USENIX Association

Proceedings of the
9th USENIX Security Symposium

Denver, Colorado, USA
August 14–17, 2000

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Publius: A robust, tamper-evident, censorship-resistant web publishing system

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

Aviel D. Rubin
AT&T Labs–Research
rubin@research.att.com

Lorrie Faith Cranor
AT&T Labs–Research
lorrie@research.att.com

Abstract

We describe a system that we have designed and implemented for publishing content on the web. Our publishing scheme has the property that it is very difficult for any adversary to censor or modify the content. In addition, the identity of the publisher is protected once the content is posted. Our system differs from others in that we provide tools for updating or deleting the published content, and users can browse the content in the normal point and click manner using a standard web browser and a client-side proxy that we provide. All of our code is freely available.

1 Introduction

The publication of written words has long been a tool for spreading new (and sometimes controversial) ideas, often with the goal of bringing about social change. Thus the printing press, and more recently, the World Wide Web, are powerful revolutionary tools. But those who seek to suppress revolutions possess powerful tools of their own. These tools give them the ability to stop publication, destroy published materials, or prevent the distribution of publications. And even if they cannot successfully censor the publication, they may intimidate and physically or financially harm the author or publisher in order to send a message to other would-be-revolutionaries that they would be well advised to consider an alternative occupation. Even without a threat of personal harm, authors may wish to publish their works anonymously or pseudonymously because they believe they will be more readily accepted if not associated with a person of their gender, race, ethnic background, or other characteristics.

Quotations about the Internet's ability to resist censorship and promote anonymity have become nearly cliche. John Gillmore's quote "The Net treats censorship as damage and routes around it" has been interpreted as a statement that the Internet cannot be censored. And Peter Steiner's famous New Yorker cartoon captioned "On the Internet, nobody knows you're a dog" has been used to hype the Internet as a haven of anonymity. But increasingly people have come to learn that unless they take extraordinary precautions, their online writings can be censored and the true identity behind their online psuedonyms can be revealed.

Examples of the Internet's limited ability to resist censorship can be found in the Church of Scientology's attempts to stop the online publication of documents critical of the Church. Since 1994 the Church has combed the Internet for documents that contain what they describe as Church secrets. Individual authors, Internet service providers, and major newspapers such as *The Washington Post*, have had to defend their publication of excerpts from Church documents (some of them fewer than 50 words) in court. The Church has used copyright and trademark law, intimidation, and illegal searches and seizures in an attempt to suppress the publication of Church documents [13]. In 1995 the Church convinced the Finnish police to force Julf Helsingius, the operator of anonymous remailer anon.penet.fi, to reveal the true name of a user who had made anonymous postings about the Church. When the Church tried to obtain the names of two more users the following year, Helsingius decided to shut the remailer down [16].

The U.S. Digital Millennium Copyright Act, established to help copyright owners better protect their intellectual property in an online environment, is also proving to be yet another useful tool for censors. The Act requires online service providers to take down content upon notification from a copyright owner that the content infringes their copyright. While there is a process in place for the content owner to refute the infringement claim, the DMCA requires the online service provider to take down the content immediately and only restore it later if the infringement claim is not proven to be valid.

We developed Publius in an attempt to provide a Web publishing system that would be highly resistant to censorship and provide publishers with a high degree of anonymity. Publius was the pen name used by the authors of the Federalist Papers, Alexander Hamilton, John Jay, and James Madison. This collection of 85 articles, published pseudonymously in New York State newspapers from October 1787 through May 1788, was influential in convincing New York voters to ratify the proposed United States constitution [17].

1.1 Design Goals

Nine design goals were important in shaping the design of Publius.

Censorship resistant Our system should make it extremely difficult for a third party to make changes to or force the deletion of published materials.

Tamper evident Our system should be able to detect unauthorized changes made to published materials.

Source anonymous There should be no way to tell who published the material once it is published on the web. (This requires an anonymous transport mechanism between publishers and web servers.)

Updateble Our system should allow publishers to make changes to their own materials or delete their own materials should they so choose.

Deniable Since our system relies on parties in addition to the publisher (as do most publishing systems, online and offline), those third parties should be able to deny knowledge of the content of what is published.

Fault tolerant Our system should still work even if some of the third parties involved are malicious or faulty.

Persistent Publishers should be able to publish materials indefinitely without setting an upfront expiration date.

Extensible Our system should be able to support the addition of new features as well as new participants.

Freely available All software required for our system should be freely available.

2 Related work

For the purposes of this paper, current Web anonymizing tools are placed into one of two categories. The first category consists of tools that attempt to provide connection based anonymity – that is the tool attempts to hide the identity of the individual requesting a particular Web page. The second category consists of tools that attempt to hide the location or author of a particular Web document. Although Publius falls into the latter category we briefly survey connection based anonymity tools as they can be used in conjunction with Publius to further protect an author's anonymity.

