the previously mentioned systems are still in their infancy—either existing in design only or deployed in a testbed fashion. Therefore it is difficult to judge the scalability of these systems. Most of these systems exhibit scalability tradeoffs. Freenet, for example, allows servers to join and leave at will, however the new servers may not immediately be able to locate all published content.

3.2 Peer-to-Peer Systems

In true peer-to-peer systems, there is no distinction between a server and client. Therefore we will identify computers participating in a peer-to-peer system as nodes. While Tangler assumes that all severs know of each other, most peer-to-peer systems are designed to scale to the point that not every node knows about all other nodes.

Gnutella [8] is a file sharing application that allows participating nodes to query for and copy files stored on other participating nodes. There is no formal publication method. A node simply interprets queries in any way it sees fit and sends back the names of files that is feels matches the query. As each request is essentially broadcast to other participating nodes the communication costs are quite high. While anonymous searching is supported, all file transfers are done in a point to point fashion and are therefore not anonymous. A node that wants a copy of a file directly contacts the node holding that file and performs the transfer. Tangler's block lookup protocol is more efficient than Gnutella's flood queries in the expected number of servers that must be contacted. However, joining and leaving the server network is far more heavy-weight and less scalable in Tangler than in Gnutella.

CFS [5] is a peer-to-peer file storage service. CFS utilizes a unique routing algorithm called Chord that not only allows nodes to join or leave at will but also greatly reduces the communication costs associated with finding nodes that hold the needed file. Tangler and Chord both employ consistent hashing to route queries. Chord is designed to support far more nodes than Tangler, however. Chord nodes only need to know about $O(\log(N))$ other nodes, making joining and leaving the system very efficient. Chord queries contact $O(\log(N))$ servers. In Tangler, queries only contact a constant number of servers, but every server must know about every other server. Unlike Chord, Tangler actively migrates data between servers so that, over time, different servers will be responsible for any given data block.

PASTRY [17], CAN [15] and Tapestry [22] are all peer-to-peer routing algorithms with goals similar to Chord that have low communication overhead and yet still scale to a very large number of participating nodes.

4. TANGLER DOCUMENT COLLECTIONS

The Tangler system consists of a publishing program that transforms documents into blocks, a reconstruction program that fetches blocks to reconstitute documents, and a network server daemon that permits the distribution and retrieval of blocks in a collec-

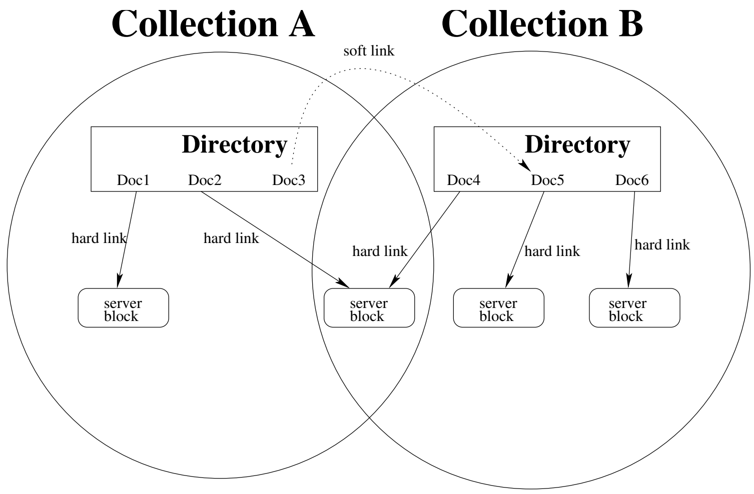

Figure 1: Collections can be made up of files (hard links) and links to other collections (soft links)

tion of servers. This section explains Tangler's approach to document naming and content authentication, and describes how Tangler transforms published content into fixed-size blocks suitable for injection into a storage network. These fixed-size blocks are constructed so that they can potentially belong to multiple documents, the property we call entanglement. Section 5 describes the algorithm used to entangle blocks. Though Tangler does not currently have an implemented network daemon, Section 6 proposes the design of a self-policing block server network that can survive certain flooding attacks and the existence of corrupt servers.

4.1 Collections

Every document in Tangler is published as part of one or more *collections*. A collection is a group of documents that are published by the same person under the same public key. Collections are published anonymously, but the person who published a collection can update it. Thus, anonymous collections may build reputations. A collection can consist of a single document, multiple versions of a single document, multiple documents, or soft links to documents in other collections. One can think of a collection as a directory containing a group of related files, subdirectories, and links. For example, a collection may contain a group of technical reports, the files making up a web site, or an index of other collections.

Each collection is named by a public key, K. Tangler refers to documents in a collection as K/*name*, where *name* represents the name chosen by the publisher for the document. For example, lets assume we wish to publish a collection consisting of the files that make up the screenplay for a movie entitled "Last *Tangle* In Paris." The parameters to the publish program are the collection members (files, directories, and soft links), and a public/private key pair. The publish program entangles the collection members. The public key is used to name the collection. The private key is used to sign the collection.

