Secure and Robust Censorship-Resistant Publishing Systems

by

Marc Waldman

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2003

David Mazières

For Mom, Dad and my sister, Dr. Meryl Waldman. Thanks for the unconditional love and support.

Acknowledgments

If the journey is the reward, then I have been handsomely rewarded. I first stepped into the Courant Institute of Mathematical Sciences (CIMS) 20 years ago, during the summer of 1983. I had just finished my freshman year in high school and had been selected to participate in the summer computer-programming course sponsored by NYU CIMS. The course was truly a wonderful experience. I enjoyed the course so much that I became actively involved with it for the following seven summers. I co- taught the course for several summers. Since that summer of 1983, I have been fortunate enough to meet many people at NYU who helped guide me on this journey. Below, I wish to thank several of them.

Professor David Mazières has been much more than a great advisor, he has also been a good friend. His door was always open to discuss any problem that happened to arise. David's enthusiasm and energy is infectious. I deeply appreciate the guidance and inspiration he has provided to me.

Professor Avi Rubin first introduced me to the area of computer security and censorship-resistant publishing. I cannot begin to express my gratitude to him. Working with Avi while designing and building Publius was a fantastic learning experience, one that I will never forget. One could not ask for a better role model.

Professor Dennis Shasha supported me during my first year of full-time study at NYU. He had great faith in me and for this I am eternally grateful. Working

with him on KLint and Plinda were some of the highlights of my academic career at NYU. Dennis is one of the brightest and kindest individuals I have ever met. It was a privilege to have worked with him.

Professor Zvi Kedem provided me with feedback on the Tangler design and was a member of my committee. I would be remiss if I didn't thank him for all his effort. In addition, I found Zvi to be one of the most dedicated and caring professors at NYU. When Zvi told the class that he would do something, you could depend on him to do it.

Henry Mullish ran the summer high school program that started me on this journey. He is truly one of the nicest people I have ever met. During my undergraduate years at NYU, I do not think a single week went by that I did not stop by his office to discuss some aspect of computing or the summer high school program. He has had a tremendous positive influence on me.

I must thank Anina Karmen and Rosemary Amico for all their help throughout the years.

On April 30th, 2003, I successfully defended my thesis. Shortly after this great event, I walked to Tisch Hall and visited room LC- 10. This is the room where the high school program first met during the summer of 1983. Although the room had been remodeled, it still brought back a flood of fond memories. It is hard to believe it has been 20 years. Yes, the journey is the reward.

Contents

Dedication iii

Acknowledgments iv

List of Figures xi

List of Tables xii

1 Introduction 1

1.1 Problem Statement 3

1.2 Dissertation outline 4

2 Design Challenges 5

2.1 Design Issues 6

2.2 Storage Servers 7

2.3 Naming and Tamper Protection 7

2.4 Data Storage 8

2.5 Untraceable Communication Channel 9

3 Related Work **10**

3.1 Censorship Resistant Systems 10

3.1.1 Usenet Eterntiy 10

3.1.2 Freenet 11

3.1.3 Intermemory 12

3.1.4 Janus and Rewebber 13

3.1.5 Free Haven 14

3.1.6 Dagster 15

3.2 Connection-Based Anonymity 16

3.2.1 Anonymizer 16

3.2.2 Crowds 17

3.2.3 Buses for Anonymous Communication 18

3.3 Integrity and Availability Schemes 20

3.3.1 SFSRO and CFS 20

3.3.2 Secure Replicated Data 22

3.3.3 Secure State Machine Replication 23

4 Publius **25**

4.1 Overview 26

4.2 Publish 27

4.3 Retrieve 29

4.4 Delete 29

4.5 Update 31

4.6 Maintaining a consistent state 33

4.7 Implementation 34
4.7.1 Publius URLs 34
4.7.2 Server software 37
4.7.3 Client software 38
4.7.4 Publishing mutually hyperlinked documents 39
4.7.5 Publishing a directory 40
4.7.6 Publius content type 43
4.7.7 User interface 43
4.7.8 Proposed Publication Mechanism 44
4.8 Limitations and threats 46
4.8.1 Share deletion or corruption 46
4.8.2 Update file deletion or corruption 46
4.8.3 Denial of service attacks 48
4.8.4 Threats to publisher anonymity 49
4.8.5 "Rubber-Hose cryptanalysis" 50
4.9 Publius Live Trial 51
4.9.1 Server Volunteers 51
4.9.2 Proxy Volunteers 52
4.9.3 Publius Usage 53
4.9.4 Issues Raised 54
4.10 Discussion 55

5 Tangler **56**

5.1 Design Goals 57

5.2 Tangler Overview 59
5.3 System and Adversary Model 61
5.4 Tangler document collections 63
5.4.1 Collections 63
5.4.2 Hash trees 64
5.4.3 Server Blocks 65
5.5 Publish 66
5.5.1 Reconstruction 68
5.5.2 Update 69
5.6 Entanglement 69
5.6.1 Secret Sharing 70
5.6.2 Entanglement Algorithm 70
5.6.3 Reconstruction Algorithm 72
5.6.4 Benefits and Limitations 72
5.7 Tangler network 73
5.7.1 Rounds 74
5.7.2 Tangler Servers 74
5.7.3 Join and Leave Protocols 75
5.7.4 Faulty Servers 76
5.7.5 Membership Changes 77
5.7.6 Block-to-server mapping 78
5.7.7 Storage Tokens 80
5.7.8 Publishing Clients 81
5.7.9 Block Storage 82

5.7.10 Storage receipts 84
5.7.11 Storage commitment 84
5.8 Implementation and Performance 86
5.8.1 Collection Data Structures 86
5.8.2 Tangler URLs 89
5.8.3 Hash Trees 91
5.8.4 Entanglement 93
5.8.5 Byzantine Agreement 95
5.8.6 Anonymous Communication Channel 98
5.8.7 Tangler Client Implementation 98
5.8.8 Tangler Server Implementation 106
5.9 Discussion and Future Work 109

6 Conclusion **111**

List of Figures

4.1 Publish Algorithm 27

4.2 Retrieve Algorithm 28

4.3 Publishing Mutually Hyperlinked Documents 41

4.4 Publius Publish Screen 45

5.1 Tangler Collections 61

5.2 Publish Algorithm 67

5.3 Consistent Hashing 80

5.4 Initial Inode Structure 87

5.5 Secondary Inode Structure 88

5.6 Collection Root Structure 89

5.7 Collection Tangler URL 89

5.8 Hash Tree of Order 3 92

5.9 Dolev-Strong Byzantine Agreement Algorithm 97

5.10 Tangler Publish Screen 101

5.11 Storage Coupon Message Format 103

List of Tables

5.1 Certifying Path Creation Time and Size (order 10) 90
5.2 Certifying Path Creation Time and Size (order 100) 90
5.3 Hash Tree Build Time In Seconds 93
5.4 Average Entanglement and Reconstruction Time In Seconds 95
5.5 Token Operation Performance In Milliseconds 106

Chapter 1

Introduction

In many cases, censoring documents on the Internet is a fairly simple task. Almost any published document can be traced back to a specific host, and from there to an individual responsible for the material. Someone wishing to censor this material can use the courts, threats, or other means of intimidation to compel the relevant parties to delete the material or remove the host from the network. Even if these methods prove unsuccessful, various denial of service attacks can be launched against a host to make the document difficult or impossible to retrieve. Unless a host's operator has a strong interest in preserving a particular document, removing it is often the easiest course of action.

A censorship-resistant publishing system allows an individual to publish a document in such a way that it is difficult, if not impossible, for an adversary to completely remove, or convincingly alter, a published document. One useful technique for ensuring document availability is to replicate the document widely on servers located throughout the world. However, replication alone does not block

censorship. Replicas need to be protected from accidental or malicious corruption. In addition, a censorship-resistant publishing system needs to address a number of other important issues, including protecting the publisher's identity while simultaneously preventing storage flooding attacks by anonymous users.

This dissertation presents the design and implementation of two very different censorship-resistant publishing systems. The first system, Publius, is a web-based system that allows an individual to publish, update, delete and retrieve documents in a secure manner. Publius's main contributions include a URL based tamper checking mechanism, a method for updating or deleting anonymously published documents and a method for publishing mutually hyperlinked documents.

The second system, Tangler, is a peer-to-peer system whose contributions include a unique publication mechanism and a dynamic self-policing network. The benefits of this new publication mechanism include the automatic replication of previously published content and an incentive to audit the reliability with which servers store content published by other people. In part through these incentives, the self-policing network identifies and ejects servers that exhibit faulty behavior. An important aspect of both systems is that the servers storing the published documents do not know the content of documents being stored. In Publius, each server stores encrypted documents. In Tangler, this is taken a step further—each server is storing blocks that have no meaning unless they are combined, in well-defined ways, with other blocks stored on the servers.

1.1 Problem Statement

In this dissertation, we will refer to the party wishing to censor documents as the *adversary*, and to the party wishing to publish documents as the *publisher*. All published documents reside on one or more *servers* that are connected to the Internet. Interested third parties, called *readers*, connect to these servers in order to retrieve and read the documents stored on them.

Censorship on the Internet exists in two main forms. Both forms attempt to prevent the readers from retrieving and reading documents stored on the servers. In the first form of censorship, the adversary attempts to prevent the reader from contacting the servers. This is usually accomplished by using some sort of blocking software such as a firewall. This form of censorship is practiced by many governments, including China, Saudi Arabia and Vietnam [58]. A number of software tools have been developed that attempt to circumvent this form of censorship [61, 26]. These tools were designed to route documents in a censorship-resistant manner—documents are routed through networks that are not blocked by the adversary.

In the second form of censorship, the adversary attempts to remove or modify documents stored on the servers. It is this form of censorship that this dissertation addresses. More specifically, this dissertation focuses on the design of censorship-resistant publishing systems. Such systems allow the publisher to publish a document in such a way that it is difficult, if not impossible, for the adversary to completely remove, or convincingly alter, a document stored on the servers. We say that such systems are robust if the document cannot be erased—the system

provides reliability. We say that a censorship-resistant publishing system is secure if the protocols can't be subverted to hide the document—the system provides availability.

1.2 Dissertation outline

This rest of this thesis is organized as follows.

Chapter 2: The issues and design challenges one must face when building a censorship-resistant publishing system.

Chapter 3: Related work.

Chapter 4: The design and implementation of Publius, a web-based censorship-resistant publishing system.

Chapter 5: The design and implementation of Tangler, a peer-to-peer-based censorship-resistant publishing system.

Chapter 6 (Conclusion): A brief summary of the contributions of Publius and Tangler.

Chapter 2

Design Challenges

Much of the early work in censorship resistant publishing has been done in the context of building a system to realize Anderson's Eternity Service [4]. The goal of the Eternity Service is to provide a server-based storage medium that is resistant to denial of service attacks and destruction of most participating file servers. An individual wishing to publish a document simply submits it to the Eternity Service along with an appropriate fee. The Eternity Service then copies the document onto a random subset of participating servers. Once submitted, a document cannot be removed from the Eternity Service. Therefore, an author cannot be forced, even under threat, to delete a document published on the Eternity Service. In his Eternity Service paper Anderson didn't specify the underlying algorithms and protocols needed to construct such a system. This has led several individuals and research groups to design and implement a variety of systems whose goals closely mirror or were inspired by the Eternity Service. In this chapter we examine the issues and design challenges one must face when building a censorship-resistant

publishing system.

2.1 Design Issues

The brief description of the Eternity Service gives us some idea of the architectural and design issues that must be addressed in order to build such a system. The most obvious issue is that of storage. We need a sufficient number of servers to participate so that published documents can be replicated. Once these servers have been procured, additional issues such as naming and storage policy must be addressed. Simply naming the documents via a publisher-provided fixed length string could cause problems if two documents are named the same. An adversary could publish a document with a specific name in order to prevent others from publishing under that same name. We need a storage policy to determine which servers will store the document. Does the document get stored on only a handful of servers or all participating servers? Storing the document on all participating servers could potentially make the document more censorship-resistant but may not make effective use of the system's limited disk space. If some of the servers could be run by potential adversaries, a mechanism will be needed to protect the integrity of the published document and possibly to protect the identity of the publisher. In the following sections, we briefly discuss each of these design issues in very general terms. Specific algorithms, protocols and cryptographic primitives are described in later sections.

2.2 Storage Servers

Clearly, a robust censorship-resistant system needs to consist of a collection of servers. A single server is vulnerable not only to a potential adversary but also to mother nature, the electric company and the network connectivity provider. An adversary can use threats, the law or other means to force the relevant parties to remove the server from the network. Ideally, the servers would be located throughout the world, in different judicial domains. This would make judicial attacks harder and probably make threat-based attacks more expensive.

All of the censorship-resistant publishing systems described in this dissertation employ a volunteer-based storage system in which individuals contribute space on their own servers. The notion of volunteering of one's computer resources for the benefit of others is well established and has been successfully applied to such areas as anonymous remailers [39] and distributed computing [57, 10]. However, allowing anyone to join the system makes it possible for adversaries to join the system. These adversaries can collude to manipulate the system in various ways. For example, a group of adversaries could attempt to censor, or expose the author of, a particular document.

2.3 Naming and Tamper Protection

Some mechanism is needed to name published documents. One possible scheme is first-come-first-served, but this scheme could lead to so called "name-squatting" attacks wherein an adversary publishes a document with a given name in order to prevent others from publishing under the same name. One way to prevent such

attacks is to name documents with the help of a cryptographic transform that cannot be easily forged (such as a hash or public key). Unfortunately, this method of naming documents usually prevents a publisher from associating a "meaningful name" with his published document. However, it does allow the publisher to embed tamper protection within the document's name. For example, a document could be named by the cryptographic hash of the document's content. An individual retrieving the document from a server can compute the cryptographic hash of the retrieved document and compare it against the hash used to name the document. If they are different, the individual knows the document has been modified. The individual can then attempt to retrieve the document from a different server.

2.4 Data Storage

There are a number of ways to store documents on the servers. The most obvious is simple replication—the document is copied to some subset of servers. However, this is not the only option. Documents can be broken into blocks and these blocks can be stored on the servers. Various transforms, such as encryption or error-correction encoding, can be applied to the document and the resultant data can be stored on the servers. We call this resultant data the *processed document*. Several censorship-resistant publishing systems encrypt documents before storing them on the servers. This is done to prevent the server's owner from knowing what content his server is storing. If a server owner does not know what content his server is storing, he may be less likely to censor or be held responsible for that content.

Another important issue is that of processed document placement—deciding which servers will store the processed document. The most widely used schemes rely on applying a function to the processed document. The output of this function determines which servers should be responsible for storing the document or fragments of the document. For example, distributed hash tables built using consistent hashing [35] map the processed document to the servers by applying the SHA-1 cryptographic hash to the processed document.

2.5 Untraceable Communication Channel

Censoring a document is sometimes easier if the adversary knows the identity of the publisher. The adversary may be able to use various legal and illegal threats to force the publisher, or relevant third parties, to remove the document or otherwise make it unavailable. Therefore, an untraceable communication channel is an important part of a censorship-resistant publishing system. It allows the publisher to protect his identity from the adversary. Untraceable communication channels typically allow two parties to anonymously communicate with each other—at least one of the parties does not know the true identity of the other party.

These channels can also be used to protect the identity of the reader. The reader retrieves a copy of the document. The adversary may wish to find and delete all copies of the document—this makes the reader a possible target of the adversary. Using the anonymous communication channel the reader can hide his true identity. In addition, the reader may be more comfortable reading a particular document if he knows his identity is protected.

Chapter 3

Related Work

In this section we survey previous work related to censorship-resistant publishing systems. Section 3.1 briefly describes several previously proposed censorship-resistant publishing systems. Section 3.2 examines the design of the untraceable communication channels that were described in Section 2.5. The last section describes systems that attempt to securely replicate data. While these systems were not designed to be censorship-resistant, secure replication is an important component of censorship-resistant publishing.

3.1 Censorship Resistant Systems

This section surveys previously proposed censorship-resistant publishing systems.

3.1.1 Usenet Eterntiy

Usenet Eternity [8] is a Usenet-based censorship-resistant publishing system. Using PGP [69], the publisher formats and signs the document to be published. The for-

matted document is then posted to the alt.anonymous.messages newsgroup. The Usenet NNTP protocol propogates the document to all subscribing news servers. This clearly leads to robust replication of the document as thousands of Usenet news servers exist worldwide [62]. However, using Usenet as the replication mechanism is not without problems. Usenet news servers typically store articles for only a short period of time. In addition, a posting can be locally censored by the news administrator or by someone posting *cancel* or *supersede* requests [62]. This latter problem is minimized by the fact that many Usenet news sites ignore cancel requests [25]. A piece of client software called the Eternity Server is used to read the anonymously posted articles in the alt.anonymous.messages newsgroup. The Eternity Server is capable of caching some newsgroup articles. This helps prevent the loss of a document when it is deleted from Usenet. The signature that is included with the posted document allows the reader to tamper check the document.

3.1.2 Freenet

Freenet [18] is composed a collection of volunteer servers that store published documents. Servers can join and leave the system at will. Each server maintains a database that "characterizes" the documents stored on some of the other servers in the network. This characterization is essentially a range of SHA-1 hash values. Documents are named by the SHA-1 hash of their content. The publisher connects to a server and submits a document for publication. The document is then stored on several Freenet servers. Exactly which servers will store the document depends on the database entries of the various servers. The database entries determine

which servers will be contacted when a retrieve or storage request is received. The reader contacts a server and requests a document by its SHA-1 hash. If the requested document is already on the server, the document is returned to the reader. Otherwise, the server forwards the request to another server. Each server forwards the request until either the document is found or the message is considered timed-out. If the document is found it is passed back through the chain of forwarding servers. Each server in this chain can cache the document locally. It is this caching that plays the main role in dealing with the censorship issue. The local caching increases the number of copies of the document, thereby making the job of the adversary a bit harder. Popular files will be retrieved, and therefore cached, more frequently. The main problem with Freenet is that there is no guarantee that one will actually be able to find the document. The database stored at each machine merely provides hints as to where a particular document might reside. A request for a file may time-out before it arrives at the server holding the file.