2.1 Connection Based Anonymity Tools

The Anonymizer (`http://www.anonymizer.com`) provides connection based anonymity by acting as a proxy for HTTP requests. An individual wishing to retrieve a Web page anonymously simply sends a request for that page to the Anonymizer. The Anonymizer then retrieves the page and sends it back to the individual that requested it.

LPWA [9], now known as Proxymate, is an anonymizing proxy that also offers a feature that can automatically generate unique pseudonymous user names (with corresponding passwords) and email addresses that users can send to Web sites. Every time a user returns to a particular Web site, the same pseudonyms are generated. The functionality of the anonymizing proxy is very similar to that of the Anonymizer.

Several anonymity tools have been developed around the concept of mix networks [5]. A mix network is a collection of routers, called mixes, that use a layered encryption technique to encode the path communications should take through the network. In addition, mix networks use other techniques such as buffering and message reordering to further obscure the correlation between messages entering and exiting the network.

Onion Routing [18] is a system for anonymous and private Internet connections based on mix networks. 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, one layer of encryption is removed according to the recipe contained in the onion. The request arrives at the recipient in plain text, with only the IP address of the last onion-router on the path. An HTTP proxy has been developed that allows an individual to use the Onion Router to make anonymous HTTP requests.

Crowds [19] is an 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 surfers running the Crowds software. Crowds users forward HTTP requests to a randomly-selected member of their Crowd. Neither the end server nor any of the crowd members can determine where the request originated. 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.

The Freedom anonymity system (`http://www.freedom.net`) provides an anonymous Internet connection that is similar to Onion Routing; however, it is implemented at the IP layer rather than the application level. Freedom supports several protocols including HTTP, SMTP, POP3, USENET and IRC. In addition Freedom allows the creation of pseudonyms that can be used when interacting with Web sites or other network users.

2.2 Author Based Anonymity Tools

Janus, currently known as Rewebber (`http://www.rewebber.de`), is a combination author and connection based anonymizing tool. With respect to connection based anonymity, Janus functions almost exactly like the Anonymizer; it retrieves Web pages on an individual's behalf. Publisher anonymity is provided by a URL rewriting service. An individual submits a URL U to Janus and receives a Janus URL in return. A Janus URL has the following form

$$\text{http://www.rewebber.com/surf-encrypted/}E_k(U)$$

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.

Goldberg and Wagner [12] describe their implementation of an anonymous Web publishing system based on a network of Rewebbers. The Rewebber network consists of a collection of networked computers, each of which runs an HTTP proxy server and possesses a public/private key pair. Each HTTP proxy server is addressable via a unique URL. An individual wishing 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, 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, then file *f* cannot be found. Only the crashed file 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 that cannot be easily memorized. In order to associate a meaningful name with these long URLs the TAZ server was invented. 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. This anonymous publishing system is not currently operating as it was built as a "proof of concept" for a class project.

Most of the previous work in anonymous Web publishing has been done in the context of building a system to realize Anderson's Eternity Service [2]. The Eternity Service is a server based storage medium that is resistant to denial of service attacks and destruction of most participating file servers. An individual wishing to anonymously 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 servers participating in the Eternity Service. 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. Below we review several projects whose goals closely mirror or were inspired by the Eternity Service.

Usenet Eternity [3] is a Usenet news based implementation of a scaled down version of Anderson's Eternity Service. The system uses Usenet to store anonymously published documents. Documents to be published anonymously must be formatted according to a specific set of rules that call for the addition of headers and processing by PGP and SHA1. The correctly formatted message is then sent to alt.anonymous.messages. A piece of software called the eternity server is used to read the anonymously posted articles from 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 problem with using Usenet news to store the anonymously published file is that an article usually exists on a news server for only a short period of time before it is deleted. In addition a posting can be censored by a particular news administrator or by someone posting *cancel* or *supercede* requests (`http://www.faqs.org/faqs/usenet/cancel-faq/`) to Usenet. A much more ambitious implementation is currently being designed (`http://www.cypherspace.org/eternity-design.html`).

FreeNet [7] is an *adaptive network* approach to the censorship problem. FreeNet is composed of a network of computers (nodes) each of which is capable of storing files locally. In addition, each node in the network maintains a database that characterizes the files stored on some of the other nodes in the network. When a node receives a request for a non-local file it uses the information found in its database to decide which node to forward the request to. This forwarding is continued 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 nodes. Each node in this chain can cache the file locally. It is this caching that plays the main role in dealing with the censorship issue. The multiple copies make it difficult for someone to censor the material. A file can be published anonymously by simply uploading it to one of the nodes in the adaptive network. The FreeNet implementation is still in its infancy and many features still need to be implemented.