The reconstruction program retrives documents and places them in subdirectories of a Tangler root directory, named by public key. For example, suppose the Tangler root is `/tangler`. The reconstruction program, when asked, might place `act1` in `/tangler/75b4e39a1b58c265f72dac35e7f940c6f093cb80/act1`, where "`75b4e39a1b58c265f72dac35e7f940c6f093cb80`" is the collection's public key.

As previously stated, collections consist both of documents and links to documents in other collections. Links to documents in other collections are referred to as *soft links*. When a collection is updated, soft links into that collection reflect the new contents.

The reconstruction program represents soft links as symbolic links in the file system. Documents within a particular collection are known as *hard links*. Two collections may actually contain hard links to the same document and share the same entangled blocks for reconstituting the document. However, if one collection is updated by linking the same name to new contents, the other collection will not reflect the change. Hard links are useful to preserve a document if one fears the collection it was published in may change or disappear.

As an example, the "Last *Tangle* In Paris" screenplay collection contains a hard link to the file `act1`. However, it might also contain a soft link to a collection published by the Paris Tourist Bureau entitled `3f9···2d1/current_events.html` (where `3f9···2d1` is the Tourist Bureau's public key). The Paris Tourist Bureau actually owns the collection and can therefore update it. Our collection merely points to it. Anyone reading the screenplay collection soft link will read the latest Paris Tourist Bureau collection. See Figure 1.

4.2 Hash trees

Tangler makes extensive use of the SHA-1 [14] cryptographic hash function. SHA-1 is a collision-resistant hash function that produces a 20-byte output from an arbitrary-length input. Finding any two inputs of SHA-1 that produce the same output is believed to be computationally intractable. Thus, we can assume no collisions and treat SHA-1 hashes as unique, verifiable identifiers for data blocks.

Tangler also relies on Hash Trees [11]. Hash trees allow one to to specify or commit to large amounts of data with a single cryptographic hash value. Using hash trees, users can efficiently verify small regions of the data without needing access to all of it. In a hash tree, the data being certified or committed to fills the leaves of an n-ary tree. Each internal node of the tree stores the cryptographic hashes of the child nodes. Assuming no hash collisions, then, the hash value of the tree's root specifies the entire contents of the tree. One can prove the integrity of any leaf of the hash tree to someone who knows the root by producing the values of intermediary nodes from the root to the leaf.

The SFS read-only file system [7] showed how to transform a file system into a hash tree. SFSRO clients can traverse a file system and verify the contents of individual file blocks starting only from a root hash. Tangler employs a similar technique to produce collections, but the specifics differ somewhat because of entanglements.

4.3 Server Blocks

In order to publish a collection, C, one runs a publisher program that takes as input a public/private key, and a directory of files. The program fetches random previously published blocks and *entangles* these blocks with the files in C to produce new blocks. (Section 6 describes how such random fetching can be implemented.) Finally, it signs a collection root structure. Thus, one must have C's private key to publish or update the collection. Once entangled, C depends inextricably on the randomly chosen, previously published blocks for the reconstruction of its files. The person publishing C must distribute both the blocks just created and the ones with which her collection is entangled. Thus, replicating others' documents is an inherent part of publishing.

The entangled output blocks of the publisher program are suitable for injection into a storage network. We call these blocks *server blocks* to differentiate them from the file *data blocks* that were input to the publisher program. Tangler names server blocks by their SHA-1 hash values. It records SHA-1 values in collection metadata structures and assumes blocks can be retrieved from the storage network by their hash values.

Each collection has a root. This root functions much as a root directory in a file system—it defines a starting point in the search for files. The one exception to the SHA-1 hash addressing scheme is the addressing of the collection root. Recall that a collection is signed and named by a public key. This public key therefore also names the collection's root block, and therefore must be present within that block. Thus, the storage network must support the retrieval of blocks by public key. As a collection can be updated, two or more collection roots with the same public key may appear in the storage network. To disambiguate the blocks, a version field is present within all collection roots. The version field is incremented each time a collection is republished.

For the rest of this section and Section 5, we assume a collection of storage servers that implement a distributed, public block pool. Participating servers can inject server blocks into this pool, and blocks can be retrieved by SHA-1 hash or public key. Section 6 discusses how to implement a storage network far less susceptible to attack and abuse than a simple block pool.

4.4 Publisher Program

The first step of the publisher program is to entangle each member file in a collection. Each file is split into fixed-size (16K) data blocks. The last data block may need to be padded to achieve the fixed size. Each data block is then entangled using the algorithm described in Section 5.2. The entanglement algorithm takes as input two random blocks from the block pool and one data block from a file being published. It outputs two new server blocks, which when combined with the randomly selected pool blocks can reconstitute the data block. Thus, for every data block a publisher entangles, she becomes interested in ensuring the availability of four server blocks in the public pool. A data block can actually be reconstructed from any three of its four associated server blocks, adding some fault-tolerance (see Section 5.3). Notice that we do not inject data blocks in the storage network, only server blocks.