3.1.3 Intermemory

Intermemory [29] is a system for achieving an immense, self-replicating distributed persistent RAM using a set of networked servers. The publisher joins the Intermemory by donating some disk space, for an extended period of time, in exchange for the right to store a much smaller amount of data in the Intermemory. Each donation of disk space is incorporated into the Intermemory. Documents stored on the Intermemory are broken into blocks. An erasure-code transform (a form of error-correcting codes) is applied to each block resulting in a collection of smaller blocks

that are suitable for storage on the Intermemory. Only a small portion of these smaller blocks are needed to reconstruct the original block. A small Intermemory prototype is described in [17]. The security and cryptographic components of the system were not fully specified in either paper, therefore it is difficult to comment on its security or anonymity properties.

3.1.4 Janus and Rewebber

Janus [20] and Rewebber [32] take a very different approach to censorship resistant publishing. Rather than attempting to highly replicate a document both systems attempt to hide the true location of a web-based document by utilizing a URL rewriting scheme. With Janus, the publisher submits URL U to the server and receives a Janus URL in return. This Janus URL has the following form

http://www.janus.de/surf-encrypted/$E_k(U)$

Where E_k(U) represents URL U encrypted with Janus's public key. This new URL hides *U's* true value and therefore may be used as an anonymous address for URL U. Upon receiving a request for a Janus URL, Janus simply decrypts the encrypted part of the URL with its private key. This reveals the Web page's true location to Janus. Janus now retrieves the page and sends it back to the requesting client. Just before Janus sends the page back to the client each URL contained in the page is converted into a Janus URL. The problem with the Janus solution is that the Janus server itself can easily censor the document—either by not retrieving it or rewriting the content before being sent back to the requesting node.

Rewebber consists of a collection of volunteer servers, each of which runs an HTTP proxy server and possesses a unique public/private key pair. Each HTTP

proxy server is addressable via a unique URL. The publisher, who wishes to hide the true location of WWW accessible file f, first decides on a set of Rewebber servers through which a request for file f is to be routed. Using an encryption technique similar to the one used in onion routing (Section 3.2.1), the URLs of these Rewebber servers are encrypted to form a URL U. Upon receiving an HTTP GET request for URL U, the Rewebber proxy uses its private key to *peel* away the outermost encryption layer of U. This decryption reveals only the identity of the next Rewebber server that the request should be passed to. Therefore only the last Rewebber server in the chain knows the true location of f. The problem with this scheme is that if any of the Rewebber servers along the route crashes, or is operated by an adversary, then file f cannot be found. Only the crashed, or adversarial, server possesses the private key that exposes the next server in the chain of Rewebber servers that eventually leads to file f. The use of multiple Rewebber servers and encryption leads to long URLs. In order to associate a meaningful name with these long URLs the TAZ server was devised. TAZ servers provide a mapping of names (ending in .taz) to URLs in the same way that a DNS server maps domain names to IP addresses.

3.1.5 Free Haven

Free Haven [21] is unique among all the previously described systems in that it requires the participating servers to monitor each other's behavior. Published files are named by a public key and broken into blocks using Rabin's Information Dispersal Algorithm (a type of error-correcting code) [48]. The individual blocks are stored on participating servers. The Free Haven servers trade the blocks of pub-

lished files and communicate using anonymous remailers. The trading of blocks induces trust relationships that are developed over time. Violations of trust are broadcast to other servers in the Free Haven network allowing each server to adjust its trust ratings. In order to reconstruct a file, a server broadcasts a request for all the component blocks of the file. Each server that possesses one of the component blocks sends it back to the requesting server via the anonymous remailer. The frequent trading of blocks and the use of anonymous remailers makes it difficult for the adversary to determine the actual location of the component blocks. Free Haven has not been fully specified or implemented, therefore it is difficult to comment on its robustness properties.

3.1.6 Dagster

The publication scheme described in [65] shares some of the goals of the entanglement scheme described in Section 5.6. The term "intertwine" is used to describe the XORing of newly published documents with 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. Although XORing is more efficient than our entanglement scheme, we believe that entanglement is a more robust scheme due to the fact that not all "intertwined" documents need to be available to recover a published document. Dagster was intentionally designed to be brittle in order to dissuade the censorship of intertwined documents. Censoring an intertwined document could prevent other, perhaps more popular, documents from being read.

3.2 Connection-Based Anonymity

Connection-based anonymity tools are meant to provide an untraceable communication channel of the kind that was described in Section 2.5

Several anonymity tools have been developed around the concept of mix networks [15]. A mix network is a collection of routing nodes, called mixes, that use a layered encryption technique to encode the path that a message should take through a network. This layered encryption technique prevents individual routing nodes along the path from determining the content and ultimate destination of the enclosed message. Traditional mixes, also known as Chaumian mixes, use batching and message reordering to obscure the correlation between messages entering and exiting the mix network. Due to the batching, message delivery times are unpredictable and often lengthy. To address these problems a number of so-called real-time mix networks have been developed. These mixes provide quick message delivery at the expense of decreased protection against traffic analysis aimed at discovering which parties are communicating. One possible defense against this type of traffic analysis is to generate dummy traffic that is indistinguishable from regular messages that enter the mix network.

3.2.1 Anonymizer

The Anonymizer [5] provides connection based anonymity for HTTP requests. An individual wishing to anonymously retrieve a web page simply sends a request for that page to the Anonymizer. The Anonymizer then retrieves the page and sends it back to the individual who requested it. The Anonymizer simply acts as a proxy

for the requesting client. Clearly, the Anonymizer does not really offer any sort of robust anonymity. The proxy system knows the identity of the requesting client and therefore could potentially divulge the identity of the requesting client to a third party.

Onion Routing and Freedom

Onion Routing [50] is a real-time, mix based system for anonymous and encrypted Internet connections. An Onion Routing user creates a layered data structure called an onion that specifies the encryption algorithms and keys to be used as data is transported to the intended recipient. As the data passes through each onion-router along the way, it is decrypted and padded to a fixed size. All data received within a brief period of time is batched and reordered before being sent to the next onion router [66]. The data arrives at the recipient in plain text. Onion routing proxies have been developed for several application level protocols including HTTP and rlogin.

The Freedom anonymity system [31] provides an anonymous Internet connection that is similar to Onion Routing; however, it is implemented at the IP layer rather than the application level. This allows Freedom to easily support transport and application-layer protocols.

3.2.2 Crowds

Crowds [53] is a real-time anonymity system based on the idea that people can be anonymous when they blend into a crowd. As with mix networks, Crowds users need not trust a single third party in order to maintain their anonymity. Crowds

users submit their requests through a crowd, a group of web clients running the Crowds software. Crowds users forward HTTP requests to a randomly-selected member of their crowd. A crowd member, upon receiving an HTTP request from another crowd member, can forward the request to yet another member of the crowd or retrieve the document associated with the HTTP request. The crowd members that forwarded the HTTP request essentially form a path from the request initiator to the server storing the requested document. Notice that each member of this path cannot determine which crowd member initiated the request. Each member of the path is simply forwarding the request and can convincingly deny generating the request. Once the requested document is received from the HTTP server it is passed back over the same crowd members that forwarded the request. The main difference between a mix network and Crowds is in the way paths are determined and packets are encrypted. In mix networks, packets are encrypted according to a pre-determined path before they are submitted to the network; in Crowds, a path is configured as a request traverses the network and each crowd member encrypts the request for the next member on the path. Crowds also utilizes efficient symmetric ciphers and was designed to perform much better than mix-based solutions.

3.2.3 Buses for Anonymous Communication

When discussing the design of mix based communication systems we mentioned that various precautions needed to be taken in order to prevent traffic analysis. One such precaution was the generation of dummy traffic. Beimel and Dolev in [11] describe a communication channel that uses a unique mechanism to prevent traffic

analysis. A variation on this design, that is also presented in the paper, allows sender-anonymous communication. The design was inspired by the operation of a city bus. The bus makes various stops (nodes) on a circular path and individuals (messages) get off the bus depending on their intended destination. The protocol assumes all communicating nodes have a unique public/private key pair and know the unique predetermined circular path of the bus. Each communicating node knows the address and public key of every other participating node. The protocol also assumes a global clock with a global pulse such that a message can be delivered from one node to its neighboring node in one clock pulse. The bus arrives at a node at the beginning of a clock pulse.

The simplest version of the protocol does not provide sender anonymity. However we will briefly describe this protocol as it forms the foundation for the version of the protocol that does provide sender anonymity. The role of the bus is very similar to that of the token in the token ring network protocol. The node that the bus stops at is the only node that can send a message. The bus contains n^2 seats where n is the total number of nodes participating in the system. Once the bus stops at node i then node i can send a message to node j by simply placing the message in seat (i, j). Node i can write to any seat in row i in order to send a message to the node represented by the corresponding column. Before the message to node j is placed in seat (i, j) it is encrypted with the public key of j. In order to thwart traffic analysis node i places an encrypted message in every seat in row i. Encrypted dummy messages are placed in seats for nodes that i does not wish to send a message to. The encryption mechanism gives no hint to an adversary as to the type of message (legitimate or dummy) being sent. Therefore an adversary

cannot learn which nodes are communicating. Every pair of nodes is equally suspect. When the bus arrives at node k, node k decrypts every message in column k with its private key and discards any dummy messages.

To support sender anonymity fewer seats can be utilized—that is fewer seats are placed in each column. When node i wishes to send a message to node j it randomly writes into one of the seats in column j. The problem is that this random writing could lead to a situation in which one message overwrites another. The paper discusses some ways of improving the performance of this scheme.

3.3 Integrity and Availability Schemes

As stated in Section 2.2, replication is one of the major components of a censorship resistant system. Therefore we briefly survey some related work in the area of systems specifically targeting replication.

3.3.1 SFSRO and CFS

SFSRO [28] is a read-only file system that allows an individual to widely replicate files on untrusted servers. Files to be replicated are broken into data blocks. Each data block is named by the SHA-1 hash of the block's content. A data block's name is called its handle. Inodes are built to allow the reconstruction of files from data blocks. Each inode stores the handles of the data blocks that are needed to reconstruct the associated file. Directory data structures are created to link file names to inodes. The handle of an inode and directory is the same as that of a data block—the SHA-1 hash of its content. Files published together are named

by a public key. A root data structure is formed to hold this public key as well as information about the published files. The root data structure plays the same role as the root in a Merkle Hash Tree [40]. The leaves of the tree are the handles of the files, inodes and directories of the file system. Once the integrity of the root data structure has been verified one can verify the integrity of any file system component. The blocks and root data structure are packaged into a database that is replicated on many untrusted servers. The untrusted servers do not have access to the private key used to sign the root data structure. This means that an adversary cannot convincingly modify any part of the file system. The worst an adversary can do is simply delete the content thereby forcing a client to find another server that is hosting the database.

The Cooperative File System (CFS) [19] replaces the SFSRO replicated database with a scalable peer-to-peer distributed hash table. Individual data blocks are stored on a dynamic set of servers. Each server is named by the SHA-1 hash of its IP address. This hash is called the server's handle. Note that this is the same naming mechanism that was used by SFSRO to name data blocks and associated data structures. Efficient organization of participating servers is achieved using Chord [64]. Using the server handles, Chord essentially organizes the participating servers into an addressable virtual ring. The ring is traversed in a clockwise manner and therefore each server has a unique successor server. A data block's handle value determines the server, s, that will store the block. In order to achieve a measure of fault tolerance the block is also stored on the k servers that follow s in a clockwise traversal of the ring. Locating a particular block requires about $\log(N)$ servers to be contacted, where N is the number of participating servers.

CFS utilizes a caching mechanism to speed up queries for popular blocks. Once a block is retrieved it is replicated on the $\log(N)$ servers that were initially needed to locate the block. This allows faster future lookups as requests for a block may be satisfied from a server's local cache. Cached blocks are replaced using a least recently used replacement mechanism.

3.3.2 Secure Replicated Data

In [34] Herlihy and Tygar describe a method for replicating data on a collection of untrusted servers in such a way that fewer than t colluding servers cannot determine the content of the files being stored. The replication scheme requires a dealer, D, that creates a secret key, k, that will be used to encrypt data that will be stored on the servers. Using Shamir's secret sharing scheme [59], the dealer creates n shares such that any t shares can be used to reconstruct k. A single share is stored on each server. Authenticated clients contact t servers for shares and reconstruct k which can be used to encrypt files to be stored on the server and decrypt data coming from the server. The technique can also be used with public key cryptography—shares can be created for the public and private key. Using public key cryptography the dealer can create different share thresholds for encryption and decryption (i.e. three shares of the public key, pk, need to be combined to form pk and seven shares of the private key, sk, need to be combined to form sk). The technique is described in a quorum based distributed computing environment. In such an environment client requested operations do not need to take place on all participating servers. Rather operations are performed on a quorum of servers. A quorum is a subset of all participating servers. As long as

one of the quorum servers is contacted during subsequent operations the system can be kept in a consistent state.

3.3.3 Secure State Machine Replication

State machine replication is a general method for implementing a fault tolerant service by replicating the service and coordinating client interaction with server replicas [54]. A service is implemented by a collection of servers, each of which provides the same interface to clients. The replication allows the service to continue operation even when some servers crash, loose network connectivity or behave in a Byzantine manner. The challenge in state machine replication is to make sure that all participating servers execute the same sequence of instructions so that client queries are answered consistently across all servers. This level of consistency is not necessary for the censorship-resistant publishing systems described in this dissertation. However, the underlying techniques and algorithms could be of use to maintain the consistency of propogated information, such as the system membership. Indeed, some of these techniques were considered during the initial design of Tangler.

Each of the state machine based systems described below implement the notion of a view—a listing of the servers known to be participating in the system. The view may periodically change due to new servers being added or servers being removed. Rampart [52, 51] and BFS [14] were both developed to provide the foundation for programmers who are building Byzantine fault tolerant systems based on the finite state machine replication technique. BFS builds on the work of Rampart, however the underlying BFS protocols and algorithms are more efficient, simpler

to implement and don't rely on synchrony assumptions for safety. Therefore our discussion below will focus on BFS. The BFS system consists of a fixed collection of servers called replicas. The replicas are uniquely numbered $1...n$ where n is the number of replicas. At any time one replica is designated is the primary. The primary is responsible for ordering the requests made by clients. Upon receipt of a client request the primary initiates a three phase protocol to atomically multicast the request to all replicas. This ensures that all non-faulty replicas execute the client instructions in the same order. If any of the replicas believe that the primary is faulty it can begin a view change protocol that requires $2f + 1$ replicas to agree the primary is faulty (f is the maximum number of faulty replicas the system can tolerate). The view change protocol moves the system to a the next view. Views are sequentially numbered which allows the primary to be inferred automatically from the view number—the primary for view v is replica number v mod n where n is the number of replicas.

Chapter 4

Publius

Publius was designed to be an easy-to-use, web-based, robust censorship-resistant publishing system. It incorporates a number of features that were either lacking, or poorly integrated into, previously implemented censorship-resistant publishing systems. These features include a secure update and delete mechanism, a web URL based tamper-check mechanism and the ability of publish and retrieve mutually hyperlinked content. This chapter describes the design and implementation of Publius. The last section of this chapter describes the large scale trial of Publius that was conducted during the summer of 2000.

The name Publius was first used by the authors of the Federalist Papers, Alexander Hamilton, John Jay, and James Madison. This collection of 85 articles, published pseudonymously under the pen name Publius, was influential in convincing New York voters to ratify the proposed United States constitution [46].

4.1 Overview

Publius assumes that there is a system-wide list of available, geographically dispersed, volunteer servers. Published documents are encrypted by the publisher and copied to some of the servers. Therefore, each server has no idea what type of document it is hosting—it simply stores random looking encrypted data. The publish operation is robust—even if an adversary controls a majority of the servers storing the document, the document can still be retrieved. The publication process produces a special URL, called a Publius URL, that allows a reader to retrieve, decrypt and tamper check the published document.

Once in possession of the Publius URL, the reader can retrieve the encrypted document from one of the servers. The reader then uses the information embedded in the Publius URL to decrypt and tamper check the document. If the tamper check fails, the reader can repeat the process with an encrypted document retrieved from one of the other storing servers. Published documents are crytographically tied to their Publius URLs. Any modification to the stored document results in a failed tamper check.

Publius also provides a way for publishers (and nobody else) to update or delete their published docments. In the next several sections, we describe the design and implementation of Publius. Almost all of the examples and pseudocode presented in this chapter assume the publisher wishes to publish a single HTML document. Publishing more complicated content, such as multiple web pages that have links to each other, is covered in Section 4.7.

```
Procedure Publish (Document D, ServerList serverList, integer n, integer k)
    Generate symmetric key K
    {D}_K=Encrypt D under key K
    shares[1..n]=From K create n shares such that k shares are needed to re-form K
    storingServers[1..n]=DetermineStoringServers(serverList)
    for i in [1..n]:
        dirName=MD5(D · shares[i])
        store {D}_K and shares[i] on server storingServers[i] in directory dirName
    publiusURL=formPubliusURL(D)
    return publiusURL
End Publish
```

Figure 4.1: Publish Algorithm

4.2 Publish

The following text describes the publish pseudocode of Figure 4.1. For each document, D, to be published a symmetric key, K, is generated. In the initial design of Publius this key was generated from a combination of the SHA-1 and MD5 hash of M. SHA-1 and MD5 are collision-resistant hash functions that produce a fixed length output from an arbitrary-length input. Finding any two inputs of SHA-1 or MD5 that produce the same output is believed to be computationally intractable. Document D is then encrypted, using the Blowfish symmetric cipher [55], to produce $\{D\}_K$, D encrypted under K.

