University of Crete
Computer Science Department

A Lightweight Censorship-Resistant Web Access Architecture

Georgios Kontaxis

Master's Thesis

June 2011
Heraklion, Greece

University of Crete
Computer Science Department

A Lightweight Censorship-Resistant Web Access Architecture

Thesis submitted by

Georgios Kontaxis

in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author:	Georgios Kontaxis
Committee approvals:	Evangelos P. Markatos Professor, Thesis Supervisor
	Sotiris Ioannidis Principal Researcher
	Maria Papadopouli Assistant Professor
Departmental approval:	Angelos Bilas Professor, Chairman of Graduate Studies

Heraklion, June 2011

Abstract

Most anti-censorship Web access mechanisms rely on a limited set of proxy nodes whose information is publicly available and, thus, communication with them may easily be blocked. In this paper, we present an architecture that leverages the Web to create an access mechanism that is highly-resilient to the blocking of its components. Specifically, we present how existing Web services can be leveraged as coordination and storage components to support a swarm of lightweight nodes that serves Web access requests and is run by volunteers inside their Web browser. A client of our system, who desires unobstructed access to a Web site, posts the respective URL on a Web service acting as a coordination point, such as a blog or a social network. Carefully crafted JavaScript code enables a continually changing body of volunteers, who reside outside the restricted zone, to fetch the content of the requested URL and store it on an uncensored user-content hosting service, from which the client ultimately accesses it. As our approach is not tied to a specific service but rather uses a plethora of existing Web services, we argue that a censorship authority will be unwilling or unable to monitor and block them all without greatly impairing user connectivity to the Internet. We explore the strengths, limitations, performance and security of our system.

Περίληψη

Οι περισσότεροι, ενάντια στη λογοκρισία, μηχανισμοί πρόσβασης στο Web βασίζονται σε ένα περιορισμένο σύνολο από κόμβους - διαμεσολαβητές των οποίων τα στοιχεία είναι δημόσια διαθέσιμα και έτσι είναι πολύ εύκολο κάποιος να εμποδίσει την επικοινωνία μαζί τους. Σε αυτή την εργασία, παρουσιάζουμε μια αρχιτεκτονική η οποία χρησιμοποιεί το Web για να δημιουργήσει έναν μηχανισμό πρόσβασης ο οποίος είναι ιδιαίτερα ανθεκτικός στην φραγή των μερών του. Ειδικότερα, παρουσιάζουμε πως υπάρχουσες υπηρεσίες του Web μπορούν να χρησιμοποιηθούν ως μέρη συντονισμού και αποθήκευσης πληροφορίας ώστε να υποστηρίξουν ένα σμήνος ελαφριών κόμβων το οποίο εξυπηρετεί αιτήματα πρόσβασης στο Web και λειτουργείται από εθελοντικές μέσα από τον Web browser τους. Χρήστες του συστήματός μας, οι οποίοι επιθυμούν ελεύθερη πρόσβαση σε ένα Web site, αναρτούν το αντίστοιχο URL σε μια Web υπηρεσία, όπως ένα blog ή σελίδα κοινωνικής δικτύωσης, η οποία λειτουργεί ως σημείο συντονισμού. Προσεκτικά κατασκευασμένος κώδικας JavaScript επιτρέπει σε ένα συνεχώς μεταβαλλόμενο σώμα εθελοντών, το οποίο βρίσκεται έξω από τη ζώνη λογοκρισίας, να φέρει το περιεχόμενο του URL και να το αποθηκεύσει σε μια υπηρεσία φιλοξενίας περιεχομένου χρηστών, η οποία δεν υπόκειται σε λογοκρισία, από την οποία τελικά θα έχει πρόσβαση στο περιεχόμενο ο χρήστης. Καθώς η προσέγγισή μας δεν σχετίζεται με μια συγκεκριμένη Web υπηρεσία αλλά μπορεί να λειτουργήσει με μια πληθώρα υπαρχόντων υπηρεσιών, υποστηρίζουμε ότι μια εξουσία λογοκρισίας δεν θα επιθυμεί ή δεν θα μπορεί να παρακολουθεί και να εμποδίζει την πρόσβαση σε όλες χωρίς να βλάψει σημαντικά την συνδεσιμότητα των χρηστών της με το Internet. Εξερευνούμε τα δυνατά σημεία, τους περιορισμούς, την απόδοση και την ασφάλεια του συστήματός μας.