Every entangled file has an associated *inode* data structure that records the SHA-1 hashes of the server blocks needed to reconstruct the file's data blocks. Once all the data blocks of a particular file have been entangled and the names of the associated server blocks recorded in an inode, the inode itself is entangled. This entanglement produces the names of four server blocks that can be used to reconstruct the inode. These four server block names are recorded, along with the associated file's name in the collection root. The collection root essentially provides a mapping between file names and inodes. The inodes, in turn, provide the information necessary to reconstruct the associated file. Collection roots also record the collection's soft links. Figure 2 shows the pseudocode for the publish algorithm.

```
Proc Publish (Collection C, PublicKey pk, PrivateKey sk)
  c=new CollectionRoot()
  for each file, f, in the post-order traversal of C:
      i=new Inode(f)
      for each data block, b, in f:
          p1=random server block selected from pool
          p2=random server block selected from pool
          (e1, e2)=entangle(b,p1,p2)
          store server blocks (p1, p2, e1, e2) in pool
          r=random_permutation(p1, p2, e1, e2)
          record b's dependency on (r) in i
      endfor
      /* entangle the inode */
      p3=random server block selected from pool
      p4=random server block selected from pool
      (e3, e4)=entangle(i,p3,p4)
      r=random_permutation(p3,p4,e3,e4)
      /* r stores the reconstruction address for inode i */
      record (f, r) in collection root
  endfor
  c.name=pk
  c.version=1
  digest=SHA_1(name, version)
  c.sig=sign(digest,sk)
End Publish
```

Figure 2: Publish Algorithm

A digitally signed collection root is padded to the same size as a server block, and also gets indexed by SHA-1 hash. Roots can therefore become entangled just as other server blocks. Soft links not only contain a target collection's public key, but also the target's version number at the time of publication, and its root block's hash. The version number ensures that a soft link will never be interpreted to point to an older version of the collection than the one visible to the publisher. The addition of the root block hashes ensure that the collection root can be reconstructed if it cannot be found, in the block pool, via public key lookup.

4.5 Retrieval

The storage network implementing the public block pool must allow anonymous queries. Users reconstructing documents need to retrieve a specific block by hash value without revealing their identity. Server blocks retrieved from the public block pool are tamper-checked by simply computing the SHA-1 hash of the block's contents and comparing it to the SHA-1 hash by which the block was named. Similarly, the signatures on collection roots must be verified.

Once in possession of a verified collection root, a server can attempt to reconstruct any of the files stored in (or named by) the collection. As you will recall, that the name of the server blocks needed to reconstruct a file's inode are listed in the collection root. The file's inode contains the names of all of the server blocks needed to reconstruct the file. Only a portion of the blocks listed in the inode will be needed. For example, an entangled data block requires only three of the four server blocks recorded for it in the inode. Once the necessary server blocks are retrieved, the reconstruction algorithm (Section 5.3) is applied to the blocks.

4.6 Update

In order to update a collection one simply republishes it using the same public/private key pair that was used to originally publish the collection, but a higher version number (for instance the date is an adequate version number). Files that have not changed since the previous version of a collection do not need to be reentangled. If the public pool contains two or more collection roots possessing the same public key, the lookup algorithm must return the root with the latest version number.

5. ENTANGLEMENT

In this section we detail the block entanglement and reconstruction algorithms. As the entanglement process relies on Shamir's secret sharing algorithm, we begin by briefly describing that algorithm.

5.1 Secret Sharing

Shamir [18] described a method of dividing up a secret, s, into n pieces such that only $k \leq n$ of them are necessary to later re-form the secret. Any combination of less than k pieces reveals nothing about the secret. The pieces are called shares or shadows and the secret, s, is represented as an element in a finite field.

To form a set of n shares one first constructs a polynomial of degree $k-1$ such that s is the y intercept of the polynomial. The coefficients of the polynomial are randomly chosen. So, for example, if our secret was the element $6 \in \mathbf{Z}_{11}$ and k equals 3 then an appropriate polynomial would be $y(x) = 7x^2 + 4x + 6$. To form the n shares we evaluate this polynomial n times using n different values of x. Each (x, y) pair formed from the evaluation of the polynomial forms a share. Of course, the x value of a share must never be 0 as that share would reveal the secret.

Performing interpolation on any of the k shares allows us to re-form the polynomial. This polynomial can then be evaluated at 0, revealing the secret. Combining fewer than k shares, in this manner, gives no hint as to the true value of the secret.

5.2 Entanglement

In our discussion of secret sharing we stated that each share consisted of an (x, y) pair. In our entanglement system the server blocks play the role of the shares.