Using Shamir's secret sharing [59], key K is split into n shares such that any k of them can re-form K. Combining fewer than k shares reveals nothing about K. The values of n and k specify the robustness property of the published document—the encrypted document is stored on n servers such that k of these servers must be available to read the document. In the initial release of Publius the default values were $n = 10$ and $k = 3$, however, each value could be changed by the publisher.

```
Procedure Retrieve (PubliusURL u)
    parsedURL=parsePubliusURL(u)
    servers=parsedURL.getServers()
    hash=parsedURL.getTamperCheck()
    k=parsedURL.getNumRequiredShares()
    serversToContact=set of all unique k-subsets of the elements in servers
    while (serversToContact != {}):
        s=serversToContact.removeAndReturnRandomElement()
        shares={}
        for each server in s:
            shares.add(retrieve appropriate share from server)
        encryptedDocument=retrieve encrypted document from random server in s
        K=form encryption key from shares
        document=decrypt encryptedDocument using key K
        documentHash=cryptographicHash(document)
        if (documentHash==hash) return document
    return "Document could not be retrieved"
End Retrieve
```

Figure 4.2: Retrieve Algorithm

Recall that each publisher possesses a list of the available Publius servers. From this list n random servers are chosen. The encrypted document and one share are stored on each of these randomly selected servers. Each server stores a unique share. On the server, the encrypted document and share are stored in a directory named from the MD5 hash of the concatenation of the document and the share. Once the encrypted document and shares have been stored on the servers, a Publius URL is formed. This URL specifies which servers contain a copy of the encrypted document and a share. In addtion, the URL contains a cryptographic hash of the original document D. As described in Section 4.3, this hash allows the reader to tamper check the document. The exact form of the Publius URL is described in Section 4.7.1.

4.3 Retrieve

The following text describes the retrieve pseudocode of Figure 4.2. A reader must possess the Publius URL of the document he wishes to read. Given a Publius URL, u, the retrieve algorithm parses u to determine the servers that are storing the document, the cryptographic hash that is needed for the tamper check, and the number of shares needed to form the decryption key. Let k be the number of shares needed to form the decryption key.

The retrieve program generates the set of all unique k-server subsets—each of the k-subsets specifies k servers that can be contacted to retrieve the shares needed to reconstruct the decryption key. In addition, the encrypted document can be retrieved from any of these k servers. Once the shares and encrypted document have been retrieved, the shares are combined to form the decryption key, K. Using K, the encrypted document is decrypted. The cryptographic hash of the decrypted document is compared with the hash that was stored in the Publius URL. If they match, the document is returned to the reader. If not, the process is repeated with a new k-subset. If no match if found after all k-subsets have been tried, a failure message is returned to the reader.

4.4 Delete

It is sometimes desirable for a publisher to be able to delete previously published documents from all servers, while nobody else should be able to delete these documents. To achieve this, the publisher enters a password, PW, during the publication process. With the addition of this password the publication process involves

sending the encrypted document, share and $H(server\_domain\_name \cdot PW)$ to the servers that will be storing the published document. Let $H(server\_domain\_name \cdot PW)$ be the MD5 hash of the concatenation of the server domain name and the password PW. The server stores this hash value in the same directory as the encrypted file and the share, in a file called *password*. The reason this value is stored as opposed to just the PW or H(PW), is that it prevents a malicious server from learning the password and deleting the associated encrypted document from all other servers that are storing it. Even though the password itself is not stored on the server one must still choose this password carefully. A carefully chosen password is necessary because this storage scheme is vulnerable to a dictionary attack.

An adversary, possessing a Publius' password file and associated server IP address, can perform the password transformation on an arbitrary list of words. In particular the adversary can perform the password transformation on a list of guessed passwords (perhaps a whole dictionary). Once this list of passwords has been transformed into the Publius password format, the adversary can compare the list to the entry in the document's password file. If a match is found then either the adversary has found the password or he has found a collision in the hash function H. In the former case, the adversary can now update or delete the document on all servers storing the document. The IP address of all servers can be determined from the Publius URL. In order to prevent the dictionary attack the system allows the publisher to specify that a particular file is undeleteable.

A publisher, at the time of publication, has the option of specifying that a document should be undeleteable. Specifying this "do not delete" option tells the

hosting servers not to create the *password* file. The lack of the *password* file causes all delete and update (Section 4.5) requests to fail. Of course this doesn't prevent a server administrator from deleting the document from his server, however it does prevent someone from trying to delete the document via the Publius Delete protocol. This makes a brute force password guessing attack a waste of time as no password will delete the file.

To delete a published document, the client sends $H(server\_domain\_name \cdot PW)$, along with the name of the directory that is storing the document to delete. The server compares the password received to the one stored, and if they match, removes the directory matching the and all of the files in it.

4.5 Update

Publius provides a mechanism for a publisher to update a document that he previously published. The previously described password mechanism is utilized to protect the document against malicious update. Thus, only the individual possesing the proper password can update a previously published document. The idea is to enable the publisher to change content without changing the URL, because others may have linked to the original URL. After the update, anyone retrieving the original URL receives the newly updated document.

To update a document the publisher supplies the update program with the name of the file containing the newly updated document, the Publius URL of the document to be updated, the original password, PW, and a new password, NPW. The update program first publishes the new document using the publish algorithm

described in Section 4.2. The new password, NPW, is the update and delete password for the newly published document. This publication process results in a Publius URL, $newURL$. The update program then parses the original URL to determine which server are storing the original document. Each of these servers is sent a message specifying the name of the directory that contains the file to update, the associated hashed password $H(server\_domain\_name \cdot PW)$, and the Publius URL of the updated document, $newURL$. Each server then places the new URL in a file called *update*. This file is stored in the same server directory that contains the document to be updated.

When a reader requests the encrypted file or a share, if the update file exists, the server returns the update URL (located in the *update* file). The readers's client software receives the update URL from k servers and compares them, if they are all equal, then the new URL is retrieved instead. If the k URLs do not match, the client software tries contacting the other servers until it receives k matching URLs or k shares. If neither of these occur then the document cannot be retrieved.

Although the update mechanism is very convenient it leaves Publius content vulnerable to a redirection attack. In this attack several malicious server administrators collaborate to insert an update file in order to redirect requests for the Publius content. A mechanism exists within Publius to prevent such an attack. During the publication process the publisher has the option of declaring a Publius URL as nonupdateable. When a Publius client attempts to retrieve Publius content from a nonupdateable URL all update URLs are ignored. See Section 4.7.1 for more information about nonupdateable URLs.

Another problem with update is that it allows servers to discover what they are

storing. During the update process the entire Publius URL is stored at each server. Once in possession of the URL each server can decrypt the associated document and learn its content.

4.6 Maintaining a consistent state

The distributed nature of Publius means that at any particular time some set of servers may be temporarily unavailable. A server can be down for maintenance, network problems may prevent communication or the server may have simply crashed. Publius was designed to with this type of environment in mind—the multiple shares and redundant storage of the encrypted files ensure that Publius content can be retrieved even when a majority of the servers are unavailable.

In an environment where servers may be temporarily unavailable the application of either the delete or update command could leave Publius documents in an inconsistent state. For example suppose a document is published on 10 servers and that 3 shares are required to form the decryption key. At some later point in time the publisher of this document issues the update command. Unfortunately only 5 of the original servers can be contacted at the time of the update. Eventually all 10 servers become available and someone attempts to retrieve the published document. Depending on which servers are contacted either the original or updated document will be returned. The published document is in an inconsistent state. This same sort of inconsistent state can arise with the delete command—only a fraction of the servers may delete the requested document. Publius does not directly deal with this problem. Instead the client software displays a message detailing the

outcome of the requested operation on each server. Each time a server performs a Publius operation it sends an HTTP status code back to the requesting client. Any status code other than 200 (the HTTP "OK" status code) is considered an error. A description of the error is placed in the body of the response. These error descriptions along with server time-out error messages are displayed after every Publius operation.

Both the update and delete operations are idempotent. This means that there is no harm in re-updating a previously updated site or trying to delete previously deleted content. Therefore if an update operation fails on a set of servers the update operation can be performed again at some later point in time. Hopefully this second update operation will be issued to the previously unavailable servers. The same can be said for the delete command.

4.7 Implementation

In this section we describe the software components of Publius and how these components implement Publius functions. The Publius protocol is implemented on top of HTTP.

4.7.1 Publius URLs

Each successfully published document is assigned a Publius URL. The desire to remain compatible with older WWW browsers influenced the design of the Publius URL. Many older browsers enforce the rule that a URL can contain a maximum of 256 characters. This limit necessitated the need for the Publius server list.

Embedding the domain name of each Publius server in the URL would have been far too costly in terms of character usage. Even if text compression techniques were used the number of server names that would fit into the 256 character limit would have been far too small for our intended purposes. However, almost all current browsers essentially have no limit on the size of the URL. This allows us to embed the server names in the Publius URL, effectively eliminating the need for the server list. Simply embedding the domain name of the server is not enough as we need to know the location of the server's CGI script as well. This script provides the required Publius server functionality. This may seem unnecessary as we could simply ask all server administrators to install the CGI script in a specific directory (e.g. cgi-bin/publius). Such a system is very inflexible and may require reconfiguring the web server or obtaining greater access privileges. Therefore the full URL (without the "http://" prefix) would need to be stored.

Since the initial design of Publius, a more efficient and simplified document publication mechanism has been developed. Although this mechanism has not been implemented, its benefits are clear. This new publication mechanism utilizes the embedded server name URL format that was just described. This new publication mechanism is described in Section 4.7.8. The original Publius URL is described in this section.

A Publius URL has the following form

$$http://!publius!/options/\ encode(name_1)\ldots encode(name_n)$$

The *options* section of the Publius URL is made up of 9 characters that define

how the Publius client software interprets the URL. The *options* section encodes four fields – the Publius version number, the number of shares needed to form the decryption key, the size of the server list and finally the update flag. The version number allows the addition of new features to future versions of Publius while at the same time retaining backward compatibility. It also allows Publius clients to warn a user if a particular URL was meant to be interpreted by a different version of the client software. The next field identifies the number of shares needed to form the decryption key K. When publishing a document an individual can specify the number of shares needed to re-form the document decryption key. This value also indicates the number of update URLs that must match before the client will retrieve the update URL returned by the servers. The update flag determines whether or not the update operation can be performed on the Publius document associated with the URL. If the update flag is set to 1 then the retrieval of updated content will be performed in the manner described in Section 4.5. However if the *update* flag is 0 then the client will ignore update URLs sent by servers in response to share and encrypted file requests. The update flag's role in preventing certain types of attacks is described in Section 4.8.

Each of the URLs $name_i$ values consists of the MD5 hash of the concatenation of the published document (prior to encryption) and share i of the encryption key. A document published with 10 shares will have 10 $name_i$ values. The *encode* function is the Base64 encoding function [1]. The Base64 encoding function generates an ASCII representation of the $name_i$ value. Note that the $name_i$ values are dependent on every bit of document D.

The $name_i$ values are used to determine which servers are hosting the encrypted

document(and associated share) and to perform the tamper check on the decrypted document. As you will recall, each publisher and reader has the list of Publius servers. Assume this list has m entries. The retrieve algorithm parses the Publius URL to determine the individual $name_i$ values. For each of the $name_i$ values the retrieve algorithm performs the following calculation

$$location_i = (name_i \bmod m)$$

Each $location_i$ value yields a unique integer that is used as pointer into the Publius server list. Each $location_i$ value points to the Publius server that is storing one copy of the encrypted file and the i^{th} share.

To tamper check the document, the retrieve algorithm computes the MD5 hash of the concatenation of the unencrypted document and the i^{th} share. If this hash value is equal to $name_i$ then the document has passed the tamper check.

Here is an example of a Publius URL:

http://!publius!/010310023/VYimRS+9ajc=
B20wYdxGsPk=kMCiu9dzSHg=xPTuzOyUnNk=
O5uFb3KaC8I=MONUMmecuCE=P5WY8LS8HGY=
KLQGrFwTcuE=kJyiXge4S7g=6I7LBrYWAV0=

4.7.2 Server software

To participate as a Publius server, one only needs to install the Publius CGI script. This CGI script constitutes the server software. The client software communicates

with the server by executing an HTTP POST operation on the server's CGI URL. The requested operation (*retrieve*, *update*, *publish* or *delete*), the file name, the password and any other required information is passed to the server in the body of the POST request. The server then attempts to perform the requested operation and sends back a status code indicating the success or failure of the operation.

The CGI script has a number of configurable parameters. The server administrator may specify the maximum number of Publius files that may be stored on the server. Once this number has been reached the server will refuse any future publish requests and reply to such with requests with an appropriate error code. Successful delete requests free disk space and lower the published file count thereby allowing a previously "full" server to continue satisfying publish request. The maximum size, in kilobytes, of a Publius file is another parameter that can be set. The server will refuse to store any file larger than this specified maximum and will return an appropriate error code to the publishing client.

4.7.3 Client software

The client software consists of an HTTP proxy that implements all of the Publius operations. Therefore the terms client software and proxy will be used interchangeably. The proxy easily interfaces with standard web browsers. The proxy transparently sends non Publius URLs to the appropriate servers and passes the returned content back to the browser.

Most browsers that support HTTP proxies allow one to specify which URLs should not be sent to the proxy. This mechanism can be used to prevent non-Publius URLs from using the proxy and thereby escape the slight performance

degradation one would expect from the use of an HTTP proxy.

The proxy performs all Publius operations and is therefore the intermediary between the user and the Publius servers. The proxy can be run remotely or on the same host as the browser. This is covered in greater detail in Section 4.9.2. Upon receiving a retrieve request for a Publius URL the proxy first retrieves the encrypted document and shares as described in Section 4.3 and then takes one of three actions. If the decrypted document successfully verifies, it is sent back to the browser. If the proxy is unable to find a document that successfully verifies an HTML based error message is returned to the browser. If the requested document is found to have been updated then an HTTP redirect request is sent to the browser along with the update URL.

4.7.4 Publishing mutually hyperlinked documents

Suppose the publisher wishes to anonymously publish HTML documents C and D. Document C has a hyperlink to document D and document D has a hyperlink to document C. The publisher now faces the dilemma of having to decide which document to publish first. If document C is published first then the publisher can change D's hyperlink to C but cannot change C's hyperlink to D because C has already been published. A similar problem occurs if the document D is published first.

The problem for the publisher is that a published document is cryptographically tied to its Publius URL—changing the document in any way changes its Publius URL. This coupled with the fact that file C and file D contain hyperlinks to each other generates a circular dependency – each document's Publius URL depends

on the other's Publius URL. What is needed to overcome this problem is a way to break the dependency of the Publius URL on the file's content. This can be accomplished using the Publius Update mechanism described in Section 4.5. The update mechanism adds a level of indirection which can be used to publish mutually hyperlinked documents. Figure 4.3 illustrates the process of publishing mutually hyperlinked content.

4.7.5 Publishing a directory

Publius contains a directory publishing tool that automatically publishes all files in a directory. In addition, if some file, f, contains a hyperlink to another file, g, in that same directory, then *f's* hyperlink to g is rewritten to reflect *g's* Publius URL. Mutually hyperlinked HTML documents are also dealt with in the manner described in Section 4.7.4.

The Publisher could solve this problem by publishing all the files in the directory, rewriting the appropriate URLs within those files and then using the update mechanism to republish the files. This would certainly work but it would be inefficient—publication and update are an expensive operations. Clearly, when publishing a directory, it pays to minimize the number of publish and update operations we perform. A more efficient directory publish mechanism is described below.

The first step in publishing a directory, D, is to publish all of D's non-HTML files and record, for later use, each file's corresponding Publius URL. All HTML files in D are then scanned for hyperlinks to other files within D. If a hyperlink, h, to a previously published non-HTML file, f, is found then hyperlink h is changed to

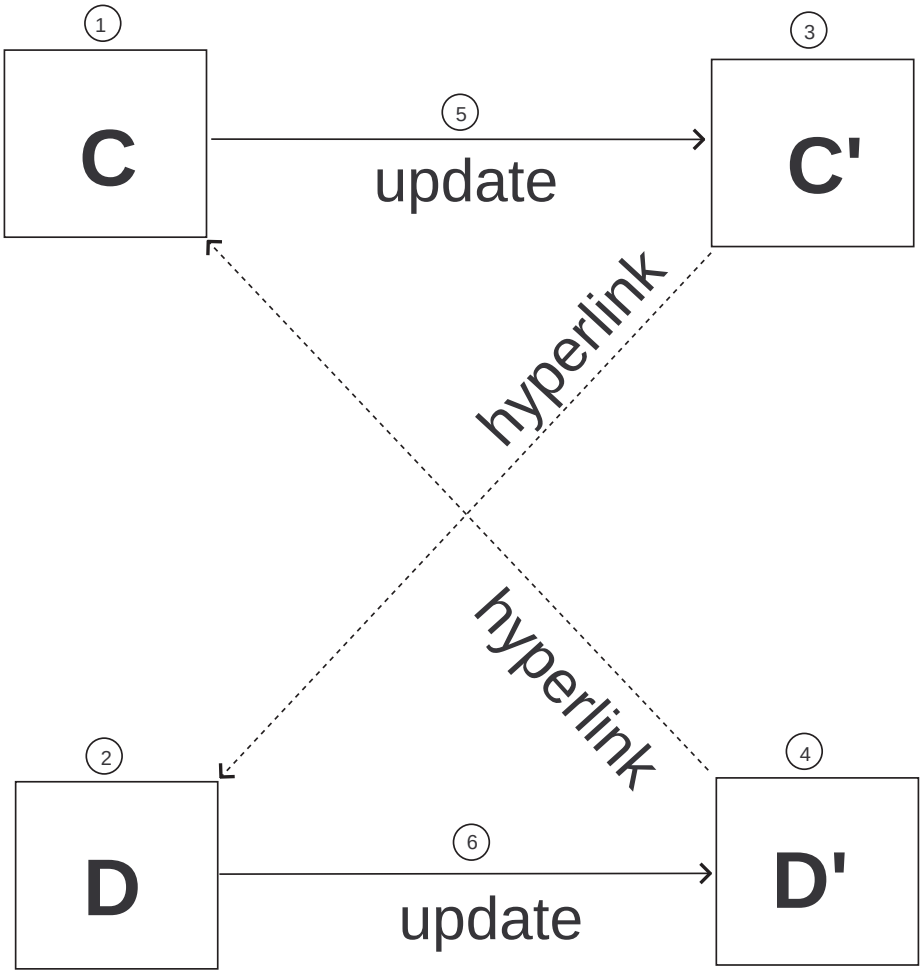

Figure 4.3: **Publishing mutually hyperlinked documents C and D**

1) Publish C, this generates C's Publius URL

2) Publish D, this generates D's Publius URL

3) Modify document C to reflect D's Publius URL, Call this document C', Publish C'

4) Modify document D to reflect C's Publius URL, Call this document D', Publish D'

5) Update C's URL to point to C'. All requests for C are redirected to C'

6) Update D's URL to point to D'. All requests for D are redirected to D'

the Publius URL of f. Information concerning hyperlinks between HTML files in directory D is recorded in a data structure called a dependency graph. Dependency graph, G, is a directed graph containing one node for each HTML file in D. A directed edge *(x,y)* is added to G if the HTML file x must be published before file y. In other words, the edge *(x,y)* is added if file y contains a hyperlink to file x. If, in addition, file x contains a hyperlink to file y the edge *(y,x)* would be added to the graph causing the creation of a cycle. Cycles in the graph indicate that we need to utilize the update mechanism for publishing mutually hyperlinked documents. This mechanism was described in Section 4.7.4.