Intermemory [11] is a system for achieving an immense self-replicating distributed persistent RAM using a set of networked computers. An individual wishing to join the Intermemory donates 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. Data stored on the Intermemory is automatically replicated and dispersed. It is this replication and dispersion that gives the Intermemory properties similar to Anderson's Eternity Service. The main focus of the Intermemory project is not anonymous publishing but rather the preservation of electronic media. A small Intermemory prototype is described in [6]. The security and cryptographic components were not fully specified in either paper so we cannot comment on its anonymity properties.

Benes [4] describes in detail how one might implement a full-fledged Eternity service. Benes and several students at Charles University are attempting to create a software implementation of the Eternity Service based on this thesis.

3 Publius

In this section we describe how our system achieves the stated goals. We call the content that is published with the desired robustness properties *Publius content.*

3.1 Overview

Our system consists of *publishers* who post Publius content to the web, *servers* who host random-looking content, and *retrievers* who browse Publius content on the web. At present the system supports any static content such as HTML pages, images, and other files such as postscript, pdf, etc. Javascript also works. However, there is no support for interactive scripting such as CGI. Also, Java applets on Publius pages are limited in what they can do.

We assume that there is a static, system-wide list of available servers. Publius content is encrypted by the publisher and spread over some of the web servers. In our current system, the set of servers is static. The publisher takes the key, K that is used to encrypt the file to be published and splits it into n shares, such that any k of them can reproduce the original K, but $k-1$ give no hints as to the key [22].

Each server receives the encrypted Publius content and one of the shares. At this point, the server has no idea what it is hosting – it simply stores some random looking data.

To browse content, a retriever must get the encrypted Publius content from some server and k of the shares. As described below, a mechanism is in place to detect if the content has been tampered with. The publishing process produces a special URL that is used to recover the data and the shares. The published content is cryptographically tied to the URL.

Any modification to the stored Publius content or the URL results in a failed tamper check. If all tamper checks fail the Publius content cannot be read.

In addition to the publishing mechanism, we provide a way for publishers (and nobody else) to update or delete their Publius content. In the next several sections, we describe the Publius functions in some detail. We use a simple example of a publisher with one HTML file. Publishing more complicated content, such as web pages that have links to each other, is covered in Section 4.

3.2 Publish

The following text describes the publish pseudocode of Figure 1. This pseudocode is executed by the Publius client proxy in response to a publish request. To publish Publius content, M, the publisher, Alice, first generates a random symmetric key, K. She then encrypts M to produce $\{M\}_K$, M encrypted under K, using a strong symmetric cipher. Next, Alice splits K into n shares using Shamir secret sharing, such that any k of them can reproduce the secret.

For each of the n shares, Alice computes

$$name_i = wrap(H(M \cdot share_i))$$

That is, each share has a corresponding name. The name is calculated by concatenating the share with the message, taking a cryptographic hash, H, of the two, and xoring the first half of the hash output with the second half. We call the xor of the two halves *wrap*. In our system, we use MD5 [20] as the hash function, so each $name_i$ is 8 bytes long. Note that the $name_i$'s are dependent on every bit of the web page contents and the share contents. The $name_i$ values are used in the Publius server addressing scheme described below.

Recall that each publisher possesses a static list of size m of the available servers in the system. For each of the n shares, we compute

$$location_i = (name_i \text{ MOD } m) + 1$$

to obtain n values each between 1 and m. If at least d unique values are not obtained, we start over and pick another K. The value d represents the minimum number of unique servers that will hold the Publius content. Clearly this value needs to be greater than or equal to k since at least k shares are needed to reconstruct the key K. d should be somewhat smaller than m. It is clearly desirable to reduce the number of times we need to generate a new key K. Therefore we need to create a sufficient number of shares so that, with high probability, d unique servers are found. This problem is equivalent to the well known Coupon Collectors Problem [15]. In the Coupon Collectors Problem there are y different coupons that a collector wishes to collect. The collector obtains coupons one at a time, randomly with repetitions. The expected number of coupons the collector needs to collect before obtaining all y different coupons is $y * \ln(y)$. By analogy, a unique slot in the available server list is equivalent to a coupon. Therefore for each key K we create $n = \lceil d * \ln(d) \rceil$ shares. Any unused shares are thrown away.

Now, Alice uses each $location_i$ as an index into the list of servers. Alice publishes $\{M\}_k$, $share_i$, and some other information in a directory called $name_i$ on the server at location $location_i$ in the static list. Thus, given M, K, and m, the locations of all of the shares are uniquely determined. The URL that is produced contains at least d $name_i$ values concatenated together. A detailed description of the URL structure is given in Section 4.