Contents

List of Figures

1 Introduction

While unobstructed access to the World Wide Web was originally taken for granted, it has lately been hindered in a plethora of networks and incidents around the world. For example, China is known to block [5] sites belonging to activist groups and South Korea [16] is seeking to filter anti-government blogs. However, it is not only typically-considered "oppressive regimes" that censor the Web access of their Internet users. As a matter of fact, very few countries around the world do not have some, limited or more extended, form of censorship [10, 12].

Anti-censorship mechanisms operate outside such restricted zones and provide a way for Web content to be fetched on behalf of their users. A number of such mechanisms exists, with open HTTP proxies, Tor [24] and VPN solutions being among the most popular choices. Existing anti-censorship mechanisms offer guarantees for unobstructed access to the Web but do not offer any guarantee for their reachability. Their set of nodes is, by design, listed publicly for users to be able to discover them. Thus, a censorship authority can easily block access to them at the network level [19].

Our motivation for this paper is the need for a new design that is resilient to black-listing attempts. We consider the World Wide Web, which is the most active Internet platform. Millions of users use the Web daily to access thousands of services. At the same time there exists a multitude of sites hosting user-generated content, ranging from entries in forums and blogs (e.g., Blogspot) to social networks (e.g., Facebook, Twitter) and media sites (e.g., Youtube). The enormous user base and number of available services provide the opportunity of using the Web as a suitable platform for implementing a censorship-resistant access mechanism that does not suffer from the black-listing problem.

Inspired by the concept of forming distributed systems using Web browsers [26] and the notion [2] of employing social networks as coordination sites for such systems, we have designed and implemented a system that combines Web services with a population of lightweight JavaScript nodes, run by volunteers inside their Web browser, to enable unobstructed Web access. Web services are employed as coordination and storage sites. Users of our system upload "access requests" for Web pages to a coordination site, e.g. Twitter, and receive the corresponding content from a storage site. According to the simulations performed in [26] (2006), popular sites can form populations ranging from 1,000 to more than 100,000 active browsers at any given time. In our system we employ such firepower of potential volunteers to serve Web access requests placed by users, and upload the respective payload in uncensored sites that host user-generated content. Thus, we enable users to bypass censorship-oriented network restrictions and access Web information indirectly.

The major advantages of our approach are twofold. First, it is hard to blacklist its usage. There are thousands of services where a user can post contents. Social networking sites, blogging and micro-blogging sites, collaborative tools, forums are only a few of the places where users can post requests. To stop the operation of our system, access to a great part of the Web would have to be blocked [1]. We acknowledge that major sites like Twitter could be prime candidates for blocking and while we have used such sites for our proof-of-concept implementation, our design is very flexible and can easily employ a variety of sites. Second, our system's operation exhibits unobservability; our network traffic patterns and application-level behavior align with the profiles of everyday users interacting with such Web services. Therefore, we argue that the actions of our system cannot be distinguished from standard Web activities.

This paper presents the following contributions.

- We design and implement a censorship-resistant Web access mechanism. We employ a plethora of existing Web services in a dynamic design to enable unobstructed Web access, which renders our system *highly resilient to black-listing*. The operation of our system is built on requests and responses that are identical to those of normal users both at the network and application level and, thus, users of our system *cannot be distinguished from other users*.
- We provide a *lightweight* design where Web volunteers that serve requests are not required to install any software; their functionality is achieved through the execution of specially crafted JavaScript code.
- We evaluate the performance of our system and show that we obtain access times comparable to those of current state-of-the-art anonymity and anti-censorship systems.
- We evaluate the security of our system and show that we are able to hold the necessary properties of *availability*, *unobservability* and *correctness* in a series of censorship scenarios.

[1]For instance more than 60% of the Alexa top 500 sites can be employed by our system.

- We present a novel technique for *bypassing the same origin policy* which allows our system to achieve read-only access of foreign domains. Traditionally, Web application code running on a user's browser is not able to access content from foreign domains. We provide a way to do so without relying on any bugs or security vulnerabilities.

2
Design

2.1 Threat Model

We consider the following while designing our system. Our adversary is a censorship authority that filters outbound access of a network towards certain Web destinations or inspects inbound content and alters or removes parts of it. We assume that the censorship authority wishes to allow some Internet access to its clients, that it can inspect and block traffic at the network level, and employ deep packet inspection. Finally, we consider our system known to any adversary.