Once all the HTML files have been scanned the dependency graph G is checked for cycles. All HTML files involved in a cycle are published and their Publius URLs recorded for later use. Any hyperlink, h, referring to a file, f, involved in a cycle, is replaced with *f's* Publius URL. All nodes in the cycle are removed from G leaving G cycle-free. A topological sort is then performed on G yielding R, the publishing order of the remaining HTML files. For example, if file B contains a hyperlink to file A then file A will be listed before file B in the output of the topological sort. Therefore, file A will be published before file B. Formally, the result of a topological sort of a directed acyclic graph (DAG) is a linear ordering of the nodes of the DAG such that if there is a directed edge from vertex i to vertex j then i appears before j in the linear ordering [2]. The HTML files are published according to order R. After each file, f, is published, all hyperlinks pointing to f are modified to reflect *f's* Publius URL. Finally the update operation is performed on all files that were part of a cycle in G.

4.7.6 Publius content type

The file name extension of a particular file usually determines the way in which a Web browser interprets the file's content. For example, a file that has a name ending with the extension ".htm" usually contains HTML. Similarly a file that has a name ending with the extension ".jpg" usually contains a JPEG image. The Publius URL does not retain the file extension of the file it represents. Therefore the Publius URL gives no hint to the browser, or anyone else for that matter, as to the type of file it points to. However, in order for the browser to correctly interpret the byte stream sent to it by the proxy, the proxy must properly identify the type of data it is sending. Therefore before publishing a file we prepend the first three letters of the file's name extension to the file. An alternative implementation could store the actual MIME type prepended with two characters that represented the length of the MIME type string. The file is then published as described in Section 4.2. When the proxy is ready to send the requested file back to the browser the three letter extension is removed from the file. This three letter extension is used by the proxy to determine an appropriate MIME type for the document. The MIME type is sent in an HTTP "Content-type" header. If the three letter extension is not helpful in determining the MIME type a default type of "text/plain" is sent for text files. The default MIME type for binary files is "octet/stream".

4.7.7 User interface

In order to use Publius an individual sets their web browser to use the client software (proxy) as their HTTP proxy. A Publius document can be retrieved in the same manner in which one retrieves any other web based document. The URL

can be entered into the browser or one can click on a preexisting hyperlink. A web based interface to the *update*,*delete* and *publish* operations has been developed (See Figure 4.4). This interface allows one to select the Publius operation (*update*, *publish* or *delete*) and enter the operation's required parameters such as the URL and password. Each Publius operation is bound to a special !publius! URL that is recognized by the proxy. For example the publish URL is !publius!PUBLISH. The operation's parameters are sent in the body of the HTTP POST request to the corresponding !publius! URL. The proxy parses the parameters and executes the corresponding Publius operation. An HTML based message indicating the result of the operation is returned.

4.7.8 Proposed Publication Mechanism

As described in Section 4.7.1, current browsers essentially have no limit on the size of the URL. This allows us to dispense with the server list and embed the storing server information directly into the Publius URL. In addition, the document's MIME type, decryption key and MD5 hash can also be placed into the URL. This simplifies the retrieval algorithm and tamper check process. One server can be randomly selected from the URL and the encrypted document can be retrieved from the server. The encrypted document can then be decrypted using the decryption key stored in the URL. The MD5 hash of the decrypted document can then be compared to the MD5 hash stored in the URL. If the tamper check fails, another copy of the encrypted document can be retrieved from one of the other servers. If the tamper check succeeds, the unencrypted document can be returned to the reader.

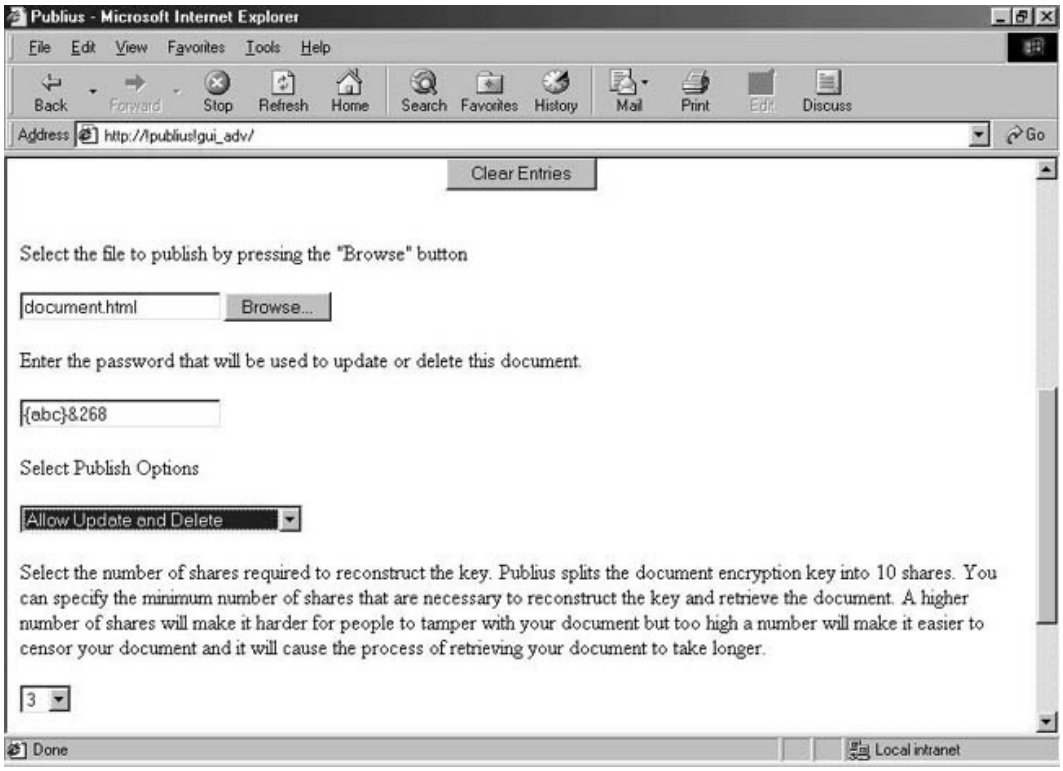

Figure 4.4: **Publius Publish Screen** In addition to the password, a publisher can specify the update and delete options that can be applied to the document. The number of shares needed to reconstruct the document can also be set.

This simplified design is clearly more efficient than the previous design as less servers need to be contacted—there is no need to contact additional servers to acquire the additional shares that make up the decryption key. Note that due to the presence of the version number field in the original Publius URL both schemes can be accommodated in future versions of Publius. The version number determines which retrieve algorithm is appropriate for the URL.

4.8 Limitations and threats

In this section we discuss the limitations of Publius and how these limitations could be used by the adversary to censor a published document, disrupt normal Publius operation, or learn the identity of the publisher of a particular document. Possible countermeasures for some of these attacks are also discussed.

4.8.1 Share deletion or corruption

As described in Section 4.2, when a document is successfully published a copy of the encrypted document and a share are stored on n servers. Only one copy of the encrypted document and k shares are required to recover the original document.

Clearly, if all n copies of the encrypted file are deleted, corrupted or otherwise unretrievable then it is impossible to recover the original document. Similarly if $n - k + 1$ shares are deleted, corrupted or cannot be retrieved it is impossible to recover the key. In either case the published document is effectively censored. This naturally leads to the conclusion that the more we increase n, or decrease k, the harder we make it for an individual, or group of individuals, to censor a published document.

4.8.2 Update file deletion or corruption

As stated in Section 4.5, if a server receives a request for a published document that has an associated update file, the URL contained in that file is sent back to the requesting proxy.

We now describe three different attacks on the update file that could be used by

an adversary to censor a published document. In each of these attacks the adversary has read/write access to all files on a server hosting the published document P, he wishes to censor.

In the first attack we describe, P does not have an associated update file. That is, the publisher of P has not executed the update operation on *P's* URL. The adversary could delete P from one server, but this does not censor the content because the document is available on other servers. Rather than censor the Publius content, the adversary would like to cause any request for P to result in retrieval of a different document, Q, of his choosing. The Publius URL of Q is Q_{url}. The adversary now enters Q_{url} into a file called "update" and places that file in the directory associated with P. Now whenever a request for P is received by the adversary's server, Q_{url} is sent back. However, a single Q_{url} received by the client does not fool it into retrieving Q_{url}. Therefore the adversary enlists the help of several other Publius servers that store P. The adversary's friends also place Q_{url} into an "update" file in P's directory. The adversary succeeds if he can get an update file placed on every server holding P. If the implementation of Publius only requires that k shares be downloaded, then the adversary does not necessarily need to be that thorough. When the proxy makes a request for P, if the adversary is lucky, then k matching URLs are returned and the proxy issues a browser redirect to that URL. If this happens the adversary has essentially censored P by essentially rerouting the request to a different document. This motivates higher values for k. The *update* flag described in Section 4.7.1 is an attempt to combat this attack. If the publisher turned the update flag off when the document was published then the Publius client interpreting the URL will refuse to accept the update URLs for

the document. Although the document might now be considered censored, the reader is not duped into believing that an updated file is the originally published document or a publisher authorized update of it.

In the second attack, P has been updated and there exists an associated update file containing a valid Publius URL that points to a published document U. To censor the document, the adversary must corrupt the update file on $n - k + 1$ servers. Now there is no way for the reader to retrieve the file correctly. If the adversary can corrupt that many servers, he can censor any document by simply deleting the shares of the decryption key. This motivates higher values for n and lower values for k.

4.8.3 Denial of service attacks

Publius, like almost all web services, is susceptible to denial of service attacks. An adversary could use Publius to publish content until the disk space on all servers is full. This could also affect other applications running on the same server. We take a simple measure of limiting each publishing command to 100 KB and allowing each server administrator to limit the number of published files that the server will store. A better approach would be to charge for space using some anonymous e-cash system.

An interesting approach to this problem is a CPU cycle based payment scheme known as Hash Cash [7]. The idea behind this system is to require the publisher to do some work before publishing. Thus, it requires more work and time for the adversary to completely fill the server's disk. Hopefully, the attack can be detected before the disk is full. In Hash Cash, a publisher wishing to store a file

on a particular server first requests a challenge string c and a number, b, from that server. The client must find another string, s, such that at least b bits of $H(c \cdot s)$ match b bits of $H(s)$ where H is a secure hash function such as MD5 and "$\cdot$" is the concatenation operator. That is, the client must find partial collisions in the hash function. The higher the value of b, the more time the client requires to find a matching string. The client then sends s to the server along with the file to be stored. The server only stores the file if $H(s)$ passes the b bit matching test on $H(c \cdot s)$.

Another scheme worth considering is to limit, based on client IP address, the amount of data that a client can store on a particular Publius server within a certain period of time. While not perfect, this raises the bar a bit, and requires the attacker to exert more effort. The IP addresses would only be stored for a short period of time—permanently storing the addresses could allow an adversary to track publish requests if he compromised the server. However, one would expect publishers to use an anonymizing proxy or other anonymizing technology to publish documents. Under these circumstances the reader's identity would be protected and the publishing limit would be placed upon the IP address of the anonymizing proxy.

We have not yet implemented either of these protection mechanisms. Other schemes designed to thwart denial of service attacks are described in [24, 44, 68].

4.8.4 Threats to publisher anonymity

Although Publius was designed as a tool to be used for anonymous censorship-resistant publishing, there are several ways in which the identity of the publisher

could be revealed.

Obviously if the publisher leaves any sort of identifying information in the published file he is no longer anonymous. Publius does not anonymize hyperlinks in a published HTML file. Therefore if a published HTML page contains hyperlinks back to the publisher's web server then the publisher's anonymity could be in jeopardy.

Publius by itself does not provide any sort of connection based anonymity. This means that an adversary eavesdropping on the network segment between the publisher and the Publius servers could determine the publisher's identity. If a server keeps a log of all incoming network connections then an adversary can simply examine the log to determine the publisher's IP address. To protect a publisher from these sort of attacks a connection based anonymity tool, such as those described in Section 3.2, should be used.

4.8.5 "Rubber-Hose cryptanalysis"

Unlike Anderson's Eternity Service [4] Publius allows a publisher to delete a previously published document. An individual wishing to delete a document published with Publius must possess the document's URL and password. An adversary who knows the publisher of a document can apply so called "Rubber-Hose" Cryptanalysis [56] (threats, torture, blackmail, etc) to either force the publisher to delete the document or reveal the document's password. If the file in question was published with the "Do Not Delete" option (See Section 4.4) then this password-based attack will not work as the publisher cannot delete the file.

Of course the adversary could try to force the appropriate server administrator,

or other relevant parties, to delete a particular document or remove the hosting server from the network. However when the published document is distributed across servers located in different countries and/or jurisdictions such an attack can be very expensive or impractical.

4.9 Publius Live Trial

In early August 2000 we ran a 2 month trial of the Publius system. A month prior to the start of the trial we sent a request for participation letter to security and cryptography related Usenet news groups and mailing lists. We requested that server volunteers meet the following 4 conditions:

1. Be connected to the Internet 24 hours, 7 days a week
2. Have the Perl scripting language and a web server installed
3. Set aside 100 MB of disk space for the storage of Publius content
4. Be willing to run the Publius server CGI script (approximately 300 lines of Perl code).

4.9.1 Server Volunteers

Exactly 100 individuals volunteered their servers. Although the requirements to run Publius were relatively minor we asked the prospective volunteers to install a small test CGI script that spoke an abbreviated version of the Publius protocol. The purpose of this script was to insure that the proper Perl libraries were installed, and to allow us to actually attempt communication with the server. Fifty three

individuals successfully installed the test script. We sent the full Publius server script to these individuals. We are not sure why only 53 of the 100 individuals installed the script. Only about 4 of the remaining 47 individuals ever contacted us again. Misconfigured servers accounted for most of the problems encountered during the test script phase of the project. Incorrect file permission settings came in a close second.

Eventually 39 individuals successfully installed the full Publius server script. These server volunteers made up the first Publius server list. Over the next several weeks another 6 individuals successfully installed the server script and were added to the list. These servers were added to the to the list as the trial progressed. As of December 2000, Publius servers exist in several countries including Denmark, England, the Netherlands, Poland and the United States.

4.9.2 Proxy Volunteers

In addition to server volunteers we also asked for proxy volunteers. Proxy Volunteers run the client software on their computer for the express purpose of allowing others to use it. We call these proxies, remote proxies. The client software was written to support multiple users at once—just like a traditional server process. Installing and running a proxy is much more involved than just running a server. The proxy is a program that is meant to be run all the time—just like a Unix daemon process. In addition, the proxy required installing the freely available crypto++ library [60]—no easy feat on some operating systems involved in the trial. Initially 33 individuals volunteered to host Publius proxies. Of these only 8 installed the software. The reason for this rather low installation rate had to do

mainly with performance, security and network bandwidth considerations. Only three of the 33 individuals had problems installing crypto++. The rest were worried that hosting a Publius proxy would either saturate their network connection, degrade server performance or open their server up to some sort of attack. As of December 2000 there were 11 Publius proxies available for individuals to connect to. The three additional proxies were added after the trial began.

The main problem with using a remote proxy is that you must completely trust it. The remote proxy sees the Publius commands and associated parameters. This means that a malicious proxy can refuse to publish or retrieve certain material. In addition, retrieved or published material can be arbitrarily modified by the proxy before it is sent to the appropriate server or browser. The easiest way to avoid these problems is to run the proxy locally. However, if this is not an option, one can try to publish or retrieve from multiple proxies, using one proxy to verify the actions of the other.

4.9.3 Publius Usage

The Publius software was purposely distributed without any logging capability built in. This made it difficult to determine the extent to which Publius was being used. Therefore at the end of trial we sent a questionnaire to the original 39 Publius server administrators asking them for counts of the various files they were storing. Shell commands were provided that made the counting procedure very easy. Twenty seven individuals responded to the questionnaire. The highest reported number of published files was 92. The average was 60. An average of 4 documents per site were published with the "Do Not Delete" option. By default,

Publius publishes a document so that it can be deleted or updated by the publisher. As this is the default option the small number of files published with the "Do Not Delete" option is not surprising. All the administrators reported either light or steady usage—several mentioned they had even forgotten that the software was installed. Four administrators reported keeping usage logs. These logs indicated that far more people attempted to read Publius material than published. A web page containing hyperlinks to some previously published Publius content was made available on the Publius web site so that individuals could test the system. This is probably the reason for the rather high read to write ratio.

Almost all Publius server administrators expressed willingness to continue hosting content after the trial had ended. Several new hosts and proxy volunteers were added after the trial had ended.

4.9.4 Issues Raised

A few individuals had problems with using Publius because of firewalls. Publius proxies by default run on port 1787—the year that articles contained in the Federalist Papers first appeared. An individual reported being unable to connect to proxies because of corporate firewall policy of not allowing connections to ports other than the "well-known" ports. In addition firewalls prevented two individuals from volunteering to host Publius proxies as incoming connections to ports other than port 80 were not allowed. Other individuals couldn't use the proxy because ISP or corporate policy dictated that all web traffic be funneled through a specific proxy, eliminating the option of specifying an alternate proxy.

Both of the previously described problems could be solved by replacing the

proxy with a CGI script. The requested operation and parameters would be sent as parameters to the CGI script which would perform the corresponding Publius operation (retrieve, publish, delete and update). While this not as convenient as the proxy it does clear up the firewall and proxy problem as all traffic goes over the standard HTTP port 80.

4.10 Discussion

Publius can be improved in a number of ways. The design proposed in Section 4.7.8 would improve Publius's performance and manageability. Fewer network connections would be needed to read a document and reader's would no longer need the Publius server list. Work remains to be done on deterring denial of service attacks aimed at filling up the disk drives of the hosting servers. Some sort of anonymous e-cash payment scheme is one possible solution. Krawczyk [36] describes how to use Rabin's information dispersal algorithm to reduce the size of the encrypted file stored on the host server. I believe this scheme could be easily and fruitfully added to Publius.