As you will recall, a file to be published is divided into fixed sized data blocks. The last data block may need to be padded to achieve the fixed size. We view each of these data blocks as a y value. Since each share consists of an (x, y) pair we assign an x value of zero to each of the data blocks. With this addition the data block becomes a server block. Call the first such server block f_1. We now randomly select b server blocks from the block pool. Each of these pool blocks consists of an (x, y) pair. We then perform Lagrange interpolation on the b pool blocks and f_1. This forms a polynomial, p, of degree b. We can now evaluate p at different nonzero integers to obtain new server blocks. Each new server block is of the form $(x, p(x))$. We then store these new server blocks in the block pool. One could conceivably store f_1 in the block pool as well, however in a censorship resistant system one would usually not store this block as it consists of plaintext and therefore is an easy target of the censor. The server blocks, being shares, give no hint as to the content they have been entangled with.

This procedure must be done for every server block of the file to be published. A data structure similar to that of an inode is necessary to record which server blocks are needed to re-form the original data blocks and therefore the published file. The Tangler publish algorithm entangles the inode as well.

5.3 Reconstruction Algorithm

In order to reconstruct a data block of a file we need to retrieve at least k of the appropriate shares. Any k of the n shares will do. Lagrange interpolation is performed on these shares producing polynomial p. Evaluation of this polynomial at zero produces the server block corresponding to our original data block. By simply stripping away the x value we are left with the original data block. This is repeated for every data block of the file we wish to reconstruct.

5.4 Implementation Issues

We have implemented the entanglement based publish and reconstruction algorithm in Java. The implementation shows that entanglement does not impose a large performance penalty as several optimizations are used to speed up what might otherwise be considered a somewhat costly computation. Our publish algorithm takes as input the files to be entangled. Each file is divided into fixed sized data blocks. Each data block is converted into a server block with $x = 0$ and then entangled with two random server blocks. The polynomial formed from the entanglement is evaluated twice to produce two additional server blocks. Essentially our publish algorithm defines a secret sharing scheme with $n = 4$ and $k = 3$, with the added feature that two of the four shares are also shares of other blocks. The newly created server blocks are written to the block pool, however we do not use these blocks in future entanglements of the same file.

In our implementation each file to be published is divided into data blocks of size 16K. Therefore each server block consists of an x value and a 16K y value. Instead of simply performing interpolation on a large (16K) y value we treat each server block as a collection of (x, y) pairs. Each pair has the same x value. The y value is 2 bytes long which means that each server block consists of a single x value and $(1024 * 16)/2 = 8192$ y values. Interpolation is performed on corresponding (x, y) pairs of each server block. For example, the third (x, y) pair of each server block is interpolated and evaluated to form the third (x, y) pair of one of the new server blocks formed during the publication process. This process is further described in the analysis section below. All interpolation is done over the finite field $GF(2^{16})$.

Our current java implementation can reconstruct files at a rate of 600 KB/sec. Publishing incurs a fairly steep upfront cost because of block pool initialization and the use of the Java secure random number generator, but the incremental cost of publishing data scales linearly. Publishing a one megabyte file took 36 seconds, a two megabyte file took 41 seconds and a five megabyte file took 53 seconds.

5.5 Analysis

In a very basic sense entanglement and reconstruction consist of a server block interpolation followed by evaluation of the associated polynomial. In this section we look at the cost, in terms of interpolation, of the entanglement and reconstruction scheme.

Given a 16K data block, d, we entangle it to produce 4 shares, any 3 of which determine d. Two of these four shares come directly from the block pool. Data block d is first converted into a server block. The x value of this server block is 0 and the 16K data block forms the 8192 y values. We perform Lagrange interpolation on these three server blocks. Below is the Lagrange interpolation formula [19] for the unique polynomial $a(x)$ of degree at most t. The value t is three in our scheme.

$$a(x) = \sum_{j=1}^{t} y_{i_j} \prod_{1 \leq k \leq t, k \neq j} \frac{x - x_{i_k}}{x_{i_j} - x_{i_k}}$$

We must perform interpolation once for each (x, y) pair in the server block. This means that we must perform interpolation (and an evaluation) 8192 times per block. However, this operation can be heavily optimized as the x value remains the same. Below we show the computation necessary for interpolating three (x, y) pairs. Let (x_1, y_1), (x_2, y_2) and (x_3, y_3) denote the three points to be interpolated. Note (x_1, y_1) and (x_2, y_2) denote points in the two public pool server blocks and (x_3, y_3) denotes the point from the data block d. All arithmetic is done over the finite field $GF(2^{16})$.

$$\begin{aligned} a(x) \quad = \quad & y_1 \left(\frac{x - x_2}{x_1 - x_2}\right)\left(\frac{x - x_3}{x_1 - x_3}\right) \\ & + y_2 \left(\frac{x - x_1}{x_2 - x_1}\right)\left(\frac{x - x_3}{x_2 - x_3}\right) \\ & + y_3 \left(\frac{x - x_1}{x_3 - x_1}\right)\left(\frac{x - x_2}{x_3 - x_2}\right) \end{aligned}$$

The $entanglement$ procedure produces two new server blocks. Each new block is formed by evaluating $a(x)$ 8192 times with x assigned a random nonzero element from $GF(2^{16})$, generated at the same time the server block was created. Notice that the x values do not change, only the y values change. Therefore we only need to compute the fractional x terms once per server block. Therefore our interpolation is now reduced to three multiplications and three additions— certainly not prohibitive. As the computation is done over the finite field $GF(2^{16})$ the addition and subtraction operations are the XOR operation. Multiplication and division consists of three tabular lookups and one addition (for multiplication), or a subtraction (for division). All relatively inexpensive operations.