2.2 System Architecture

Our system relies on a swarm of lightweight nodes running inside the Web browsers of volunteers. A client who desires uncensored Web access implicitly requests that volunteers fetch Web pages on his behalf. To protect the volunteers, our design gives them the ability to opt-out from serving sites that may host controversial content (section 3.3). Coordination is achieved through a Web service, e.g. Facebook or Twitter, which is employed as a “command and control” (C&C) module. Our system operates and coordinates itself unattended. We require human administrative intervention to update the list of Web services for our system to use and deploy the node-enabling code to more Web sites. Web content is stored in uncensored user-content hosting services, such as Pastebin, serving as Object Stores. Our design is displayed in Figure 2.1.

Assume there is a client who lives inside a restrictive network and wishes to access Web information, otherwise censored or blocked entirely. Instead of accessing Web pages directly, he instructs his browser to use a local proxy of our system.

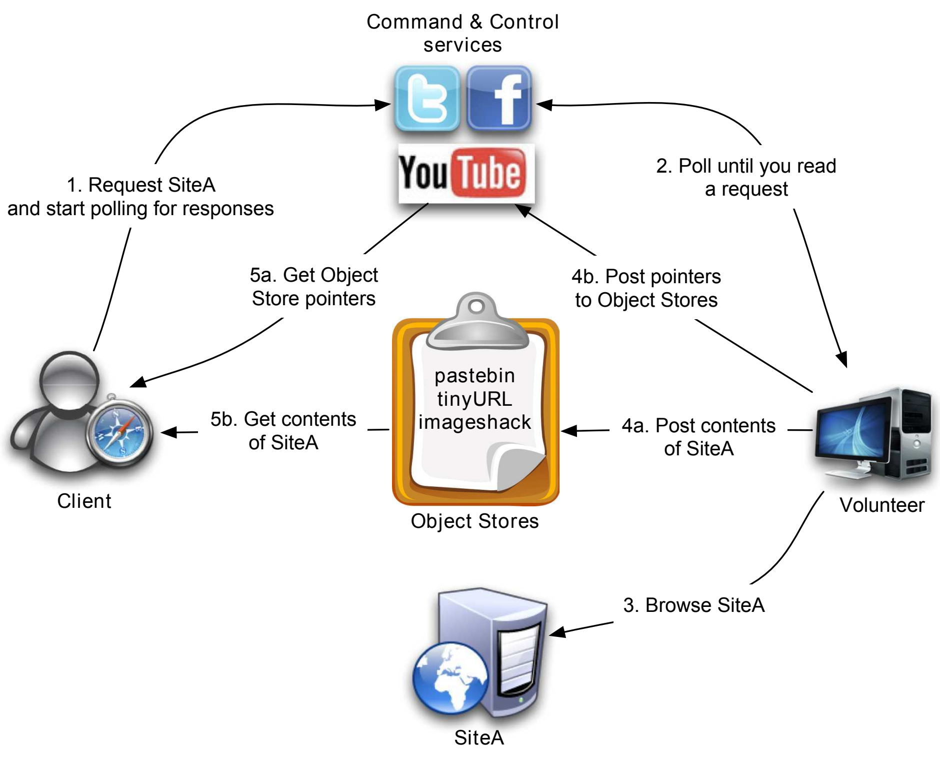
Command & Control services
1. Request SiteA and start polling for responses
2. Poll until you read a request
5a. Get Object Store pointers
4b. Post pointers to Object Stores
pastebin
tinyURL
imageshack
5b. Get contents of SiteA
4a. Post contents of SiteA
Client
Object Stores
Volunteer
3. Browse SiteA
SiteA

FIGURE 2.1: Architecture of our system. The client puts requests for sites he wants to access in a C&C service (step 1). Volunteering nodes poll the C&C service (step 2) and when a request is received they browse the requested sites and retrieve their contents (step 3). Once content is retrieved, they store it in an Object Store (step 4a) and post a pointer to the hosted content back in the C&C service (step 4b). Finally, the client gets the pointers from the C&C service (step 5a) and accesses the Object Stores to get the content of the pages he has requested (step 5b).