Publius's main contributions beyond previous anonymous publishing systems include an automatic tamper checking mechanism, a web URL-based method for updating or deleting anonymously published material, and methods for publishing mutually hyperlinked content in a censorship-resistant publishing system.

Chapter 5

Tangler

Tangler is a peer-to-peer-based censorship-resistant publishing system. Like other censorship-resistant publishing systems it attempts to prevent the adversary from altering or completely removing a published document. However, Tangler uses a number of unique mechanisms to accomplish this. Chief among these is its publication mechanism, called entanglement, and its self-policing storage network. The benefits of entanglement include the automatic replication of previously published content and an incentive to audit the reliability with which servers store content published by other people. In part through these incentives, the self-policing network identifies and ejects servers that exhibit faulty behavior. As in Publius, Tangler servers do not know the content of documents that are being stored. However, while in Publius, each server stores encrypted documents, Tangler takes this a step further—servers store blocks that have no meaning unless they are combined, in well defined ways, with blocks from other servers. Combining the blocks in different ways creates different documents.

The first half of this chapter describes the high-level design of Tangler. The second half describes the implementation and performance of Tangler.

5.1 Design Goals

The following design goals were important in shaping the design of Tangler.

Dynamic server participation. New volunteer servers should be allowed to join and participating servers allowed to leave. These servers will store the blocks of published documents.

Document and block decoupling. The blocks stored on the servers should not be associated with any particular document. Rather, each block should able to be used in the reconstruction of several documents. The particular document formed depends on how the various blocks are combined. Therefore, the servers are just holding blocks—meaningful documents are created by individuals who request and combine blocks from the various servers.

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. Once a document has been published, only its publisher can update it.

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

Publishing limit. A server can publish no more than a certain fraction of what he is willing to store. This is intended to limit the damage done by 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.

Useful work outweighs adversarial behavior. 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 still performs useful work before being allotted full access to the system. Once adversarial behavior is detected the server is scheduled for ejection from 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 be able to 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.

Dynamic Address Space. With high probability, no minority set of servers can control access to a particular document. This helps to minimize attacks in which adversarial servers are added to the system for the purpose of targeting a specific document.

5.2 Tangler Overview

This section describes the high-level design of Tangler. Tangler was designed to consist of a small group of servers (≤ 100), called the Tangler servers, operated by volunteers around the world. Over time, servers will leave and new ones will join. Like Usenet, we assume that the servers have large storage capacities and good network connectivity, and each server stores an important fraction of the content of the entire system.

There is consensus among all participating servers as to which other servers are members of Tangler. In addition, each participating server knows the public

key and capacity of every other Tangler server. This public key is needed to authenticate messages exchanged between participating servers.

The Tangler servers respond to requests from clients. *Publishers* are clients that can store data blocks on the servers. *Readers* are clients that request data blocks from servers. The reader combines these data blocks to form a document.

To publish a new document, d, the publisher first retrieves random data blocks from the servers. These data blocks belong to previously published documents. The publisher then divides document d into equal sized data blocks and combines these blocks with the previously retrieved data blocks. The process of combining the data blocks of new documents with those of previously published documents is called *entanglement*. Entanglement has several nice properties that include making the replication and verification of previously published content an inherent part of the publication process. The output of the entanglement process is a set of blocks. These blocks, which can be used to reconstruct one or more documents, are stored on the Tangler servers. The servers make the blocks available to clients.

In order to store blocks on the servers, publishers need to obtain storage tokens. Storage tokens can be acquired from the servers. These tokens are dispensed at the discretion of servers. In the current version of Tangler, publisher's authenticate with a particular server. However, different servers can have different policies. For example, some servers could charge e-cash for tokens or require that the client perform some sort of work [44, 68]. Once in possession of a token, the publisher can anonymously store a block on a server. The use of storage tokens prevents an adversary from attempting to completely fill all available storage—a storage flooding attack.

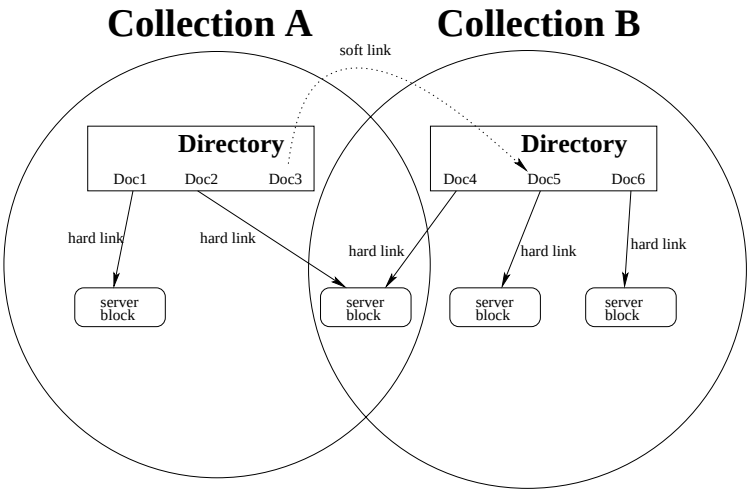

Figure 5.1: Collections can be made up of files (hard links) and links to other collections (soft links)

Each Tangler server monitors all other servers in order to determine which, if any, are faulty or adversarial. An adversarial server is one that fails to follow, or attempts to subvert, the Tangler protocols. If a majority of servers agree that a particular server is faulty or adversarial then that server is ejected from the system. All Tangler servers will refuse to communicate with a server once it has been ejected. This monitoring and ejection mechanism forms the basis of Tangler's self-policing mechanism.

5.3 System and Adversary Model

As previously stated, Tangler is composed of a collection of volunteer servers. We will call these servers the Tangler servers. These servers communicate over the Internet. Some of the volunteer servers may fail, potentially in a Byzantine way.

Byzantine servers may collude in various ways in an attempt to disrupt or otherwise break the system or its various protocols. For example, Byzantine servers could attempt to limit access to specific documents or try to discover who published or read a particular document. Nodes that fail in a benign or Byzantine manner are said to be faulty. In a networked environment it is certainly possible that a particular server may be non-faulty but, due to a fault in the communication links, be unable to communicate with any other server. As all other participating servers cannot differentiate between a faulty server and one that is out of contact, we treat communication link failures as server faults [33]. Non-faulty nodes have not failed and follow the Tangler protocols.

In order to fulfill the security and robustness properties of Tangler we assume that the majority of Tangler servers are non-faulty and that failures are uniformly distributed. We believe this is a reasonable assumption due to the fact that volunteer servers reside on different network segments and are managed independently.

Tangler uses TCP, a reliable transport, to provide a virtual connection between any two communicating hosts. However, an adversary can eavesdrop or attempt to interfere with communication by modifying or dropping packets in transit or replaying old packets. We assume an adversary can delay messages being sent to or sent by a particular node. However, we assume an adversary cannot deprive a server of communication (sending and receiving of messages) indefinitely.

The Tangler design makes a number of synchrony assumptions that distinguish it from the secure state machine systems described in Section 3.3. In Tangler, published material expires over time and previously used disk space needs to be reclaimed so that new or republished material can be stored. In order to fairly

treat all published material we assume all non-faulty servers keep time to within $m = 5$ minutes of the correct time. As described in Section 5.7.1, this can be accomplished by having every server run NTP [42]. Timing constraints are needed to preserve liveness and to eliminate faulty nodes that attempt to abuse the system by refusing to answer queries made by other participating servers. These timing constraints also place Tangler in the synchronous message passing communication model.

5.4 Tangler document collections

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 storage on the Tangler servers. These fixed-size blocks are constructed so that they can potentially belong to multiple documents, the property we call entanglement. Section 5.6 describes the algorithm used to entangle blocks. Section 5.7 describes the design of Tangler's self-policing mechanism that can survive certain flooding attacks and the existence of corrupt servers.

5.4.1 Collections

Every document in Tangler is published as part of one or more *collections*. A collection is a group of documents that are published together by the same person. Collections are published anonymously and named by a public key. The person who published a collection can later update it. Thus, anonymous collections may build reputations (the associated public key provides the publisher with linkable

anonymity [30]). A collection can consist of a single document, multiple documents, or links to documents in other collections. One can think of a collection as a directory containing a group of files, subdirectories, and links. For example, a collection may contain a group of technical reports, the files that make up a web site, or an index of other collections.

Each collection is named by a public key, K. The corresponding private key is used to sign the collection. The signing process is described in more detail in Section 5.5.

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*. Links to documents within a particular collection are known as *hard links*. Two different 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 naming public key to a new set of documents, the other hard linked 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. See Figure 5.1.

5.4.2 Hash trees

Tangler makes extensive use of the SHA-1 [45] cryptographic hash function. SHA-1 is a collision-resistant hash function that produces a 20-byte output from an arbitrary-length input. Tangler treats SHA-1 hashes as unique, verifiable identifiers for published document blocks.

Tangler also relies on Hash Trees [40]. Hash trees allow one to specify or commit

to large amounts of data with a single cryptographic hash value. Using hash trees, an individual 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 it's 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. This proof is called the certifying path [41].

5.4.3 Server Blocks

In order to publish a collection, C, one runs a publish program that takes as input a public/private key pair, and the group of files, directories and links that make up the collection. The program breaks the files into fixed-sized data blocks and *entangles* them with blocks that have been randomly retrieved from the various Tangler servers. These randomly chosen blocks are actually the blocks of previously published documents. This *entanglement* procedure produces new blocks that are stored on the servers. Section 5.7 describes how this random fetching is implemented. Finally, using the private key, it signs a collection root data 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 publisher of C needs to assure availability of both the blocks just created and the ones with which her collection is entangled. Thus, republishing others' documents is an inherent part of publishing.

The entangled output blocks of the publish program are suitable for storage on the Tangler servers. We call these blocks *server blocks* to differentiate them from the file *data blocks* that were initially created by the publish program. Tangler names server blocks by the SHA-1 hash value of the block's content. The collection's SHA-1 hash values are recorded in metadata structures (inodes, directories, and collection roots) so that the retrieval program can later reconstruct the collection from the component blocks.

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 named by a public key. This public key also names the collection's root block, and therefore must be present within that block. Thus, the Tangler servers support the retrieval of blocks by public key as well as by SHA-1 hash value. As a collection can be updated, two or more collection roots with the same public key, but different SHA-1 hash values, may appear on the servers. To disambiguate the blocks, a version field is present within all collection roots. The version field is incremented each time a collection is republished.

5.5 Publish

The first step of the publish program is to entangle each member file in a collection. Each file is split into fixed-size 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 5.6.2. The entanglement algorithm takes as input

```
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 Tangler Servers
          p2=random server block selected from Tangler Servers
          (e1, e2)=entangle(b, p1, p2)
          store server blocks (p1, p2, e1, e2) on Tangler Servers
          r=random_permutation(p1, p2, e1, e2)
          record b's dependency on (r) in i
      /* entangle the inode */
      p3=random server block selected from Tangler Servers
      p4=random server block selected from Tangler Servers
      (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
  c.name=pk
  c.version=1
  digest=SHA_1(name, version, filenames, inodes)
  c.sig=sign(digest, sk)
End Publish
```

Figure 5.2: Publish Algorithm

two random blocks from the Tangler servers and the data block from the file being published. It outputs two new server blocks, which when combined with the previously randomly selected blocks can reconstitute the original data block. Thus, for every data block a publisher entangles, she becomes interested in ensuring the availability of four server blocks in the Tangler network. A data block can actually be reconstructed from any three of its four associated server blocks, adding some fault-tolerance (see Section 5.6.3). 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 5.2 shows the pseudocode for the publish algorithm.

A digitally signed collection root is padded to the same size as a server block. Collection roots can therefore become entangled just like 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 hashes (the result of entangling the collection root). 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 even if it cannot be found, via public key lookup, on the Tangler servers.

5.5.1 Reconstruction

In order to reconstruct a collection, a collection's server blocks must be retrieved from the Tangler servers. Server blocks retrieved from the servers are tamper-checked by simply computing the SHA-1 hash of the block's content. This hash value must be the same as the hash value used to retrieve the block. Similarly, the signatures on collection roots are verified.

Once in possession of a verified collection root, a reader can attempt to reconstruct any of the files stored in (or named by) the collection. 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. In our current entanglement scheme only three fourths of the server blocks listed in the inode are needed. Once the necessary server blocks are retrieved, the reconstruction algorithm (Section 5.6.3) is applied to the blocks.

5.5.2 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 with a higher version number. Files that have not changed since the previous version of a collection do not need to be reentangled. If a server holds two or more collection roots possessing the same public key, the server must return the root with the latest version number.

5.6 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.

5.6.1 Secret Sharing

Shamir [59] 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 fewer than k pieces reveals nothing about the secret. The pieces are called shares or shadows. The secret and shares are elements over 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 one 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.

5.6.2 Entanglement Algorithm

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.

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_0. We now randomly select b ($b = 2$ in the current implementation of Tangler) server blocks from the servers. Each of these selected blocks consists of an (x, y) pair. We then perform Lagrange interpolation on the b server blocks and f_0. This forms a polynomial, p, of degree b. We can now evaluate p at random nonzero integers to obtain new server blocks. Each new server block is of the form $(x, p(x))$, where x is the nonzero random integer value. We then store these new server blocks on the servers. One could conceivably store f_0 on the servers as well, however one of the goals of Tangler is to dissociate stored blocks from documents. Storing f_0 would be in direct conflict with this goal as f_0 clearly belongs to a specific document. In addition, in a censorship resistant system one would usually not store f_0 as it consists of plaintext and therefore is an easy target of the censor. The server blocks, being shares, are information-theoretically unrelated to all other published blocks.

This procedure must be done for every data block of the file to be published. An inode is created 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.

In the current implementation of Tangler each x value is an element over the finite field $GF(2^{16})$. For efficiency, each y value (the data block) is processed in 16 bit chunks, each chunk is treated as an element over the finite field $GF(2^{16})$. Therefore each server block consists of a single x value and many y values. The entanglement implementation is fully described in Section 5.8.4.

5.6.3 Reconstruction Algorithm

In order to reconstruct a data block of a file we need to retrieve at least k of the appropriate shares (server blocks). 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. This is repeated for every data block of the files and inodes that we wish to reconstruct.

5.6.4 Benefits and Limitations

Entanglement has three beneficial consequences. The first is that it promotes the replication of blocks of previously published documents (the blocks stored on the servers). A publisher could easily generate random 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, replicating and monitoring availability of 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 means that there is a decoupling between blocks and documents. A particular block can be used create many documents. Servers are simply hosting blocks that can be combined in different ways to produce different documents.

The third benefit of entanglement is the incentive to cache 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.

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 a potential limitations of the technique.

The benefits of entanglement don't come without some cost. The entanglement algorithm does increase the amount of time it takes to publish and re-form a document. Section 5.8.4 describes the performance of the entanglement algorithm. In addition, there is a storage and bandwidth overhead associated with the entanglement mechanism. For each data block that is entangled, the publisher needs to assure that at least three server blocks are stored on the servers. To achieve a degree of of fault tolerance at least four server blocks need to be stored on the servers. Two of the four server block are new. Therefore, entanglement doubles the storage and triples the bandwidth of the reader.

5.7 Tangler network

The Tangler network consists of a collection of volunteer block servers, called the Tangler servers. These servers store and serve the blocks of published collections. In addition, each server monitors the other Tangler servers to ensure that they are complying with the Tangler protocols. This monitoring forms the basis of the self-policing network that attempts to identity and eject faulty or malicious servers.

5.7.1 Rounds

Server blocks published on Tangler servers do not persist indefinitely. The blocks expire after a predetermined amount of time. In Tangler, time is divided into rounds. A round consists of a 24 hour period. Therefore we say that published server blocks persist for a specified number of rounds. As stated in Section 5.3, all non-faulty servers have their local clocks synchronized to within 5 minutes of the current time. This requirement can be fulfilled by having the servers run the NTP daemon [47]. NTP security issues are addressed in [42, 43]. A server that does not synchronize its clock in this manner will be out-of-step with the other Tangler servers—a problem that could lead to the ejection of the server. We arbitrarily start round zero at January 1st, 2002 (12:00AM) UTC. A new round starts every day at 12:00AM UTC. This implies that any participating server can determine the current round number without the need to interact with other servers. We call 14 consecutive rounds an epoch.

5.7.2 Tangler Servers

The Tangler model assumes a collection of servers around the world, run by volunteers opposed to censorship. Clients publish documents by anonymously submitting server blocks to the Tangler servers. Each server has a long-lived public key, used for authentication. Servers can communicate with each other both directly and anonymously (using an anonymous communication channel). Different servers may dedicate different amounts of storage to Tangler, but each publicly commits to providing a certain capacity. There is a general consensus on the public keys and capacities of available servers. We describe the certification, consensus and

communication process in the following subsections.

In Section 5.1 we stated that one of the goals of Tangler was to allow new servers to join and to allow old servers to leave. A server may wish to leave if, for example, the server's administrator no longer wishes to participate. Ejection of a server requires a majority of the participating servers to agree that a server is faulty. This implies that a majority of the Tangler servers must be non-faulty if we wish to prevent a group of cooperating faulty servers from ejecting non-faulty servers—effectively taking control of the Tangler network. Therefore we cannot simply allow all volunteers to join the Tangler network at once. Instead of automatic join mechanism, we propose a two-layered membership system that allows a server to fully participate in the Tangler network once it has completed a probational service obligation. We describe this membership system in the next section.

5.7.3 Join and Leave Protocols

In a Byzantine environment in which every node is a possible adversary a fundamental problem is that of resource discovery. More specifically, the problem is one of learning the addresses of the currently running Tangler servers. An adversary could simply set up a collection of servers in an attempt to spoof [27] the Tangler network. Clearly, this problem has no satisfactory solution as any information sources could be adversarial. Therefore we simply assume that the list of Tangler servers can be obtained from a trusted Tangler server, web server, usenet group, or some other out-of-band means. Similar techniques have been successfully used for anonymous remailers. Once in possession of this information a node p may attempt

to join the Tangler network. Define the *view* for round r as the list of servers (and associated public keys) that are members of the Tangler network during round r. Once in possession of the view for round r, node p can attempt to join the Tangler network by contacting each server in turn and sending its listening port number, public key and the number of megabytes of disk storage it will be donating to the Tangler network. This join message must be signed by the corresponding private key. All servers join as probational servers. Probational servers simply store server blocks and respond to client queries for these blocks. A probational server earns the right to become a full-fledged Tangler server after it has been a probational server for two consecutive epochs and has not been ejected, by the Tangler servers, due to faulty behavior. The block storage process is described Section 5.7.7.