The reconstruction algorithm also utilizes interpolation in exactly the same manner. In this case the polynomial is evaluated once at zero. This evaluation reveals the secret (the data block).

5.5.1 Benefits

Entanglement has three beneficial consequences. The first is that it promotes the replication of blocks of previously published documents (the blocks in the pool). A publisher could easily generate random, useless server blocks to entangle with, or else entangle exclusively with blocks of his own previously published documents. However, neither of these alternatives is as beneficial as using the blocks of documents published by others. If we assume that an individual who publishes a document has a direct interest in caching and replicating it, then his actions are indirectly helping all documents entangled with it.

The second consequence is that each server block now "belongs" to several documents—those documents that have become entangled with the server block. This leads to the third consequence of entanglement—incentive to store the server blocks published by others. If a set of entangled blocks are necessary to reconstruct your own document then you have some incentive to retain and replicate these blocks.

5.5.2 Limitations

We believe the entanglement process provides several benefits and can be profitably grafted onto many censorship-resistant systems. However the system is not perfect. Below we outline some potential limitations of the technique.

Any censorship-resistant system incorporating entanglements will need to build in some incentive for an individual to entangle with unknown content. If all individuals only entangle with a few popular documents (e.g. The Declaration of Independence) then the desired replication of other, lesser known, documents will not take place.

Although the entanglement system does allow the reconstruction of a file from server blocks produced by others, at least one server block that the publisher has generated is required. As you will recall, b blocks are chosen from the block pool and are entangled with a data block from the file to be published. Several new server blocks are then produced. At least one of these new server blocks is needed to perform the interpolation required during document reconstruction process. If an adversary manages to delete all of these newly created server blocks then the original file cannot be reformed, even if all the original pool blocks are available.

6. TANGLER NETWORK

In this section, we propose a specific design for a Tangler network. The network is a collection of block servers that accept queries either for SHA-1 hashes or public keys and return corresponding blocks. The Tangler network protocol is intended to complement the benefits of entanglement. Specifically, while entanglement makes replicating other people's documents an inherent part of publishing, the Tangler network protocol additionally makes auditing servers' behavior inherent to publishing. Thus, in the ordinary course of events, a well-behaved server will very likely obtain irrefutable evidence of any malicious server's bad behavior. The evidence can then be used to eject the bad server from the system.

One of the important goals of the Tangler network is to let the system accept new servers without fully trusting them. The network must therefore withstand misbehaving servers. Because bad servers are quickly detected when they misbehave, the worst a bad Tangler server can do is reduce the capacity of the system by whatever storage it is contributing. However, the protocol does not let new servers consume storage during their first month of operation. Whatever capacity new servers provide only increases replication. By the time a server can actually reduce the system's capacity, it must have been performing useful work for some time and thus will have contributed more block-days of storage than it has consumed. Moreover, if, after being ejected, a malicious server attempts to rejoin the system under a different identity, the server will actually reverse what little damage it has inflicted by restoring lost capacity to the system.

The Tangler model assumes a collection of servers around the world, run by volunteers opposed to censorship. Users publish documents by anonymously submitting server blocks to servers. Blocks persist for a minimum of two weeks in the system, but must be refreshed by users to persist indefinitely. Each server has a long-lived public key, used for authentication. Servers can communicate with each other both directly and anonymously (using other servers as a mix network [2]). Different servers may dedicate different amounts of storage to Tangler, but each publicly certifies its capacity. There is a general consensus on the public keys and capacities of available servers.

The list of servers is maintained using a standard group membership algorithm (e.g., [16]). Convincing other nodes of a server's corruption is straight-forward. Many forms of corruption result in two contradictory messages digitally signed by the same server—a succinct proof of the server's misbehavior. The other main threat is that a bad server will refuse to answer requests properly. When a server fails to answer, the requester can forward its request through any other server, enlisting that server as a witness. Any participating server can validate the response to any Tangler request. Thus, the witness will either detect misbehavior or return a valid response to the requester. All Tangler operations are idempotent, so if a witness is faulty, the requester can safely resend its request through a different witness.

Each server has the right to consume other servers' storage in proportion to its own capacity. This right is conferred by digitally signed *storage credits*, good for the storage of one block for two weeks from the date of the credit's issue. Servers delegate the credits they receive to users who wish to publish blocks. How a server apportions its credits is entirely at the discretion of its operator. One server might introduce blocks in exchange for e-cash payments. Another might charge hashcash [1]. Another might charge for publication in human time—posing challenges that could not be answered by automatic "spamming" programs [13]. Alternatively, a server might only accept blocks from particular pseudonyms, or from members of the organization sponsoring the server. Whatever the policies of individual servers, however, no server can influence or limit how other servers dedicate their space.