The local proxy is a modified version of a standard HTTP proxy and translates all URL requests, from the client's Web browser, to posts (access request messages) towards the C&C service. For example, if the client wishes to access the front page of the CNN Web site, i.e. `http://www.cnn.com/index.html`, then the following steps will take place:

1. The respective URLs from all HTTP access requests, that the client's Web browser will issue, will be intercepted by the local proxy on the client's end and uploaded to the C&C component of our system. [1] In the most naive form, these posts will simply be the requested URLs. The amount of sites

[1] In our implementation, we have used a form of prefetching. The volunteering nodes parse HTML content, recognize image files and other embedded content such as CSS files and scripts, upload them to the Object Store, and rewrite their address in the served HTML code for the client's proxy to access directly. Thus, the proxy only issues a single request for the Web site rather than a batch.

that can be used as C&C is enormous, as a multitude of forums, blogs and social networking sites are suitable.

Service nodes of our system are volunteering Web surfers that, during their daily activity, visit sites carrying our specialized JavaScript code and, by executing it, serve the Web access requests of clients. When a browser opens a Web site, standard practice is to render the HTML in its window and execute any JavaScript code, which usually adds functionality to the site. This is how our own code is also executed. While this code takes advantage of JavaScript and HTML capabilities, it does not depend on any kind of browser or page-specific vulnerability.

2. Volunteering nodes periodically access the C&C service to poll for new posts. When a new access request appears on the C&C (in our case `www.cnn.com/index.html`), nodes make an effort for providing an answer.
3. Volunteering nodes access the URL referenced by the access request and retrieve its contents (HTML text, CSS styles, images, scripts).
4. After the content is retrieved, two tasks are performed.
   (a) First, the nodes store the content in services that can host user-generated content. We call these services "Object Stores".
   (b) Second, the nodes return pointers to the stored content back to the C&C.
5. In the final step, the client's local proxy periodically reads the C&C, waiting for replies to his requests.
   (a) Once such replies appear on the C&C, it retrieves them to receive pointers (URLs) to the content stored in the Object Stores.
   (b) Once the pointers are known, the proxy contacts directly the Object Stores in order to get the content of the requested page.

The way the client experiences the entire process is that he types a URL in his Web browser, waits and then the content he has requested begins rendering in front of him. This process bares no difference to accessing a URL without help from our system except for an added delay, presented in section 4, and of course the fact that, without our system, there is the possibility that the respective content is censored or not received at all.

2.3 Advantages of our Approach

First, the use of existing Web services as the medium through which all requests are propagated, provides a *flexible infrastructure*. Our design is not bound to a specific Web service but can use any user-content hosting site. With a plethora of suitable services being available and new ones continuously emerging, our system is *highly* *resilient against black-listing*, thereby **available** in hostile environments. Even if adversaries blacklist certain services, our system can "hop" amongst a very large number of sites, several of which are used by hundreds of millions of users around the world. Therefore, to block our system one must block access to a great part of the Web. We argue that a censorship authority will be unwilling or unable [4] to do so.

Second, our design is **lightweight**. We employ existing Web infrastructure rather than demand additional setup overhead. Also, our nodes operate within the Web browser and require no additional software installation or configuration. Therefore, our intuition (supported by experimental data [26]) is that the population of volunteers joining our system will be at least comparable, if not much larger, than current volunteer-based approaches. Moreover, as clients and volunteers never communicate directly, nor do they require any type of handshake between them, our design accounts for a *high churn rate* in terms of volunteering nodes entering and leaving our system. At the same time, clients of our system simply have to acquire a copy of the local proxy and set their browser to use it. No other configuration is necessary and the internals of our system remain transparent to them while using it. As far as they are concerned, our system is yet another Web proxy.

Third, we consider the **unobservability** of our system. Communication in our system is conducted through standard HTTP traffic. The use of Web sites oriented towards hosting user-provided content creates enough noise for our operations to blend-in. For instance, in Twitter, with desktop clients polling the service once a minute and 25% of messages being uploaded containing URLs [14], we argue that such environment is adequate for us to operate efficiently and at the same time blend-in. Moreover, as clients post the URLs they want to access on popular sites used as a C&C, these requests do not raise suspicion as they blend in with millions of posts made by legitimate users towards these sites. As the general rule for such URLs is to be short (the product of a URL shortening service such as bit.ly or tinyURL) we comply with this trend and our proxy shortens its URLs too, prior to posting. Similarly, all posts from volunteering nodes toward the Object Stores, hosting arbitrary user-generated content, cannot be distinguished from other posts.

2.4 Limitations

Our system offers unobstructed access to Web sites but is unable to support Web sessions (i.e. HTTP cookies and HTTP parameters). Although, we offer basic (HTTP POST) Web write functionality, we consider that the ability to access information is the primary issue in cases of censorship (e.g. oppressive regimes) and that read-only Web access is a fair trade-off for black-listing resilience.