Figure 2 illustrates the publication process.

3.3 Retrieve

The following text describes the retrieve pseudocode of Figure 3. This pseudocode is executed by the Publius client proxy in response to a retrieve request. The retriever, Bob, wishes to view the Publius content addressed by Publius URL U. Bob parses out the $name_i$ values from U and for each one computes

$$location_i = (name_i \text{ MOD } m) + 1$$

Thus, he discovers the index into the table of servers for each of the shares. Next, Bob chooses k of these arbitrarily. From this list of k servers, he chooses one and issues an HTTP GET command to retrieve the encrypted file and the share. Bob knows that the encrypted file, $\{M\}_K$ is stored in a file called *file* on each server, in the $name_i$ directory. The key share is stored in a file called *share* in that same directory.

Next, Bob retrieves the other $k - 1$ shares in a similar fashion (If all goes well, he does not need to retrieve any other files or shares). Once Bob has all of the shares, he combines them to form the key, K. Then, he decrypts the file. Next, Bob verifies that all of the $name_i$ values corresponding to the selected shares are correct by recomputing

$$name_i = wrap(H(M \cdot share_i))$$

using M that was just decrypted. If the k $name_i$'s are all correct (i.e. if they match the ones in the URL),

```
Procedure $Publish$ (document $M$)
    Generate symmetric key $K$
    Encrypt $M$ under key $K$ producing $\{M\}_K$
    Split key $K$ into $n$ shares such that $k$ shares are required to reconstruct K
    Store the $n$ shares in array $share[1..n]$
    $locations\_used = \{\}$
    for i = 1 to $n$:
        $name$=MD5($M \cdot share[i]$)
        $name$=XOR(top_64_bits($name$),bottom_64_bits($name$))
        $location$=($name$ MOD $serverListSize$)+1
        if ($location$ is not a member of $locations\_used$):
            $locations\_used = locations\_used \cup \{location\}$
            $serverIP\_Address = serverList[location]$
            Insert ($serverIP\_Address$, $share[i]$) into Publish_Queue
            $publiusURL = publiusURL \cdot name$
        endif
    endfor
    if (sizeof($locations\_used$)$< d$) then
        Empty (Publish_Queue)
        return $Publish(M)$
    else
        for each ($serverIP\_Address$, $share$) in Publish_Queue:
            HTTP_PUT($\{M\}_K$ and $share$ on Publius Server with IP address $serverIP\_Address$)
            return $publiusURL$
    endif
End $Publish$
```

Figure 1: Publish Algorithm

Bob can be satisfied that either the document is intact, or that someone has found a collision in the hash function.

If something goes wrong, Bob can try a different set of k shares and an encrypted file stored on one of the other n servers. In the worst case, Bob may have to try all of the possible $\binom{n}{k}$ combinations to get the web page before giving up. An alternate retrieval strategy would be to try all $n* \binom{n}{k}$ combinations of shares and documents. Each encrypted document can be tested against each of the $\binom{n}{k}$ share combinations.

If we are willing to initially download all the shares from all the servers then yet another method for determining the key becomes available. In [10], Gemmell and Sudan present the Berlekamp and Welch method for finding the polynomial, and hence the key K, corresponding to n shares of which at most j are corrupt. The value j must be less than $(n-d)/2$ where d is one less than the number of shares needed to form the key. However if the number of corrupt shares is greater than $(n-d)/2$ we are not quite out of luck. We can easily discover whether K is incorrect by performing the verification step described above. Once we suspect that key K is incorrect we can just perform a brute force search by trying all $n* \binom{n}{k}$ combinations of shares and documents. The following example illustrates this point. If we have $n = 10$ shares and require 3 shares to form K then the Berlekamp and Welch method will generate the correct polynomial only if less than $((10-2)/2) = 4$ shares are corrupted. Suppose 6 shares are corrupt. Of course we don't know this ahead of time so we perform the Berlekamp and Welch method which leads us to key K. Key K is tested against a subset of, or perhaps all, the encrypted documents. All of the tamper check failures lead us to suspect that K is incorrect. Therefore we perform a brute force search for the correct key by trying all $n* \binom{n}{k}$ combinations of shares and documents. Assuming we have a least one untampered encrypted document this method will clearly succeed as we have 4 uncorrupted shares, only three of which are needed to form the correct key.

Once the web page is retrieved Bob can view it in his browser. In our implementation, all of the work is handled by the proxy. Publius URLs are tagged as special, and they are parsed and handled in the proxy. The proxy retrieves the page, does all of the verification, and returns the web content to the browser. So, all of this is transparent to the user. The user just points and clicks as usual. Section 4 describes Publius URL's and the proxy software in detail.