A Tangler server can leave the network gracefully by sending a signed *leave* message to all Tangler servers. A server may send this message if the server's administrator no longer wishes to participate in Tangler.

5.7.4 Faulty Servers

In Tangler we want to handle servers that exhibit either fail-stop or Byzantine behavior. However, a server exhibiting Byzantine faults is usually far more detrimental to the system than a server exhibiting fail-stop behavior. Therefore we wish to identify and isolate these servers quickly. Some forms of Byzantine behavior can be identified by a succinct proof. A succinct proof usually takes the form of two contradictory messages signed by a particular server. Examples of succinct proofs are described in Section 5.7.11). Once collected, this proof is sent to all Tangler servers. Each server then validates this proof and then schedules the faulty server

for ejection.

Other forms of misbehavior require witness servers to certify faulty behavior. A witness server is just another Tangler server. Witnesses are enlisted when a particular server, f, fails to properly answer a request within a specific amount of time. The witness re-issues the same request. If the witness request succeeds, the result is sent back to the node that enlisted the witness. As Tangler requests are idempotent, any number of witnesses can be enlisted to re-issue the request. If the witness requests fail, then the witness internally flags f as faulty. Once a majority of the servers have certified that a particular a server is faulty, it can be ejected. The ejection process is described in Section 5.7.5. This ejection mechanism also applies to probational servers.

A server that doesn't respond to a request may simply be down for a short amount of time. Therefore an unresponsive server is not immediately considered for ejection. An unresponsive server can only be ejected once it is unresponsive for 3 or more rounds during a window period of 14 rounds (the length of an epoch). Unresponsive probational servers are ejected in the same manner.

5.7.5 Membership Changes

At the midpoint of every round each Tangler server has the opportunity to vote-out other servers. If a majority of servers vote-out a particular server, that server is essentially ejected from the system—other Tangler servers will no longer communicate with the voted-out server or accept messages that it has signed. However, in an environment where servers can be adversarial an algorithm is needed to prevent an adversarial server from sending different votes to different servers. This

is accomplished using a Byzantine agreement algorithm [22, 37]. In a Byzantine agreement algorithm a node, called the leader, sends a message (his vote) to all other nodes. These nodes, in turn, send messages among themselves to verify that the leader did indeed send each of them the same vote. If these nodes determine that the leader has sent different votes to different nodes then the leaders votes are discarded. In Tangler, rather than wait for each node to take its turn being the leader, all servers execute the agreement algorithm concurrently. That is each Tangler server acts as both a leader (voter) and verifier of votes. The implementation of this algorithm is discussed in Section 5.8.5.

Rather than simply voting-out servers, each leader votes on which servers should remain a member of the Tangler network. Only servers that receive a majority vote remain part of the Tangler network. This design was adopted because it allows membership changes to be made independent of previous rounds—a server that misses the votes of other Tangler servers due to a communication outage will learn the membership of the Tangler network during the next round. If we adopted a vote-out strategy then a server that missed the votes would need to contact other servers to determine which servers were ejected during the previous round.

5.7.6 Block-to-server mapping

The Tangler network uses consistent hashing [35] to map blocks to servers. In consistent hashing the 160-bit output of the SHA-1 hash function is mapped onto points of a circle. The circle is made up of 2^{160} addressable points. Each server block is assigned the point on the circle corresponding to the SHA-1 hash value of its content. Collection root blocks are assigned the point on the circle corre-

sponding to a hash of their public key. Each server is also assigned a number of points on the circle proportional to its stated capacity—one point per 100 MBytes of storage. A server's points are calculated from its public key, K, and the current round number, d. We want each server to have, with very high probability, a new set of points every 14 rounds(one epoch). Therefore each 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, during each round roughly 1/14 of a server's points change.

Each block is stored on the servers immediately clockwise from it on the circle. Figure 5.7.6 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 server 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 round number are well-known information, anyone can compute the current set of points on the circle. To retrieve 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, faulty, or simply does not have the block.

Note that it is up to clients to publish blocks on the servers where readers will look for those blocks. Tangler does not prevent users from publishing blocks elsewhere. However, since storage priveledges are limited (see Section 5.7.7), it

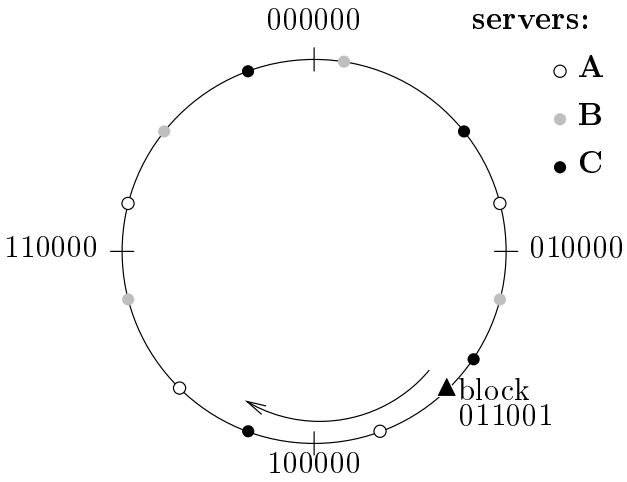

Figure 5.3: Use of consistent hashing to distribute server blocks among servers.

is in a client's best interest to store blocks wisely. As server points move around the circle, users will need to republish server blocks. The main impetus behind this address space design is to prevent an adversary from dominating parts of the address space for any extended period of time. As time and Tangler server membership changes, so does the address space. The addition or subtraction of a server can change the successor server for a particular server block or collection root.

5.7.7 Storage Tokens

Publishers need *storage tokens* to store blocks on the Tangler servers. Each storage token is good for the storage of one server block for one epoch (14 rounds) from the round that the token was first issued. Storage tokens are issued by servers.

Recall that every Tangler server donates disk space for use by Tangler. This disk space is used to store server blocks. In return for the disk space donation,

each server is entitled to a number of storage tokens from every other server. The number of storage tokens a server is entitled to is proportional to its disk space contribution. A server that has contributed p percent of the total disk space available to Tangler is entitled to p percent of the storage tokens available on every other server. The number of storage tokens available on a specific server is simply the amount of disk space the server has donated divided by the size of a server block (16KB in the current version of Tangler). For example, suppose the total disk contribution of all the servers is 100 GB and that server S has contributed 2 GB. Then server S is entitled to 2/100 of the storage tokens available on each of the Tangler servers. Assume server B has donated 800 MB of disk space to Tangler. Server S is entitled to $2 \cdot 800/100 = 16$ MB worth of storage tokens on server B. Sixteen MB represents 1000 storage tokens (16 MB/16KB = 1000).

Storage tokens are nothing more than digital signatures of server block names (SHA-1 hashes), blinded [16] so that servers do not know for what server blocks they are issuing tokens. At the beginning of every round, each server, S, creates a round-specific temporary public key, KT_S, for signing storage tokens. The server certifies the key for use in round d with its long-lived public key K_S, producing $\{\text{TOKEN-KEY}, d, KT_S\}_{K_S^{-1}}$. A server must certify exactly one such temporary key per round and must not reuse keys; two round d TOKEN-KEY certificates bearing different public keys is succinct proof of a faulty server.

5.7.8 Publishing Clients

For the purposes of publishing, we classify clients as either server-based clients or external clients. Server-based clients are run by individuals or organizations that

are hosting a Tangler server. As every Tangler server is entitled to a certain number of storage tokens, these clients have direct access to storage tokens. External clients are not directly affiliated with any server and therefore need to find some server that is willing to give them storage tokens. How a server allocates storage tokens is entirely at the discretion of its operator. The current implementation authenticates clients but other servers might allocate tokens in exchange for e-cash payments. Another might charge hashcash [7]. Another might charge for publication in human time—posing challenges that could not be answered by automatic "spamming" programs [44, 68]. Alternatively, a server might only give tokens to particular pseudonyms, or from members of some organization

5.7.9 Block Storage

Recall that every Tangler server knows the amount of disk space that all other servers have contributed to Tangler. This means that every server can calculate the number of storage tokens it is entitled to and the number of storage tokens it must give to all other servers.

The Tangler block storage protocol is meant to allow a publishing client to anonymously store blocks on the Tangler servers while at the same time prevent adversaries from filing all available storage by anonymously publishing bogus server blocks—a storage flooding attack. The Tangler protocol consists of two phases. In the first phase the publishing client acquires signed storage coupons from a server, S. Each coupon contains the name of a specific Tangler server, R, and the number of storage tokens, n, that the redeemer of the coupon is entitled to. The client presents this coupon to server R. Server R verifies the authenticity of the

coupon by checking S's signature. In addition R makes sure that server S is indeed entitled to at least n storage tokens. Every redeemed coupon reduces the number of tokens that server S is entitled to. If the coupon is valid, server R creates n storage tokens and sends them to the client.

In the second phase of the storage protocol, the client attempts to store server blocks on server R by sending the server blocks, and corresponding storage tokens, to server R. The clients sends the blocks and storage tokens over the anonymous communication channel. In response to a valid storage token and server block combination, server R sends a storage receipt back to the client.

If server R refuses to give the client tokens or refuses to store a block, the client can enlist witness servers who will re-issue the requests on behalf of the client. If server R still refuses to honor the valid storage coupon or storage token then the witness servers will label server R as faulty and schedule the server for ejection.

Storage coupons and tokens are server block specific. Each storage coupon contains a list of n blinded server block names (SHA-1 hashes). The client sends the list of blinded server block names to the server. The coupon-issuing server incorporates these names into the coupon. The blinding prevents the coupon-issuing server from knowing what server blocks the client wishes to store. The same holds true for the server that will grant the storage tokens in exchange for the coupon. The storage tokens are created from the blinded list of server block names stored in the coupon. Once the client acquires the signed storage tokens, the client unblinds them and uses them to anonymously store server blocks.

5.7.10 Storage receipts

Once the client has obtained a storage token—say from server A for block m—the client sends m and the associated unblinded storage token over the anonymous communication channel. Server A, if non-faulty, replies with a signed storage receipt, $\{\text{RECEIPT}, \text{SHA-1}(m), d\}_{K_A^{-1}}$ where d is the current round number and K_A^{-1} is $A's$ temporary private key for round d. If A refuses to issue a storage receipt, the client anonymously enlists Tangler servers as a witnesses. Each witness presents A with m and the storage token. If A still refuses to acknowledge receipt, the witness internally marks the server as faulty. If a majority of the Tangler servers agree that server A is faulty, it will be scheduled for ejection.

5.7.11 Storage commitment

At the end of a round, each server commits to its newly received blocks and sends a signed storage commitment certificate to all other Tangler servers. This commitment consists of the current round number, the root of a balanced hash tree and the number of elements in the tree. The leaves of this tree contain a sorted list of hashes of every block the server is serving. A similar tree is constructed for the public keys naming the collection roots that the server is storing. The root of this tree is also part of the round commitment. A non-faulty server must never sign more than one storage commitment per day. Two distinct commitments signed by the same server for the same round constitute proof of a faulty server.

Storage commitments prevent servers from discarding or suppressing blocks they have agreed to publish. Every block lookup becomes a possible audit of a server's behavior. For example a Tangler server can query another Tangler server,

S, for block m. If server S is non-faulty and is storing block m, it will return a certifying-path that shows m is a member of S's hash tree. If server S does not have the block it will send a certifying-path showing that m is not in S's hash tree. Of course server S could simply have refused to answer or send illegible messages. In this case the witness servers can be employed to attempt to label server S as faulty. If a majority of servers become convinced, after being enlisted as witnesses or through direct contact, that server S is faulty it will be scheduled for ejection.

The hash tree created during storage commitment also plays an important role in the publication process. Clients publishing documents use the hash trees to select random blocks to entangle with. To retrieve a random block, a user first selects a random Tangler server A. The client knows A's block capacity, c_A, and so can simply pick a random n, $0 \leq n < c_A$, and request the server block located in position n in the hash tree. Server A must respond with the server block, b and a certifying path that proves that block b is indeed the n^{th} block. Verification of this certifying path is easy because the tree is balanced and the number blocks in the root hash is known (it is part of the round commitment). Of course, requests for random blocks are sent over the anonymous communication channel so that servers cannot identify publishers by the blocks they have entangled with.

The collection root hash tree in the storage commitment can be used to detect new collections. A search engine could make use of this information to index Tangler collections.

5.8 Implementation and Performance

This section describes the Java (version 1.4) based implementation and performance of Tangler. All performance measurements were made on Linux based 1.1 GHz workstations with 512 MB of memory.

5.8.1 Collection Data Structures

This subsection describes the basic collection data structures that are needed to publish and retrieve collections. These data structures are created during the publication process that was described in Section 5.5.

Server Blocks

The server block is the Tangler unit of storage. Each server block is 16 KB bytes in size. The first two bytes of the server block are reserved and used to encode information about the server block. Currently, only the first byte is actually used to encode information. A server block with a first byte value of one indicates that the server block contains a collection root. As described in Section 5.8.3, the collection root is treated specially by the Tangler servers.

Inodes

During the publication process, each entangled file is broken into equal-sized data blocks. Each of these data blocks is entangled to produce four server blocks. The names of these server blocks are stored in an inode. As you will recall, the name of a server block is the SHA-1 hash of the block's content. Inodes are the same

```
struct initialInode {
        long                    fileSizeInBytes;
        ReconstructionHandles   nextInode;
        ReconstructionHandles   rh[203];
}
```

Figure 5.4: Initial Inode Structure

size as server blocks and can therefore become entangled just like any other server block. Each SHA-1 hash value is 20 bytes in length. Four hash values need to be stored for each entangled data block. We call these four SHA-1 hash values the reconstruction handles. An inode can hold a maximum of $n = \lfloor 16384/80 \rfloor = 204$ reconstruction handles. This means that each inode holds enough reconstruction handles to entangle a file of approximately 3.2 MB in size. To accomodate larger files, we allow one of the reconstruction handles to specify an entangled inode. Tangler has two slightly different inode formats. The first, shown in Figure 5.4, is called the *initial inode*. It contains a field that records the size, in bytes, of the associated file (prior to entanglement), and a field that identifies the next entangled inode. The *fileSizeInBytes* field allows us to determine if the *nextInode* field is meaningful—if the *fileSizeInBytes* field contains a value greater than 3.2 MB then we know the *nextInode* is meaningful. With the addition of these fields, the inode has enough room to store 203 reconstruction handles. The second inode format, shown in Figure 5.5, is used to store reconstruction handles that do not fit in the initial inode. Notice that this second inode format also contains a nextInode field. This allows us to entangle a file of any size—the necessary inodes are all linked by the *nextInode* field. Our inode structures differ from those found in a traditional UNIX system. In particular we have not implemented single, double

```
struct secondaryInode {
        ReconstructionHandles       nextInode;
        ReconstructionHandles       rh[203];
}
```

Figure 5.5: Secondary Inode Structure

and triple indirect pointers [6]. While these pointers are useful for random access of files, their implementation in Tangler would require additional entanglements with little added benefit. Tangler does not currently support random access within entangled files.

Collection Roots

In Section 5.5 we briefly described the collection root as the data structure that held the public key that named the collection, and provided a mapping between file names and inodes. While this is essentially true, the collection root performs a number of other functions. The collection root structure is shown in Figure 5.6. The public key that names the collection is stored in the *namingPublicKey* field. The signature of the data structure is stored in the *signatureOfCollectionRootContent* field.

In order to support different entanglement algorithms, in the future, we specify a transformation name in the collection root. This field tells Tangler the type of transform that was applied to the collection. The default transformation name is "Entanglement.3.4"—the entanglement scheme that requires 3 of 4 server blocks to recover the original data block. The *rootInodeReconstructionHandles* specify the reconstruction handles for the root directory of the collection. Although Tangler does support anonymous publication, a publisher may wish to specify a contact

```
struct collectionRoot {
        PublicKey               namingPublicKey;
        Signature               signatureOfCollectionRootContent;
        integer                 collectionVersionNumber;
        String                  transformationName;
        ReconstructionHandles   rootInodeReconstructionHandles;
        String                  optionalEmailAddress;
        String                  commaSeperatedKeywords;
        String                  collectionDescription;
}
```

Figure 5.6: Collection Root Structure

tangler://collection/*collectionPublicKey*/*version*/*reconstructionHandles*/*filePath*

Figure 5.7: Collection Tangler URL

e-mail address in the *optionalEmailAddress* field. Of course, the publisher could specify an anonymous remailer address in this field. The last two fields in the structure are meant to assist readers and possibly Tangler based search engines. The publisher of a collection can include keywords and a description of the collection. The total size of the collection root cannot exceed 16 KB—the size of a server block. All collection roots are padded to 16 KB to allow them to be entangled just like any other server block.

5.8.2 Tangler URLs

The output of the publish program is a Tangler URL. This URL provides a standard form for specifying and linking to Tangler collections and files. Figure 5.7 shows the structure of a Tangler URL. The Tangler URL first identifies the URL

Num. Blocks	Create Time In ms	Size In Bytes
10,000 (160 MB)	6	17,141
100,000 (1.6 GB)	12	17,326
312,500 (5 GB)	19	17,397
625,000 (10 GB)	19	17,457

Table 5.1: Certifying Path Creation Time and Size (order 10)

Num. Blocks	Creation Time In ms	Size In Bytes
10,000 (160 MB)	12	20,365
100,000 (1.6 GB)	12	20,553
312,500 (5 GB)	17	20,993
625,000 (10 GB)	23	21,613

Table 5.2: Certifying Path Creation Time and Size (order 100)

type—in this case it is a collection. Other types are possible. A Tangler URL can also specify a file or directory. However, since the URL format for these types is similar to the collection URL we will just describe the collection URL. The *collectionPublicKey* field contains the public key that names the collection. The *version* field specifies the version of the Tangler collection. The next field specifies the reconstruction handles needed to the reconstruct collection root if it cannot be found, via public key lookup, on the Tangler servers. Lastly, the *filePath* field specifies an optional path, from the collection root directory, to a file or directory within the collection. This is useful for useful for specifying soft-links.