Furthermore, our implementation does not currently offer a solution for embedded objects such as Flash, Java and ActiveX. Objects like these cannot be encoded in some way using in-browser JavaScript or be uploaded to a content-specific Object Store. We measured the extent to which this limitation can break the user's access to information. We analyzed the top 1K sites from Alexa and recorded the ratio of these three types of embedded objects against HTML and image elements. 89% of the sites do not contain any embedded object while in the worst, for us, case one page had 1.5% of its elements embedded.

2.5 Use Case

In general, we consider that multiple instances of our system may be active, e.g. operated by initiatives or individuals promoting unobstructed Internet access. An instance is defined by a set of Web sites, employed as coordination and storage components, and a set of volunteers. Although the latter set may be dynamic, its population is defined by an instance-specific volunteer-enabling site. Volunteers lie in censorship-free networks and wish to participate in our system so as to fetch Web content on behalf of the clients behind restricted networks.

For a volunteer to join our system, he must locate a site carrying our node-enabling JavaScript code and have his browser execute it. We consider the straightforward case where there is a public Web portal that the volunteer knows or can trivially find out and through that discover which site to visit in order to join our system.

The next step is for a client to join our system. The client has to acquire a copy of our modified HTTP proxy (pre-configured to join our system), run it locally and configure his Web browser to use it.

3 Implementation

In this section we present the implementation of our proposed system. While our design is flexible and can use a very large number of Web services, in our prototype we build our platform around two popular sites; Twitter and Pastebin. Let it be noted that, as we treat Web services as read/write stores, any service offering such functionality can be seamlessly integrated in our system as long as an intermediate function is implemented to translate our system's put/get requests into service-specific actions. In Section 3.1 we provide details on the components, outlined in the previous section, that comprise our system. In Section 3.2 we elaborate on several issues that have emerged during the implementation process and the measures taken to resolve them.

3.1 System Components

Client's local Proxy. It is a standard HTTP proxy modified to support the necessary functionality for joining our system. It is responsible for translating the client's HTTP requests into access requests for our system, i.e., requests that are to be served outside a censorship zone through our system's infrastructure. Our modification lies in the fact that the proxy does not directly contact the remote server. Instead, it places the client's request on a C&C queue where it will be served by a volunteering node and fetch a reply once available. In our prototype implementation, a modified instance of Privoxy 3 was employed.

The reply to an access request is assembled like so; a volunteering node, after serving the request (steps 2, 3 in Figure 2.1), places the returned content in the Object Store (step 4a) and adds a reference in the C&C queue (step 4b). The proxy uses that reference (step 5a) to download the content from the Object Store and deliver it to the client (step 5b).

Command & Control. This component acts as a coordination point between clients and volunteering nodes. Its primary function is that of a queue where clients place, through their local proxies, requests for specific URLs. In turn, volunteers respond with references to objects containing the requested content. For security reasons, which are discussed in section 5, we require that the C&C queue is a global *append-only queue*. This means that it is not possible to remove elements from the queue.

For our implementation we considered many viable candidates for the role of the C&C and decided to assign this role to Twitter, a popular message exchange site, that organizes its content in a queue-like way. To achieve write permissions for all volunteering nodes and clients and still maintain our requirement for an append-only queue we employ the use of hashtags (#). Each volunteer and client is equipped with a dedicated Twitter account for the purpose of participating in our system. They upload messages (tweets) to their Twitter profile so as to accommodate steps 1 and 4b of Figure 2.1. Each message includes a hashtag identifier for the C&C queue, e.g. #1337_queue. A read-only view of the queue is accomplished by searching in Twitter for the specific hashtag (e.g. http://twitter.com/#!/search/%231337_queue). In that case, Twitter returns, in chronological order, all messages uploaded by any Twitter user containing that specific hashtag. That way, although each volunteer and client writes in his own account, all volunteers and clients of our system read from the global queue. Since each volunteer has write permissions only to his own account, he is unable to delete messages from the queue other that his own, hence the global append-only property. [1]

Although an existing infrastructure may operate and organize itself unattended, we require human intervention to add more Web services to our system, for serving as C&C and Object Stores if current ones are blocked or taken offline, and also deploying our node-enabling code to more sites for volunteers to find. We consider that multiple instances of our system may be active on the Internet at the same time, run by individuals between different initiatives. These people will take up the role of maintaining an instance if and when it is necessary.