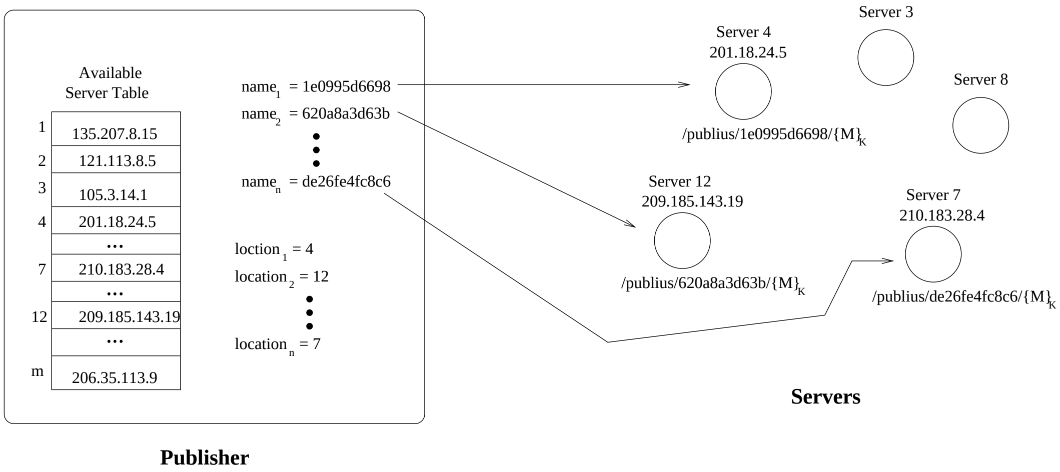

Figure 2: **The Publius publication process** The publisher computes the $name_i$ values by hashing the web page and the symmetric key shares together. Then, those values are used to compute the locations. The publisher then uses the location value as an index into the static location table and publishes the encrypted file, along with the share in a directory named $name_i$ on the appropriate server.

3.4 Delete

It is desirable for Alice to be able to delete her Publius content from all servers, while nobody else should be able to delete this content. To achieve this, just before Alice publishes a file she generates a password PW. Alice then sends the encrypted document, share and $H(server\_domain\_name \cdot PW)$ to the servers that will be hosting Alice's published document. $H(server\_domain\_name \cdot PW)$ is the hash of the domain name of the server concatenated with a 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 Publius content from all other servers that are hosting it.

We implemented *delete* as a CGI script running on each server. To delete Publius content, Alice sends $H(server\_domain\_name \cdot PW)$ to each hosting server, along with the $name_i$ that corresponds to the that server. The server compares the password received to the one stored, and if they match, removes the directory matching the $name_i$, and all of the files in it.

3.5 Update

Our system provides a mechanism for Alice to update something that she previously published. We use the same password mechanism that is used to delete content. Thus, Alice can change any web page that she published, but nobody else can. The idea is to enable Alice to change content without changing the URL, because others may have linked to the original site. After the update, anyone retrieving the original URL receives the new content.

In addition to *file*, *share*, and *password*, there is a file called *update* on the servers in the $name_i$ directory. Initially, if Alice has not updated the content, the file does not exist. When Bob retrieves the URL, if the update file is missing, everything proceeds as described in Section 3.3.

To update the content, Alice specifies a file name containing the new content, the original URL, the original password PW and a new password. The update program first publishes the new content by simply calling publish with the file name and the new password. Once the new content is published, the original URL is used to find the n servers that host the Publius content. Each of these servers receives a message from Alice (a call to a CGI script) containing the original password stored on that server (recall that this is $H(server \cdot PW)$), the old $name_i$, and the new URL. Each server then places the new URL in the *update* file and deletes the contents in the old *file*.