The Tangler network hides the identity of the server whose credit was used to introduce any particular block. In fact, servers may not themselves know for which blocks their credits have been used. To foil attempts to trace publication by traffic analysis, new storage credits all become available at the same time, once per day.

6.1 Block-to-server mapping

The Tangler network uses consistent hashing [10] to map blocks to particular servers. The 160-bit output of the SHA-1 hash function is mapped onto points of a circle. Each server block is assigned the point on the circle corresponding to the SHA-1 hash value of its contents. Collection root blocks are assigned the point on the circle corresponding to a hash of their public key. Each server is also assigned a number of points on the circle proportional to its stated capacity—for example one point per 100 MBytes of storage. A server's points are calculated from its public key, K, and the number d of days since January 1, 1970. Server A with N points and public key K_A is assigned the values:

$$\text{SHA-1}\left(K_A, \lfloor dN/14 \rfloor\right), \text{SHA-1}\left(K_A, \lfloor dN/14 \rfloor - 1\right), \\ \ldots \text{SHA-1}\left(K_A, \lfloor dN/14 \rfloor - (N-1)\right)$$

Thus, each day roughly $1/14$ of a server's points change. After two weeks, a server has, with very high probability, entirely new points on the circle.

Each block is stored on the servers immediately clockwise from it on the SHA-1 circle. Figure 3 gives a simplified example, using a 6-bit hash function. There are three servers in the example, A, B, and C, each with four points on the circle. A block with hash 011001 is represented as a black triangle. Going clockwise from the block's position, we cross points belonging first to server A, then to server C. Thus, if we are replicating blocks twice, the block will be stored on servers A and C.

Since server public keys and the day number are well-known information, anyone can compute the current set of points on the circle. To look up a block given its hash (or public key, for collection roots), one must contact the server corresponding to that block's successor on the circle, trying subsequent points if the immediate successor is unavailable, misbehaving, or simply does not have the block. Even if a server is misbehaving or if there are slight inconsistencies in the list of known servers, the lookup algorithm will very likely locate a replica of the block in question.

Note that it is up to users to publish blocks on the servers where people will look for those blocks. Tangler does not prevent users from publishing blocks elsewhere. However, since storage credits are limited, it is in a user's best interest to spend her credits wisely. As server points move around the circle, users will need to republish server blocks. Since the duration of server points and storage credits is both two weeks, we expect that the maintainer of a stable collection would reach a steady state of reinjecting $1/14$ or 7% of her server blocks every day.

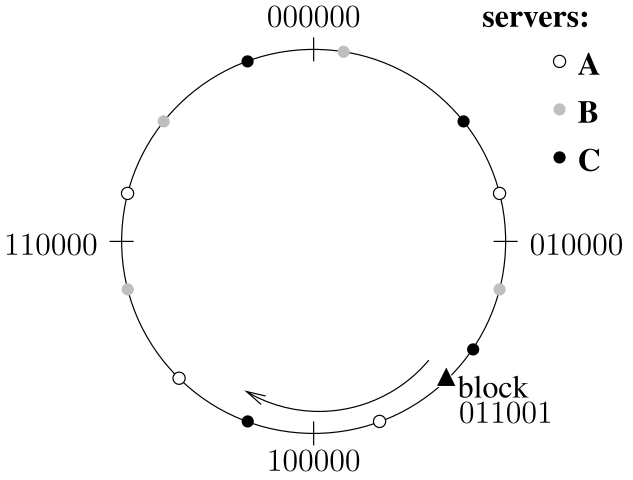

Figure 3: Use of consistent hashing to distribute server blocks amongst servers.

6.2 Block introduction protocol

The block introduction protocol has three phases. First, servers non-anonymously request storage credits from each other on behalf of anonymous users wishing to publish blocks. Second, servers issue *storage certificates* in exchange for blocks they receive when a storage credit is redeemed. Finally, each server commits to the blocks it is serving by producing a hash tree of all its blocks and signing the dated root of the tree. We describe each stage in more detail.

6.2.1 *Storage credits*

Every day, servers must accept $1/15$ of their stated capacity in new blocks to store for two weeks. [1] The process begins when servers request storage credits from each other. Servers exchange credits in proportion to their capacities. If server A has capacity c_A, server B capacity c_B, and the total capacity of all non-probationary servers is C, then A and B will request $c_A c_B / 15C$ credits from each other. New servers cannot request credits during their probationary period, but must grant credits just the same.

Credits are nothing more than digital signatures of block hashes, blinded [3] so that servers do not know for what blocks they are issuing credits. Specifically, each server A creates a daily temporary public key KT_A for signing storage credits. It certifies the key and the day, d, with its long-lived public key K_A, producing $\{\text{CREDIT-KEY}, d, KT_A\}_{K_A^{-1}}$. A server must certify exactly one such temporary key per day and must never reuse keys; two CREDIT-KEY certificates from the same server with the same d or KT_A are grounds for expulsion from the system.