5.8.3 Hash Trees

Tangler servers accept new server blocks during each round. At the end of every round, each server must commit to these blocks, as well as its previously stored unexpired blocks. The committment process entails building a Merkle Hash Tree, the leaves of which hold the names (SHA-1 hash values) of the server blocks the server is committing to. The commitment certificate actually contains the roots of two different hash trees. One hash tree commits the stored server blocks, the other commits the server blocks that represent collection roots. The second hash tree is necessary in order to ensure that servers are responding properly to queries for public keys (collection roots). A Tangler server that does not have a particular server block or public key must return a certifying path, thereby proving that it has not committed to the server block or public key. This proof mechanism is described in more detail in Section 5.8.7. Both hash trees are constructed from B-trees. Each internal and leaf node of the tree is named by the SHA-1 hash of its content. The order(branching factor) of each tree is 10. The order of the trees is an important issue due to the fact that certifying paths must be built to fullfill certain hash tree queries. The size of the certifying is proportional to $(log_r N)(r-1)$ where r is the order of the tree and N is the number of server blocks in the the hash tree. The height of the tree is $(log_r N)$ and $(r-1)$ is the number of hash tree entries that need to be included in the certifying path, per tree level. Table 5.1 and 5.2 shows the cost, in milliseconds, and size, in bytes, of certifying path creation. Note that the size of the certifying path in these tables include the 16 KB server block that is being certified.

Figure 5.8 shows a hash tree of order 3. Each leaf contains the names (SHA-1

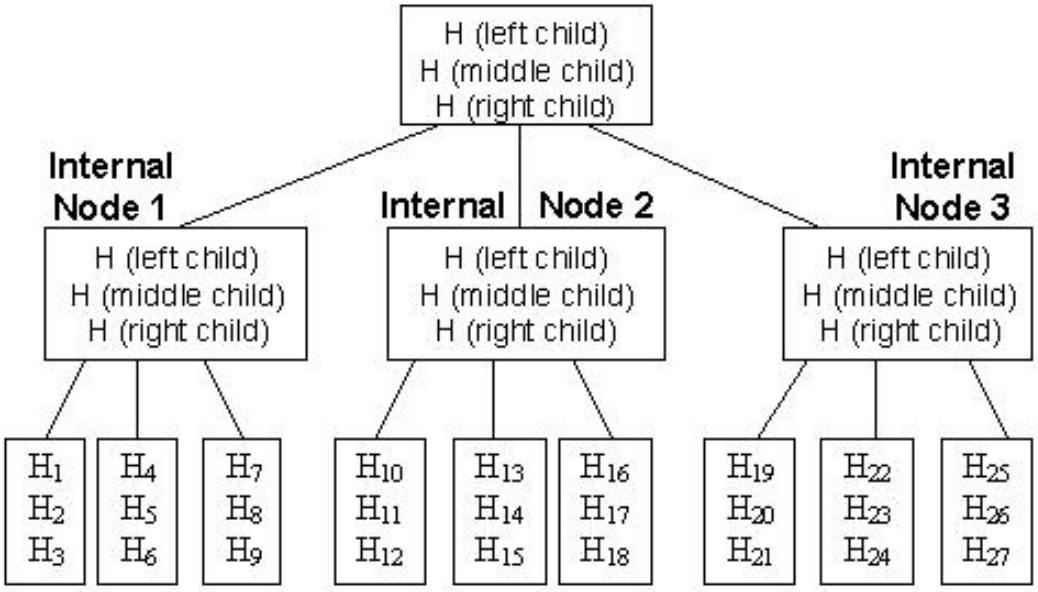

Figure 5.8: **Hash Tree of Order 3** The names (SHA-1 hashes) of the server blocks are stored in the leaves. Each internal node contains the SHA-1 hash of each of its children.

hashes) of 3 server blocks. Let H_i, H_{i+1} and H_{i+2} be the names of the three server blocks stored in a leaf. The tree is built such that the server block names stored in the leaves are in sorted order, $H_i \leq H_{i+1} \leq H_{i+2}$. Each internal node contains the SHA-1 hashes of its children. The SHA-1 hash of the root node is the value that each server must commit to in the commitment certificate.

If a server is called upon to prove that it has indeed committed to the server block with name H_5 then it must supply a certifying path to the verifying agent (client or other Tangler server). The certifying path for H_5 consists of H_4, the server block named by H_5, H_6, the SHA-1 hash of the left and right child of internal node 1, and the SHA-1 hash of internal node 2 and internal node 3. Given these values the verifying agent can derive the SHA-1 hash of the root. This hash

Number Server Blocks	Average Time In Seconds
10,000 (160 MB)	1.02
100,000 (1.6 GB)	8.19
312,500 (5 GB)	25.12
625,000 (10 GB)	51.34

Table 5.3: Hash Tree Build Time In Seconds

value must match the value that the server has committed to in the commitment certificate. Otherwise, the certifying path is invalid.

At the end of each round, each server completely rebuilds the hash tree. A fraction of the blocks stored in each hash tree will expire at the end of each round. These blocks need to be removed to create room for the set of blocks that will be stored during the next roun. Server blocks are stored on disk and placed in different directories based on the first two bytes of their SHA-1 hash values. Each directory contains a file that records the expiration round of every server block in the directory. Every time a server block is stored in one of these directories, the expiration information is updated. As is shown in table 5.3, building the server block hash tree is not a terribly expensive operation, especially in light of the fact that the tree is rebuilt only once every 24 hours.

5.8.4 Entanglement

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 (server blocks),

any 3 of which determine d. Two of these four shares come directly from the Tangler servers. Data block d is first converted into a server block. The x value of this server block is 0 and the 16 KB data block forms the 8192, 16-bit, y values. We perform Lagrange interpolation on these three server blocks (the two from the Tangler servers and the converted data block). Below is the Lagrange interpolation formula [63] for the unique polynomial $a(x)$ of degree at most t. The value t is three in our scheme.

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

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 blocks taken from the Tangler servers and (x_3, y_3) denotes the point from the data block d. All arithmetic is done over the finite field $GF(2^{16})$.

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

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 $GF(2^{16})$. Notice that the x values do not change, only the y values change.

Size Of File	Avg. Time To Entangle	Avg. Time To Reconstruct
682 KB	2.84	0.49
1.6 MB	6.92	1.15
18 MB	67.32	13.71
42 MB	172.77	52.88

Table 5.4: Average Entanglement and Reconstruction Time In Seconds

Therefore we only need to compute the fractional x terms once per server block. 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.

Table 5.4 shows the average time to entangle and reconstruct files of various sizes. The measurements were taken with an on-disk server block cache—all of the random blocks used for entanglement were taken from this cache. As we would expect, reconstruction is much faster than entanglement. Both of these operations are done by a background thread in the client, therefore an individual can continue to use the client software while a publish or retrieve operation is in progress.

5.8.5 Byzantine Agreement

As stated in Section 5.7.5, the Tangler network attempts to eject faulty servers and allows the addition of new servers. This membership change mechanism is democratic—a majority of servers must elect to add or eject a server. This voting mechanism is accomplished using the Dolev-Strong authenticated byzantine agree-

ment algorithm described in [22]. Figure 5.9 shows this algorithm. The algorithm shown is a slightly modified version of the one that appears in [49]. The algorithm has been modified for readability and applicability to Tangler.

The Dolev-Strong byzantine algorithm algorithm is synchronous, requiring that time be divided into a series of phases. The number of phases required by the algorithm is $t + 1$ where t represents the maximum number of faulty nodes the system can tolerate. The Tangler design assumes that a majority of servers are non-faulty, therefore if we have n servers, we need $\lfloor (n-1)/2 \rfloor + 1$ phases. The Internet is an asynchronous communication network which means we must simulate the phases of the algorithm. In order to do so, we must place an upper bound on the amount of time needed by each Tangler server to process and send the various messages generated during each phase. Tangler adopts the simple phase model described in [67]. In this phase model each phase takes $(1+\rho)(\delta_{max} + \delta + \gamma)$ clock time per pulse. This model assumes that the processors have clocks that are δ-synchronized, have ρ-bounded drift and that δ_{max} and γ are upper bounds on the message delay and local computation in a phase, respectively.

As stated in 5.7.1, each Tangler server is synchronized to within five minutes of the current time. This fixes the value of δ to 5 minutes. In a network as large and complex as the Internet determing the appropriate value of δ_{max} is a very difficult task [9, 3]. Message latency between various nodes is affected by such issues as routing, network traffic and network hardware. Therefore we are quite generous in assigning values to the variables in our phase model. If we also assume a δ_{max} and γ value of 5 minutes and 20 second value of ρ then each phase requires 20 minutes. A Tangler network of 100 servers (the maximum envisioned number)

During phase 1:
    Leader server S_0 signs its vote and sends it to servers $S_{1..n-1}$
During phase k, $1 \leq k \leq t+1$:
    when server S_i ($i \in 1, 2, \ldots, n-1$) receives a vote v bearing k signatures do:
        V_i:=$V_i \cup \{v\}$
        if (v is one of the first two distinct votes received) and
        (server S_i has not already sent v to all other servers) then
            add signature to vote v
            send v to all servers that have not yet signed it
At end of phase $t+1$:
    $\forall i \in 1, 2, \ldots, n-1$:
        if cardinality($V_i = 1$) then
            Server S_i records the leader's vote as v
        else:
            Server S_i records the leader's vote the null vote

Figure 5.9: Dolev-Strong Byzantine Agreement Algorithm

would require 51 phases which would take 17 hours. If we begin the byzantine agreement algorithm at six hours into the round we have more than enough time to complete the phases.

At the start of this voting mechanism, each server sends its signed vote to all other Tangler servers. This vote contains a list of all Tangler servers that the voting server believes should remain in the Tangler network. Notice that we are not explicetly voting-out servers, but rather voting-in Tangler servers. This design was adopted because it allows membership changes to be made independent of previous rounds—a server that misses the votes of other Tangler servers due to a communication outage will learn the membership of the Tangler network during the next round. If we adopted a vote-out strategy then a server that missed the votes would need to contact other servers to determine which servers were ejected during the previous round.

The Tangler voting mechanism actually consists of n concurrent invocations of the authenticated byzantine agreement algorithm, where n is the number of Tangler servers. In a single byzantine agreement invocation, a specific node, called the leader, sends its vote to all other participating nodes. During future phases, these nodes exchange signed messages to verify the value sent by the leader. However, in the case of Tangler, we want each server to vote at the same time—each node should be the leader. This is accomplished through n concurrent invocations of the authenticated byzantine agreement algorithm. Each invocation is identified by round and leader, thereby overcoming the impossiblity result of [38]. A server remains in the Tangler network only if it receives a majority of votes. A Tangler server ignores the final results of the vote if it doesn't receive votes from a majority of the servers or if a majority of the servers are voted out. Both of these conditions violate the assumptions under which the Tangler network was constructed.

5.8.6 Anonymous Communication Channel

Some of the protocols used within Tangler require an anonymous communication channel. The current implementation of Tangler utilizes the Java Anon Proxy [12]. The Java Anon Proxy is a "real-time" mix-net designed to support anonymous web browsing. Tangler takes advantage of this by sending all anonymous published messages on top of HTTP. Replies are also sent back using HTTP.

5.8.7 Tangler Client Implementation

The Tangler client software is responsible for publishing, and retrieving collections and contacting Tangler servers to act as witnesses. On startup the client must

have access to a file called the *round-info* file that contains the domain name, port numbers, public key, and disk space contribution size of every server participating in the Tangler network. We assume the client can get this file from some trusted source (web site, usenet news, known Tangler server, etc).

Once the round-info file is read, a number of threads are started. These threads are described below.

Block Manager Thread

The block manager thread is responsible for maintaining the local server block cache. This cache of blocks is used by the client during the publication (entanglement) process. Each server block is used only once during the entanglement process, therefore the block cache must be refreshed periodically. This refresh period is triggered once the number of blocks in the block cache falls below a particular watermark. The block manager requests random blocks from participating Tangler servers. The requests are made in a round-robin fashion. In order to prevent the server from selecting the random block, the block manager thread requests blocks by their position in the server's server block hash tree. For example, suppose Tangler server S is listed in the round-info file as donating enough disk space to hold 1000 server blocks. The block manager thread generates a random number, n, between 0 and 999 and sends this number, in a properly formatted request message, to server S. Server S returns the nth server block in its server block hash tree along with the certifying path for this server block. The certifying path is sent in order to prove that the block is indeed the nth block. This ensures that all server blocks have the same probability of being retrieved and used for en-

tanglement. All client block requests are done over an anonymous communication channel to prevent the server from identifying which client is requesting the block. Tangler servers also periodically request blocks in this same manner. This is done to ensure that other Tangler servers are not refusing to serve specific blocks. A server that repeatedly fails to serve a requested block is marked, by the requesting server, as a candidate for ejection.

Publisher Thread

The publisher thread is responsible for maintaining a list, called the block-publish-list, of server blocks that need to be stored on the Tangler servers. All server blocks belonging to collections that the client wants to publish are maintained on this list. Although, some collections may share server blocks, only a single entry is kept for each server block. Associated with each server block on the list is a reference count that records the number of collections that entangle with the particular server block. When the client no longer wishes to maintain a specific collection, the reference counts associated with the collection's entangled server blocks are decremented. If the reference count of a server block reaches zero it is removed from the list. Every time a new set of collection server blocks are added to the list, the publisher thread saves the list to disk. This allows the client to recover the list following a crash. Figure 5.10 shows the Tangler Collection Publish Screen.

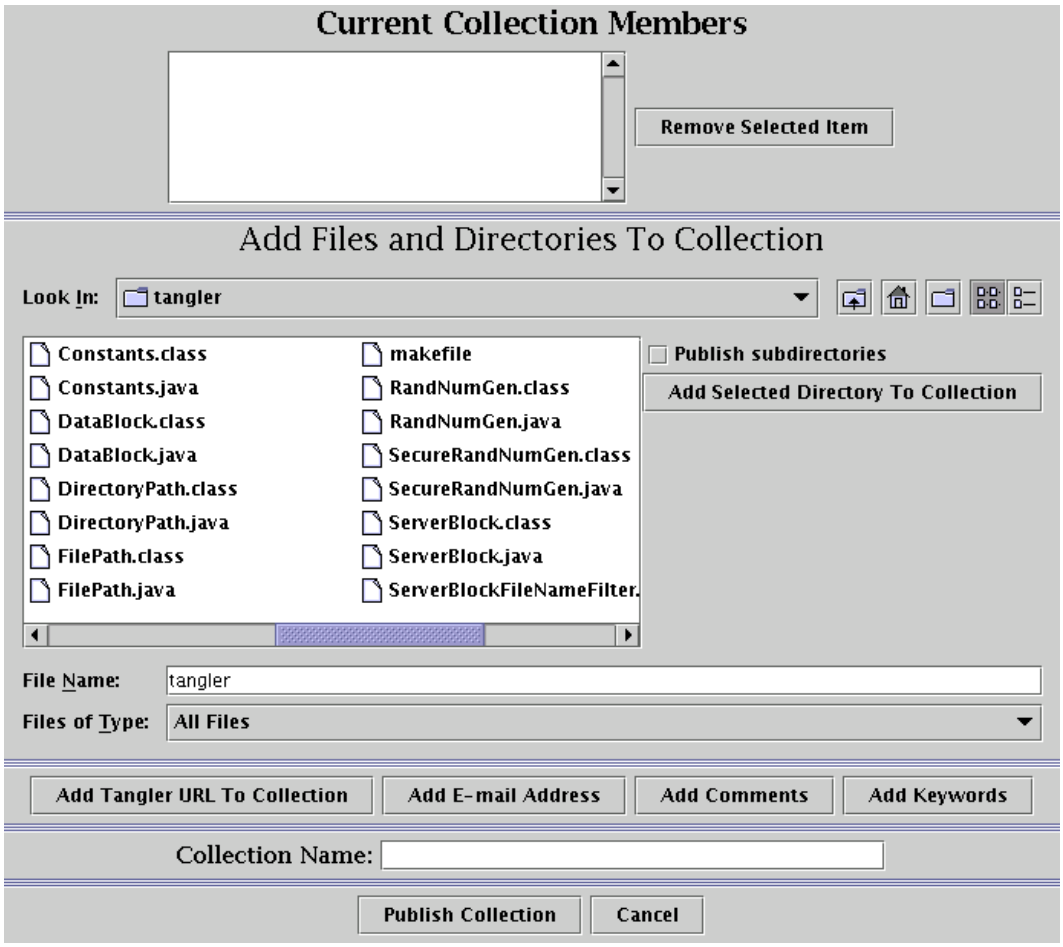

Figure 5.10: **Tangler Collection Publish Screen** File, Directories and Tangler URLs can be added to the collection.

Token Manager Thread

The token manager thread is responsible for acquiring the storage tokens needed to store the server blocks listed in the block-publish-list. Acquisition of these tokens is done differently depending on the type of client that wishes to store the blocks.

As described in Section 5.7.8, there are two different types of publishing clients. The first type of client, called the server-based client, is run by an individual that

is hosting a Tangler server. The second type of client, called the external client, is not associated with a Tangler server but still wishes to store or retrieve collections from the Tangler network. Both types of clients need to acquire storage coupons that are presented to servers in exchange for storage tokens. Mechanisms by which external clients can obtain storage coupons are described in Section 5.7.8. Below we describe how a server-based client acquires storage tokens.