When Bob retrieves Publius content, if the update file exists, the servers return the update URL instead of the contents. Bob receives the update URL from k servers and compares them, if they are all equal, he then retrieves the new URL instead. Of course, Bob is not aware of what the retrieve program is doing behind the scenes. From his point of view, he makes

```
Procedure Retrieve (PubliusURL $U$)
    // $k$ is the number of shares needed to reconstruct Key $K$
    // $n$ is the number of name[i] values stored in the PubliusURL $U$
    // URL $U = name[1] \dots name[n]$
    $S$={set of all unique $k$-subsets of the elements $(1..n)$}
    for each element $s$ in $S$:
        $R$=randomValue($k$) // choose a random integer in range $1..k$
        for $i = 1$ to $k$:
            $v = i^{th}$ element of set $s$
            $location$=($name[v]$ MOD $serverListSize$)+1
            $serverIP\_Address$=$serverList[location]$
            $share[i]$=retrieve file "share" from server at $serverIP\_Address$
            $tamperCheckValue[i]$=$name$[v]
            if ($i$==$R$) then
                $encryptedDocument$=retrieve $\{M\}_k$ from server at $serverIP\_Address$
            endif
        endfor
        $K$=reconstructKeyFromShares($share[1] \dots share[k]$)
        $M$=Decrypt $encryptedDocument$ using key $K$
        $tamperCheckPassed$=TRUE
        for $i = 1$ to $k$:
            $V$=MD5($M \cdot share[i]$)
            $V$=XOR(top_64_bits($V$),bottom_64_bits($V$))
            if ($V \neq tamperCheckValue[i]$) then
                $tamperCheckPassed$=FALSE
                break
            endif
        endfor
        if ($tamperCheckPassed$) then
            return $M$
        endif
    endfor
    return "Document cannot be retrieved"
End Retrieve
```

Figure 3: Retrieve Algorithm

a request and receives the web page. If the k URLs do not match, Bob (his proxy) then tries the other $n-k$ servers until he either gets k that are the same, or gives up. In Section 5 we discuss other ways this could be implemented and several tradeoffs that arise.

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 publicaton 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 4.1 for more information about nonupdateable URLs.

4 Implementation issues

In this section we describe the software components of Publius and how these components implement Publius functions.

4.1 Publius URLs

Each successfully published document is assigned a Publius URL. A Publius URL has the following form

$$http://!anon!/options\ encode(name_1) \dots encode(name_n)$$

where $name_i$ is defined as in Section 3.2 and the encode function is the Base64 encoding function (verb+http://www.ietf.org/rfc/rfc1521.txt+). The Base64 encoding function generates an ASCII representation of the $name_i$ value.

The *options* section of the Publius URL is made up of 2 characters that define how the Publius client software interprets the URL. This 16 bit *options* section encodes three fields – the version number, the number of shares needed to form a key, and finally the update flag. The version number allows us to add 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 key K. The last field is the *update* flag that determines whether or not the update operation can be performed on the Publius content represented by the URL. If the *update* flag is a 1 then the retrieval of updated content will be performed in the manner described in Section 3.5. However if the *update* flag is 0 then the client will ignore update URLs sent by Publius servers in response to share and encrypted file requests. The update flag's role in preventing certain types of attacks is described in Section 5.

Many older browsers enforce the rule that a URL can contain a maximum of 256 characters. The initial "http://!anon!/" string is 14 characters long, leaving 242 characters for the 20 $name_i$ values. Base64 processes data in 24 bit blocks, producing 4 ASCII characters per 24 bit block. This results in 12 ASCII characters per $name_i$ value. Twenty hashes produce 240 ASCII characters. Thus, older browsers restrict us to 20 different publishing servers in our scheme. We use the two remaining characters for the *options* section described above.

Here is an example of a Publius URL:

http://!anon!/AH2LyMOBWJrDw=
GTEaS2GlNNE=NIBsZlvUQP4=sVfdKF7o/kl=
EfUTWGQU7LX=OCk7tkhWTUe=GzWiJyio75b=
QUiNhQWyUW2=fZAX/MJnq67=y4enf3cLK/0=

4.2 Server software

To participate as a Publius server, one only needs to install a CGI script that we provide. All 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. We recommend limiting the amount of disk space that can be used each time the CGI script executes. Our CGI script is freely available (see Section 7).

4.3 Client software

The client software consists of an HTTP proxy and a set of publishing tools. An individual wishing only to retrieve Publius content just requires the proxy. The proxy transparently sends non Publius URLs to the appropriate servers and passes the returned content back to the browser. Upon receiving a request for a Publius URL the proxy first retrieves the encrypted document and shares as described in Section 3.3 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.

4.4 Publishing mutually hyperlinked documents

Suppose Alice wants to anonymously publish HTML files A and B. Assume that file A contains a hyperlink to file B. Alice would like the anonymously published file A to retain its hyperlink to the anonymously published file B. To accomplish this, Alice first publishes file B. This action generates a Publius URL for file B, B_{url}. Alice records B_{url} in the appropriate location in file A. Now Alice publishes file A. Her task is complete.

Alice now wishes to anonymously publish HTML files C and D. File C has a hyperlink to file D and file D has a hyperlink to file C. Alice now faces the dilemma of having to decide which file to publish first. If Alice publishes file C first then she can change D's hyperlink to C but she cannot change C's hyperlink to D because C has already been published. A similar problem occurs if Alice first publishes file D.

The problem for Alice is that the content of a file is cryptographically tied to its Publius URL – changing the file 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 file'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 3.5.