A credit consists of a CREDIT-KEY certificate and the signed hash of some block m, $\{\text{SHA-1}(m)\}_{KT_A^{-1}}$. Credits are produced using blind signatures, so that A does not see the contents of the blocks for which it is issuing credits. When servers request credits on behalf of users, the users themselves blind the requests. Thus, even the server requesting a credit will likely not know what block that credit is for.

All requests for credits are numbered, tagged with the day number, and signed by the requesting server. Thus, a server can easily

[1] Because servers accumulate new blocks for the following day before deleting old blocks, a server may need to store 15 days worth of blocks. Thus, the daily block intake of $1/15$ capacity.

prove that another, bad server has requested too many credits by producing either a signature with too high a number or two different signed requests with the same day and number. If a bad server ever refuses a legitimate request for a credit, the requester forwards the request through other servers which it uses as witnesses. If the server persists in refusing to issue credits, it eventually gets ejected from the system. Note that there is no problem disclosing credits to witnesses, because the credits are blinded. Moreover, witnesses can easily check whether a credit corresponds to a particular blind request without needing to unblind the request.

At the end of the day, servers use remaining credits to republish some of their expiring blocks on servers that will be responsible for the next days points in the SHA-1 circle.

6.2.2 *Storage receipts*

Once a user has obtained a storage credit—say from server A for block m—she must transfer m's contents to A. This is done by anonymous communication through other servers. If m has not previously been published, the user sends its contents to A along with the block's unblinded storage credit. A, if honest, replies with a signed storage receipt, $\{\text{RECEIPT}, \text{SHA-1}(m), d\}_{K_A^{-1}}$. If A refuses to issue a storage receipt, the user anonymously enlists another server as a witness. The witness presents A with m and the storage credit. If A still refuses to acknowledge receipt, the witness forwards the request through other servers who either obtain a receipt from A or eventually eject A from the system.

If m has already been published, then instead of forwarding its contents to A, the user forwards the identity of another server, B, currently serving m. A must then either obtain m from B, return to the user a storage commitment from B that does not include m (see below), or else initiate the process of ejecting B from the system. In the case of a storage commitment excluding m, as explained below, if B was supposed to have stored m, the user will have a succinct proof of B's misbehavior and can anonymously initiate its ejection from the system.

6.2.3 *Storage commitment*

At the end of the day, after the exchange of storage receipts, each server makes its newly received blocks available and publishes a signed storage commitment. The commitment consists of the current date and the root of a balanced hash tree. The leaves of this tree contain a sorted list of hashes of every block the server is serving, followed by a sorted list of (public key, collection version) pairs, and for each block of either type the number of days the block has to live (initially 14). A server must be able to produce its current signed commitment and any node of the hash tree upon request, or else face ejection after the requester involves witnesses. A server must never sign more than one storage commitment per day. Two distinct commitments signed by the same server for the same day constitute grounds for expulsion.

Storage commitments prevent servers from discarding or suppressing blocks they have agreed to publish. Once a server has committed to storing a block by signing the hash tree root, it must produce that block on demand or face ejection after the requester involves witnesses. Every block lookup becomes a possible audit of a server's behavior. A server that has published a block can anonymously ask another server, the witness, to verify that a particular block is being stored on server A. The publishing server sends the storage receipt, that server A had previously signed, along with the verification request. This storage receipt proves that server A committed to hosting the block. Therefore, if server A cannot produce the block being verified, it will face eventual ejection from the system.

The hash trees in storage commitments play several other roles in Tangler. Users publishing documents use the trees to select random blocks to entangle with. To retrieve a random block, a user first selects a random server A, weighing the probabilities of the servers by their capacities. The user knows A's block capacity, c_A, and so can simply pick a random n, $0 \leq n < c_A$, and walk the hash tree from A's storage commitment root to find the hash of the nth block. (This is easy, since the tree is balanced and all leaves store the same number of hashes.) The user can then simply request block number n from A by its hash. Fetching random blocks in this way implicitly audits servers' behavior, as a server which loses some percentage of its blocks will very likely be discovered. Of course, requests for random blocks are sent anonymously so that servers cannot identify publishers by the blocks they have entangled with.

Servers also use the hash trees in commitments to fill their excess capacity. As discussed below, in the event of an unavailable server that does not appear to be corrupt, the network will have more storage than it issues credits. Though not enforced, good servers can fill any extra space with blocks near their recently acquired points on the circle from servers about to relinquish nearby points. Because the leaves of the commitment hash trees store blocks in sorted order, it is easy to find blocks near a particular point on the circle. Filing extra capacity in this way also implicitly audits other servers.

The public key list in a storage commitment can also be used to detect new collections. A search engine could make use of this information to index Tangler collections.