The server-based client can easily generate storage coupons for itself as it has access to the server's private key. The client can then send these coupons to the appropriate Tangler server. The client calculates the number of server blocks it is entitled to—the calculation is based on the associated server's disk space donation (as described in Section 5.7.7). Let *entitlementNumber(s)* be the number of storage tokens that the server-based client is entitled to on server s. The token manager thread creates a list of size *entitlementNumber(s)* for each server s in the Tangler network. The thread then traverses the block-publish-list and determines which server to publish each server block on. A server block to be stored on server s is placed in one slot of the list corresponding to that server. A server block is only added to the list if the list has an open slot. The consistent hashing based algorithm described in Section 5.7.6 dictates which list the server block will be placed in. If a server block cannot be placed on a particular list, the block is placed on the list of a successor server. If the thread reaches the bottom of the block-publish-list, and there are still open slots in the server lists, the process is repeated—for each server block in the block-publish-list, the thread finds the most appropriate server list that does not already contain the server block. Eventually, either all the server-block-list slots are filled or all the server blocks in the block-

```
message CouponMessage {
        Long                    roundNumber;
        String                  issuingServerID;
        String                  redeemingServerID;
        LinkedList              blindedShaHashes;
        Signature               issuingServerSignature;
}
```

Figure 5.11: Storage Coupon Message Format

publish list can no longer be placed on the lists without adding redundant entries. In either case, the thread keeps a pointer into the server-block-list. This pointer points to the last server block added to an empty slot in the server lists. During the next round, the storage token creation process begins from that pointer. This gives every server block the opportunity to eventually be stored.

Once the token manager thread has determined where the blocks should be stored, the storage coupons are created. The format of a storage coupon is shown in Figure 5.11. The *issuingServerID* field identifies the Tangler server that has signed the message and also designates which server should be "billed" for these tokens. Recall that each server must grant storage tokens to other servers. The number of storage tokens is proportional to the amount of storage a server has donated for use by the Tangler network. Every server keeps track of the number of storage tokens it has granted during the round. When a server receives a storage coupon it decrements the number of tokens that the issuing server is entitled to for that round—this is why we used the term "billed." Notice that the last field requires the client to sign the message using the private key of the issuing server. This is not a problem because the server-based client has direct access to the server's private key.

Once the storage coupon is constructed, the coupon is sent to the appropriate server. If the server does not answer, the coupon is added to a retry queue. When all coupons have been sent, all entries on the retry queue are sent again. If the server refuses to answer for the second time the client contacts each Tangler server, in turn, in order to enlist them as witnesses. If any of the witnesses are successful in procuring the signed tokens, the client stops enlisting witnesses for that request. All requests for tokens and witnesses are made over the anonymous communication channel. Table 5.5 shows the amount of time that is needed for various token operations including blinding and verifying tokens.

Once the client receives the signed storage tokens back from the server, or via a witness, it can start storing the blocks. The block and the signed token are sent to the appropriate server over the annonymous communication channel. If a server refuses to store a block, the client enlists witness servers that attempt the storage operation on the client's behalf. The client contacts witnesses using the anonymous communication channel.

The client must be careful in the way it redeems the storage tokens. A server that receives a request for tokens and then a short period later, receives a block storage request, will likely be able to correlate the token request with the storage request. This can degrade client anonymity. This is especially a problem for the server-based client as the client and server are essentially operated by the same individual or organization and storage coupons are signed by corresponding server. To prevent this form of correlation, a round is divided into a token acquisition phase and a token spend phase. During the token acquisition phase the client requests tokens from the servers and enlists witnesses against servers that do not provide

the tokens. During the token spend phase clients send the blocks and tokens to the appropriate servers and enlist witnesses when a server refuses to store a block. All clients choose a random point within the phases to begin—the clients do not operate in lock-step requesting tokens and storing blocks at the same time. The token acquisition phase starts at the beginning of a new round and ends at the midpoint of the round. The storage phase starts at the round midpoint and lasts until the end of the round.

Unfortunately, there is another attack on client anonymity that is not thwarted by the addition of the token acquisition and token spend phases. An adversarial server can attempt to learn which server blocks correspond to a particular storage coupon. In this attack, the adversarial server only grants storage tokens to a single client. Once the token spend phase begins the client will send the tokens and corresponding blocks to the server. Since the server only granted tokens to a single client the correspondence between the request and blocks is obvious. Once again, this attack is mainly a problem for server-based clients—the server signed the client's storage coupon. Notice that by refusing to grant storage tokens the adversarial server has put its Tangler membership in jeapardy. Clients that didn't receive storage tokens in response to their requests contacted witness servers. These witness servers recorded the fact that the adversarial server refused to give tokens. This will lead the servers to eject the adversarial server. However, this doesn't protect the client. Therefore, prior to beginning the storage phase, each client asks every Tangler server for the list of servers that have refused to issues tokens during the round. If a majority of servers claim a particular server did not issue tokens then the client will not attempt to store blocks on that server. This prevents

Token Operation	Avg. Time Per 1000 Tokens
Blind Token	2,621
Sign Blinded Token	66,476
Verifying Token Signature	1,133
Verifying Token Signature and Unblind	1,289

Table 5.5: Token Operation Performance In Milliseconds

the adversarial server from matching token and storage requests.

5.8.8 Tangler Server Implementation

Upon startup, each Tangler server attempts to locate a file, called *round-info*, that specifies the members of the Tangler network. In order to bootstrap the Tangler network, we assume this file is initially created by the individuals that are organizing the Tangler network. This file contains the domain name, port numbers, public key, and disk space contribution size of every server participating in the Tangler network. If a server does not find itself listed in the round-info file, the server ceases operation and records the failure in a local log file. The round-info file is round specific—a new file is created for each round. The following subsections describes the important threads that are running within the server.

Server Thread

The Tangler server consists of two main threads that run concurrently. The first, called the Server Thread, is responsible for interacting with other Tangler servers. The second thread, called the Client Thread, is responsible for interaction with clients. The operation of the Client Thread is described in the next subsection.

All communication between Tangler servers is conducted over long-lived authenticated bi-directional communication channels. The first task of the server thread is to setup these channels. The thread sorts the round-info entries by public key and attempts to open a socket connection to all servers that follow its entry in the sorted list. This is done to allow each server to have a single bi-directional authenicated connection to every other server.

The Server Thread is responsible for keeping track of the current round number and the server members of the round. At the end of a round the Server Thread signals the Client Thread (see Section 5.8.8 to commit all newly acquired server blocks by rebuilding the server block and collection public key hash trees. Once these trees have been built the Server Thread forms a new commitment certificate. The commitment certificate contains the roots of the two hash trees (server block and collection public key) and also contains the new temporary round-key that will be used by the server to sign tokens. This commitment certificate is sent to all other servers.

In order to prevent a server from sending two different commitment certificates during a particular round, each server echoes every commitment certificate that it receives. The echo mechanism is utlized once per unique commitment certificate—every time a new commitment certificate is seen by the server it is sent to all other servers. As every commitment certificate is signed, two commitment certificates, for the same round, bearing different values is succinct proof of a faulty server. This proof is sent to all other servers so that the faulty server can be scheduled for ejection.

The Server Thread keeps track of which servers it has deemed faulty. At the

appropriate time the Byzantine Agreement based voting algorithm is run. The server sends its vote and tally's the votes of others. A server that is voted-out writes an appropriate log message to the system logger and exits.

Client thread

The first task of the Client Thread is to build the server block and collection root hash trees. Recall that one of the fields of the round-info file records the amount of disk space that each server is willing to donate to the Tangler network. This disk space donation translates into a fixed number of server blocks. A Tangler server must ensure that it is always hosting at least this many server blocks. The Client Thread may need to generate random server blocks in order to meet this requirement. The reason for the requirement is that Tangler clients request blocks from the Tangler servers. These blocks may ultimately be used during the entanglement process. The generation of random server blocks, is done prior to building the server block hash tree. Once all the necessary server blocks have been generated the Client Thread sorts the server block by their SHA-1 hash values and builds the server block and collection root hash trees. The sorting is performed to enable the client thread to generate certifying paths as described in subsection 5.8.7.

The Client Thread, as its name suggests, handles all client requests. It accepts requests for server blocks, creates certifying paths, accepts new server blocks, creates storage tokens, accepts server join requests and acts as a witness at the request of a client.

5.9 Discussion and Future Work

Once a server is ejected there is a risk of revealing which blocks that server's tokens have supported—those blocks will slowly disappear unless the user is able to publish them through another server. Other possible attacks include reducing performance by acting correctly but deliberately slowly.

Possible enhancements to Tangler include mechanisms 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. In addition, Tangler needs better mechanisms to resist being taken-over by a majority of adversarial servers. The easiest mechanism to employ is to limit the number of Tangler servers that can join from an particular subnet. However, this mechanism is relatively easy to defeat [23]. Another way of dealing with this problem is to simply form a new Tangler network when it appears the network has been taken-over by some adversary. Therefore many Tangler networks could exists simultaneously. Clients simply use the network that they trust.

Tangler's Byzantine Agreement algorithm is synchronous and therefore requires that time be divided into a series of phases. Each phase takes a specific amount of time. Asynchronous agreement algorithms such as those described in [14, 13] would, in most circumstances, allow the Tangler servers to come to agreement much quicker. However, these algorithms are quite complex and difficult to implement.

The Tangler protocols we have proposed provide anonymity for publishing while preventing storage-based flooding attacks. Though blocks are dispersed untrace-

ably across all servers, no server can consume more than its fair share of storage. Because different servers employ different client association policies, it is unlikely that an attacker could simultaneously monopolize all servers' available storage tokens. The Tangler protocol also implicitly audits servers' behavior at many stages, ensuring that faulty servers can be ejected. By disallowing servers from publishing during their probationary period, the protocol ensures that even bad servers do more good than harm.

Chapter 6

Conclusion

Publius and Tangler have made important contributions to the area of censorship-resistant publishing systems and are among the few systems to have actually been implemented.

An important aspect of both systems is that the servers storing the published documents do not know what type of documents are being stored. In Publius, each server stores encrypted documents. In Tangler, this is taken a step further—each server is storing blocks that have no meaning unless they are combined, in well-defined ways, with other blocks stored on the servers.

Publius's main contributions beyond previous anonymous publishing systems include a URL based tamper checking mechanism, a method for updating or deleting anonymously published material, and methods for anonymously publishing mutually hyperlinked content.

We believe Tangler is the first implemented censorship-resistant publishing system to attempt to identify and eject adversarial members of the system. The Free

Haven system (Section3.1.5) attempts to utilize a "reputation system" to identify and penalize adversarial servers. However, the Free Haven design has never been fully specified or prototyped and therefore it unclear whether this system is practical.

Tangler's entanglement mechanism is the first implemented fault-tolerant publishing mechanism that provides a publisher with some incentive to cache and reinject the blocks of documents belonging to others. The entanglement mechanism also disperses the responsibility for serving documents as it breaks the one-to-one correspondence between data blocks and documents—a particular data block can be used to reconstruct more than one document. While we believe entanglement could be fruitfully utilized by other censorship resistant publishing systems it is particularly well suited for use with Tangler's self-policing mechanism. Both mechanisms work to ensure that previously published documents remain accessible.

Tangler's other contributions include a design and implementation of a dynamic address space and a solution for storage based denial-of-service attacks. Tangler's dynamic address space attempts to prevent a minority of malicious servers from gaining permanent control over a particular document. Tangler's token storage mechanism allows a system to support anonymous publication and yet still prevent an adversary from filling all available storage.

Tangler is the first censorship-resistant publishing system in which participating servers are truly prevented from suppressing information. Tanger servers have no control over which blocks they store, and a server that drops even a small fraction of its blocks will be ejected with high probability.

Perhaps more interestingly, Tangler is the first system that completely disso-

ciates the contents of stored data from any published documents. This has not only technical but also interesting legal implications, as questions of responsibility and liability for published content are far from clear-cut. As the system gains use, some of these questions may be resolved, and future systems may need to adapt to the changing legal landscape. However, ultimately we believe that the Internet will remain a place in which, through one mechanism or another, people can widely distribute content despite any efforts to silence them.

Bibliography

[1] RFC 1521. `http://www.ietf.org/rfc/rfc1521.txt` [cited 2/5/2003].

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures And Algorithms*. Addison-Wesley Publishing Company, 1983.

[3] Yair Amir and Avishai Wool. Evaluating quorum systems over the internet. In *Proceedings of Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 26–35, 1996.

[4] Ross Anderson. The eternity service. In *Proceedings of Pragocrypt 1996*, 1996.

[5] Anonymizer. `http://www.anonymizer.com` [cited: 5/1/2002].

[6] Maurice J. Bach. *The Design Of The Unix Operating System*. Prentice Hall, 1990.

[7] Adam Back. Hash cash: A partial hash collision based postage scheme. `http://www.cypherspace.org/~adam/hashcash/` [cited: 5/1/2002].

[8] Adam Back. The eternity service. *Phrack Magazine*, 7(51), 1997. `http://www.cypherspace.org/~adam/eternity/phrack.html` [cited: 5/1/2002].

[9] Omar Bakr and Idit Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pages 243–252. ACM Press, 2002.

[10] Arash Baratloo, Mehmet Karaul, Zvi M. Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the Ninth International Conference on Parallel and Distributed Computing Systems*, 1996.

[11] Amos Beimel and Shlomi Dolev. Buses for anonymous message delivery. In *Second International Conference on FUN with Algorithms*, pages 1–13, Elba, Italy, May 2001. Carleton Scientific.

[12] Oliver Berthold, Hannes Federrath, and Stefan Kopsell. Web mixes: A system for anonymous and unobservable internet access. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 115–129. Springer-Verlag, 2000.

[13] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132. ACM Press, 2000.

[14] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999. USENIX Association.

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

[16] David Chaum. Blind signature system. In *Advances in Cryptology: Proceedings of Crypto '83*, page 153, 1983.

[17] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of ACM Digital Libraries*. ACM, August 1999.

[18] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[19] 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.

[20] Thomas Demuth and Andreas Rieke. On securing the anonymity of content providers in the world wide web. In *Proceedings of SPIE '99*, volume 3657, pages 494–502, 1999.

[21] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[22] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal of Computing*, 12(4):656–666, 1983.

[23] John R. Douceur. The sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002.

[24] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology: Proceedings of Crypto '92*, pages 139–147. Springer-Verlag, 1992.

[25] Tera News Usenet News FAQ. `http://www.teranews.com/faq.html` [cited 4/4/2003].

[26] Nick Feamster, Magdelena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing web censorship and surveillance. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, California, August 2002. ACM Press.

[27] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An Internet con game. Technical Report TR-540-96, Princeton University, Princeton, New Jersey, December 1996.

[28] Kevin Fu, M. Frans Kaashoek, and David 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.

[29] Andrew V. Goldberg and Peter 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.

[30] Ian Goldberg. A pseudonymous communications infrastructure for the internet. PhD thesis, Computer Science Department, University of California, Berkeley, 2000.

[31] Ian Goldberg and Adam Shostack. Freedom network 1.0 architecture. November 1999.

[32] Ian Goldberg and David Wagner. TAZ servers and the rewebber network: Enabling anonymous publishing on the world wide web. *First Monday*, 3, 1998. `http://www.firstmonday.dk/issues/issue3_4/goldberg/index.html` [cited: 5/1/2002].

[33] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, pages 79–90, Urbana-Champaign, Illinois, 1995.

[34] Maurice P. Herlihy and J.D. Tygar. How to make replicated data secure. In *Advances In Cryptology: Proceedings of Crypto '87*, pages 379–391. Springer Verlag, 1988. Lecture Notes in Computer Science No. 293.

[35] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina 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.

[36] Hugo Krawczyk. Secret sharing made short. In *Advances in Cryptology: Proceedings of Crypto '93*, pages 136–143. Springer-Verlag, 1993.

[37] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[38] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. On the composition of authenticated byzantine agreement. In *Proceedings of the Thirty-Fourth Annual ACM symposium on Theory of Computing*, pages 514–523. ACM Press, 2002.

[39] David Mazières and M. Frans Kaashoek. The design, implementation and operation of an email pseudonym server. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 27–36. ACM Press, 1998.

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

[41] Silvio Micali. CS proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[42] David L. Mills. Network time protocol (version 3) — specification, implementation and analysis. *RFC 1305*, March 1992.

[43] David L. Mills. Public key cryptography for the network time protocol. Technical Report 00-5-1, Electrical Engineering Dept., University of Delaware, Newark, Delaware, May 2000.

[44] Moni Naor. Verification of a human in the loop or identification via the turing test. Unpublished draft `http://www.wisdom.weizmann.ac.il/~naor/PAPERS/human_abs.html` [cited: 5/1/2002], 1996.

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

[46] U.S. Library of Congress. About the federalist papers. `http://lcweb2.loc.gov/const/fed/abt_fedpapers.html` [cited 5/1/2002].

[47] Network Time Protocol. `http://www.ntp.org` [cited: 5/1/2002].

[48] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[49] Michel Raynal. *Distributed Algorithms and Protocols.* John Wiley & Sons, Inc., 1988.

[50] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Proxies for anonymous routing. In *Proceedings of the 12th Annual Computer Security Applications Conference*, pages 95–104. IEEE Computer Society, December 1996.

[51] Michael K. Reiter. Distributing trust with the rampart toolkit. *Communications of the ACM*, 4(39):71–74, April 1996.

[52] Michael K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 1(22):31–42, January 1996.

[53] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information System Security*, 1(1), April 1998.

[54] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[55] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Cambridge, England, December 1993. Springer-Verlag.

[56] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.

[57] SETI@Home Software. `http://setiathome.berkeley.edu/` [cited: 5/1/2002].

[58] Noah Shachtman. Why countries make sites unseen. July 2002. `http://www.wired.com/news/politics/0,1283,53933,00.html` [cited: 2/5/2003].

[59] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

[60] Crypto++ Web Site. `http://www.eskimo.com/~weidai/cryptlib.html` [cited: 2/5/2003].

[61] Peek-A-Booty Web Site. `http://www.peek-a-booty.org/` [cited: 2/5/2003].

[62] Tim Skirvin. Usenet cancel faq v1.75. September 1999.

[63] Douglas Stinson. *Cryptography: Theory and Practice*. CRC Press, Inc, 1995.

[64] Ion Stoica, Robert Morris, David Karger, and M. Frans Kaashoek. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceeding of ACM SIGCOMM 2001*, pages 149–160, San Deigo, California, August 2001.

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

[66] Paul F. Syverson, Gene Tsudik, Michael G. Reed, and Carl E. Landwehr. Towards an analysis of onion routing security. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, 2000.

[67] Gerard Tel. *Introduction to Distributed Algorithms, Second Edition.* Cambridge University Press, 2000.

[68] Luis von Ahn, Manuel Blum, Nicholas Hopper, and John Langford. Captcha: Using hard AI problems for security. *Advances in Cryptology: Proceedings of Crypto '03*, 2003.

[69] Phil Zimmerman. PGP user's guide. December 4, 1992.