Using the update mechanism Alice can easily solve the problem of mutually hyperlinked files. First Alice publishes files C and D in any order. This generates Publius URL C_{url} for file C and Publius URL D_{url}

for file D. Alice now edits file C and changes the address of the D hyperlink to D_{url}. She does the same for file D – she changes the address of the C hyperlink to C_{url}. Now she performs the Publius Update operation on C_{url} and the newly modified file C. The same is done for D_{url} and the newly updated file D. This generates Publius URL C_{url_2} for for file C and Publius URL D_{url_2} for file D. The problem is solved. Suppose Bob attempts to retrieve file C with C_{url}. Bob's proxy notices the file has been updated and retrieves the file from C_{url_2}. Some time later, Bob clicks on the D hyperlink. Bob's proxy requests the document at D_{url} and is redirected to D_{url_2}. The update mechanism ensures that Bob reads the latest version of each document.

4.5 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, as described in the previous section.

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 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 Publius Update trick that Alice uses when publishing her mutually hyperlinked files C and D (Section 4.4).

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. 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 [1]. 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 a Publius Update operation is performed on all files that were part of a cycle in *G*.

4.6 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. Indeed, this is the desired behavior as we do not wish to give the hosting server the slightest hint as to the type of content being hosted. 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. We prepend the three letter file extension rather than the actual MIME type because MIME types are of variable length (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 3.2. 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 User interface

The client side software includes command line tools to perform the publish, delete and update operations described in section 3. The retrieve operation is performed via the Web browser in conjunction with the proxy. In addition, a Web browser based interface to the tools has been developed. This browser based interface allows someone to select the Publius operation (*retrieve*, *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 !anon! URL that is recognized by the proxy. For example the publish URL is !anon!PUBLISH. The operation's parameters are sent in the body of the HTTP POST request to the corresponding !anon! URL. The proxy parses the parameters and executes the corresponding Publius operation. An HTML message indicating the success or failure of the operation is returned. If the retrieve operation is requested, and is successful, the requested document is displayed in a new Web browser window.

5 Limitations and threats

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

5.1 Share deletion or corruption

As described in section 3.2, when a document is successfully published a copy of the encrypted document and a share are stored on each of the 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.

5.2 Update file deletion or corruption

As stated in section 3.5, if a server receives a request for Publius content 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, Mallory, has read/write access to all files on a server hosting the Publius content P, he wishes to censor.

In the first attack we describe, P does not have an associated update file. That is, the author of P has not executed the Publius Update operation on *P's* URL. Mallory could delete P from one server, but this does not censor the content because there are other servers available. Rather than censor the Publius content, Mallory 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}. Mallory 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 Mallory's server, Q_{url} is sent back. Of course Mallory realizes that a single Q_{url} received by the client does not fool it into retrieving Q_{url}. Therefore Mallory enlists the help of several other Publius servers that store P. Mallory's friends also place Q_{url} into an "update" file in P's directory. Mallory's censorship clearly 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 Mallory does not necessarily need to be that thorough. When the proxy makes a request for P, if Mallory is lucky, then k matching URLs are returned and the proxy issues a browser redirect to that URL. If this happens Mallory has censored P and has replaced it with Publius Content of his own creation. This motivates higher values for k. The *update* flag described in section 4.1 is an attempt to combat this attack. If the publisher turned the update flag off when the content was published then the Publius client interpreting the URL will refuse to accept the update URLs for the document. Although the content might now be considered to be censored, someone is not duped into believing that an updated file is the Publius content originally published.

In the second attack, P has been updated and there exists an associated update file containing a valid Publius URL that points to Publius Content U. To censor the content, Mallory must corrupt the update file on $n - k + 1$ servers. Now there is no way for anyone to retrieve the file correctly. In fact, if Mallory can corrupt that many servers, he can censor any document. This motivates higher values for n and lower values for k.

One other attack is worth mentioning. If Mallory can cause the update files on all of the servers accessed by the client to be deleted, then he can, in effect, restore Publius content to its previous state before the update occurred. This motivates requiring clients to retrieve from all n servers before performing verification.

The attacks described above shed light on a couple of tradeoffs. Requiring retrievers to download all n shares and n copies of the document is one extreme that favors censorship resistance over performance. Settling for only the first k shares opens the user up to a set of corrupt, collaborating servers. Picking higher values for k minimizes this problem. However, lower values of k require the adversary to corrupt more servers to censor documents. Thus, k, the number of shares, and the number of copies of the page actually retrieved, must be chosen with some consideration.

5.3 Denial of service attacks

Publius, like 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 100K. A better approach would be to charge for space.