6.3 Discussion and limitations

The Tangler protocol we propose provides anonymity for publishing while preventing flooding attacks. Though blocks are dispersed untraceably across all servers, no server can consume more than its fair share of storage. Because different servers employ different block admission policies, it is unlikely that an attacker could simultaneously monopolize all servers' available storage credits. If an attacker did, servers could charge e-cash for storage credits and use the revenue to dedicate more resources to the system. The Tangler protocol also implicitly audits servers' behavior at many stages, ensuring that bad servers can quickly be ejected. By disallowing servers from publishing during their first month of operation, the protocol ensures that even bad servers do more good than harm.

One of the limitations of the protocol is its synchrony requirements. Certain behavior should obviously result in immediate expulsion from the system—for instance issuing two different storage commitments on the same day. It is less obvious what to do with a server that becomes unavailable for 24 hours, however. The basic protocol would eject such a server when it failed to issue storage credits or produce a storage commitment. Other options include delaying updates until the server returns (unscalable), or simply waiting for a few update cycles before ejecting the server.

If an unavailable server is not ejected before the system updates, there is a risk of revealing which blocks that server's credits have supported—those blocks will slowly disappear unless the user publishes them through another server. If the unavailable server still counts towards the non-probationary capacity of the system, servers will have more capacity than they issue storage credits (since the unavailable server will not request its credits). Some blocks originally introduced by the unavailable server may therefore be preserved by other servers filling their excess capacity. Other blocks will be reintroduced by other users because of entanglement. Nonetheless, a gradually increasing number of blocks will disappear.

Other possible attacks include attempting to use all of one server's credits towards a small set of other servers, so as to block a collection root that needs to be published on those servers. A corrupt server might also reduce performance by acting correctly but deliberately slowly. The Tangler network also needs to resist traditional denial of service attacks, such as a flood of block lookup requests. Conventional defenses such as hashcash are adequate for many attacks, but anonymity makes it harder to trace bad users.

7. SUMMARY

We have described Tangler, a distributed document storage system with censorship-resistant properties. Tangler transforms published documents into fixed-size blocks in such a way that many blocks actually belong to multiple documents. This technique, known as entanglement, diffuses responsibility from particular servers for particular documents, makes replicating other people's documents an inherent part of publishing, and furthermore gives anyone a plausible excuse for replicating any block in the system.

We also described the design of a self-policing storage network in which volunteers accept to store and serve entangled blocks. The network gives servers discretion over what they publish, but prevents any server from controlling what other servers publish. The protocol conceals the relationship between blocks and the servers that introduce them, but blinded storage credits prevent any server from consuming more space than it provides.

The Tangler network additionally leverages entanglements to make auditing of servers' behavior an inherent part of publishing. Most forms of misbehavior result in a server's immediate expulsion. The worst a bad server can generally do is reduce the capacity of the system by whatever storage it is was contributing before the expulsion. Because new servers are blocked from publishing during a probationary period, an ejected, malicious server that rejoins the system under a new identity will only reverse the damage it has done by restoring the system's lost capacity.

8. ACKNOWLEDGMENTS

We would like to thank Chuck Blake, Roger Dingledine, Frank Dabek, Kevin Fu, Frans Kaashoek, Zvi Kedem, Robert Morris, and the anonymous reviewers for their helpful and insightful comments.

9. REFERENCES

[1] A. Back. The eternity service. *Phrack Magazine*, 7(51), 1997. `"http://www.cypherspace.org/~adam/eternity/phrack.html"`.

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

[3] D. Chaum. Blind signature system. In *Advances in Cryptology—CRYPTO '83*, 1983.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. 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.

[6] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[7] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

[8] Gnutella Web Site. http://gnutella.wego.com/.

[9] A. V. Goldberg and P. N. Yianilos. Towards and archival intermemory. In *Proc. IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, pages 147–156. IEEE Computer Society, April 1998.

[10] D. Karger, E. Lehman, T. Leighton, M. levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[11] R. Merkle. A digital signature based on a conventional encryption function. *Advances in Cryptology: Proceedings of Crypto 87*, pages 369–378, 1987.

[12] Mojo Nation Web Site. `http://www.mojonation.net/`.

[13] M. Naor. Verification of a human in the loop or identification via the turing test. Unpublished draft from `http://www.wisdom.weizmann.ac.il/~naor/PAPERS/human_abs.html`, 1996.

[14] National Institute of Standards and Technology. Secure hash standard, 1995.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, San Diego, California, USA, August 2001.

[16] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 341–353, August 1991.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. `http://www.research.microsoft.com/~antr/pastry/`, 2001.

[18] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, Nov. 1979.

[19] D. Stinson. *Cryptography: Theory and Practice*. CRC Press, Inc, 1995.

[20] A. Stubblefield and D. S. Wallach. Dagster: Censorship-resistant publishing without replication. Technical Report TR01-380, Rice University, Houston, Texas, July 2001.

[21] M. Waldman, A. D. Rubin, and L. Cranor. Publius: A robust, tamper-evident and censorship-resistant web publishing system. In *Proceeding of the 9th USENIX Security Symposium*, pages 59–72, Denver, Colorado, August 2000.

[22] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-01-1141, U. C. Berkeley, April 2000.