An interesting approach to this problem is a CPU cycle based payment scheme known as Hash Cash (`http://www.cypherspace.org/~adam/hashcash/`). The idea behind this system is to require the publisher to do some work before publishing. Thus, it becomes difficult to efficiently fill the server disk. Hopefully, the attack can be detected before the disk is full. In Hash Cash, a client 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 · s)* match b bits of *H(s)* where H is a secure hash function such as MD5 and "·" 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 · s)*. Another scheme we are 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. We have not implemented either of these protection mechanisms yet.

Dwork and Naor in [8] describe several other CPU cycle based payment schemes.

5.4 Threats to publisher anonymity

Although Publius was designed as a tool for anonymous 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 all 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 hosting Publius Content 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 Crowds should be used in conjunction with Publius.

5.5 "Rubber-Hose cryptanalysis"

Unlike Anderson's Eternity Service [2] 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 [21] (threats, torture, blackmail, etc) to either force the publisher to delete the document or reveal the document's password.

Of course the adversary could try to force the appropriate server administrators to delete the Publius Content he wants censored. However when Publius Content is distributed across servers located in different countries and/or jurisdictions such an attack can be very expensive or impractical.

6 Future Work

Most of the browsers and proxies in use today do not impose the 256 character limit on URL size. With this limit lifted a fixed table of servers is no longer needed as the Publius URL itself can contain the IP addresses of the servers on which the content is stored. With no predefined URL size limit there is essentially no limit to the number of hosting servers that can be stored in the Publius URL. The Publius URL structure remains essentially the same – just the IP addresses are added. The *option* and $name_i$ components of the URL remain as they are still needed for tamper checking and URL interpretation. We intend to use this URL format in future versions of Publius.

During the Publius publication process the encrypted file, along with other information, is stored on the host servers. Krawczyk in [14] describes how to use Rabin's information dispersal algorithm to reduce the size of the encrypted file stored on the host server. We are planning to use this technique to reduce amount of storage needed on host servers.

7 Conclusions and availability

In this paper we have described Publius, a Web based anonymous publishing system that is resistant to censorship. Publius's main contributions beyond previous anonymous publishing systems include an automatic tamper checking mechanism, a method for updating or deleting anonymously published material, and methods for anonymously publishing mutually hyperlinked content.

The current implementation of Publius consists of approximately fifteen hundred lines of Perl. The source code is freely available at `http://www.cs.nyu.edu/~waldman/publius.html`.

Acknowledgements

We would like to thank Usenix for supporting this work. We would also like to thank Adam Back, Ian Goldberg, Oscar Hernandez, Graydon Hoare, Benny Pinkus, Adam Shostack, Anton Stiglic, Alex Taler and the anonymous reviewers for their helpful comments and recommendations.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures And Algorithms.* Addison-Wesley Publishing Company, 1983.

[2] R. J. Anderson. The eternity service. In *Pragocrypt 1996*, 1996. http://www.cl.cam.ac.uk/users/rja14/ eternity/eternity.html.

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

[4] T. Benes. The eternity service. 1998. http://www.kolej.mff.cuni.cz/eternity/.

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

[6] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. N. Yianilos. A prototype implementation of archival intermemory. In *Proc. ACM Digital Libraries.* ACM, August 1999. http://www.intermemory.org/.

[7] I. Clark. A distributed decentralised information storage and retrieval system. 1999. http://freenet.sourceforge.net/Freenet.ps.

[8] C. Dwork and M.Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO '92*, pages 139–147. Springer-Verlag, 1992.

[9] E.G. Gabber, P.B. Gibbons, D.M. Kristol, Y. Matias, and A. Mayer. Consistent, yet anonymous web access with LPWA. *Communications of the ACM*, 42(2):42–47, 1999.

[10] P. Gemmell and M. Sudan. Highly resilient correctors for polynomials. *Information Processing Letters*, 43:169–174, 1992.

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

[12] I. Goldberg and D. 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.

[13] Wendy M. Grossman. *Wired*, 3(12):172 –177 and 248–252, December 1995. http://www.wired.com/wired/archive/3.12/ alt.scientology.war_pr.html.

[14] H. Krawczyk. Secret sharing made short. In *Advances in Cryptology—CRYPTO '93*, pages 136–143. Springer-Verlag, 1993.

[15] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[16] Ron Newman. The church of scientology vs. the net. February 1998. http://www2.thecia.net/users/rnewman/ scientology/home.html.

[17] U.S. Library of Congress. About the federalist papers. http://lcweb2.loc.gov/const/fed/abt_fedpapers.html.

[18] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Proxies for anonymous routing. In *12th Annual Computer Security Applications Conference*, 1996. http://www.onion-router.net/Publications.html.

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

[20] R. Rivest. The MD5 message digest algorithm. *RFC 1321*, April 1992.

[21] B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.

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