Object Store. This component is used for the hosting of content objects, i.e. the content of the pages the client requested. Several existing Web services are eligible for this role. These services, and others of their kind, pose an exact match to our needs. They are freely available tools for hosting that expect high loads of traffic and provide a highly available point on the Internet that hosts user-generated content for a short (or long) period of time while allowing direct reference links towards it. We have employed Pastebin, offering a content-uploading API, as an example Object Store.

[1]In sites where users comment, the C&C queue is a page and volunteers use individual accounts to write.

Volunteering node. The concept of a node is manifested as a volunteer's Web browser, executing our carefully crafted JavaScript code which polls the C&C for page requests from clients, serves them and returns the corresponding content to the Object Store. The code is hosted on a volunteer-enabling Web page and contains hard-coded any configuration, including the addresses of Web sites used as C&C points and Object Stores. For each such site a function is present following the naming convention *<site>_read* and *<site>_write*. The diff between functions for different sites is merely variable names and parameters in the HTTP GET/POST (read/write) requests that need to take place. The code takes advantage of HTML and JavaScript features but does not require any kind of browser or page-specific vulnerability; using JavaScript we are able to dynamically manipulate the DOM of an HTML page so that we may add or remove elements such as iframes. The sequence of iframe (and other DOM objects) creation, manipulation and the association between them implements the functions of a node.

The first thing the code does, once loaded in the Web browser's window, is to poll the C&C site for URL requests to serve (step 2 in Figure 2.1). The polling interval is customizable and has been set to one minute, which is consistent with the polling rate of Twitter desktop and mobile software applications. The code opens an HTML iframe to the C&C to read the request queue. We describe in detail how it is able to access the iframe content from a different domain in section 3.2.

The volunteering node takes up the first available queue request and initiates the serving procedure. As multiple idle nodes may exist, it is expected that more that one of them will simultaneously serve a single request. This kind of redundancy is a desired property to provide stronger security (correctness) and is discussed in detail in section 5. A client's local proxy is responsible for adding a "terminate <request>" message to the queue so that nodes do not attempt to serve it any more. If the client fails to do so, endless loops are avoided by having each volunteer not serve twice the same access request from the C&C queue.

Once a client's request is selected, depending on its type, either HTTP GET or HTTP POST, the appropriate course of action is determined. Our main goal is to support HTTP GET requests to facilitate Web access read requests, while POST requests are usually for write requests.

- If it is an HTTP POST, a blank iframe is created where an HTML form is populated with the appropriate target and data fields and submitted. All the required information is either part of the user request in the C&C queue or, if too long, stored in a content object and referenced by the request.
- If it is an HTTP GET towards a target URL, then the client expects some content so it must be fetched, placed in the Object Store and referenced in a reply back to the client. The code spawns an iframe in which the target URL is loaded in such way that its elements are accessible to the volunteering node. We provide more details on how this is possible in section 3.2. Once the iframe has finished loading, its HTML text and any references or embed-

ded content (e.g. images, CSS styles, scripts) are copied to local JavaScript variables and later included in a content object.

The plain text elements of the resulting content are uploaded in ASCII-encoded form to an appropriate Object Store hosting site. We selected Pastebin but any user-content hosting site will do, including places such as comment replies in articles and videos, in sites like YouTube. The volunteering node places an HTTP GET request towards the service's REST API and reads back the address where the content is stored. Pastebin offers short hash-based addresses but for other hosting services one may employ short URLs to acquire very short URL pointers.

At this point, the client's target URL has been fetched, its content inspected and its elements transformed into content objects. Finally, a blank iframe is set and an HTML form, containing the request reply with the necessary references to the Object Store, is used to write the information to the C&C.

We assume that volunteers execute our node-enabling code in an isolated instance of their browser (or different browser, e.g. Internet Explorer when they are using Firefox) so as not to interfere with potential existing sessions with some of the Web services employed by our system, e.g. already logged into Twitter. We discuss in section 5 that even if that assumption does not hold, there will be no privacy risk for the volunteers and only a minor discomfort as they will be logged off their current sessions with the Web services that our system has to employ. Nevertheless, recent work from browser vendors [15] enables such isolated instances ("profiles") to be present concurrently and operate seamlessly in separate browser tabs.

3.2 Technical challenges

Cross-Origin Bypass Mechanism. In the core of the volunteering node's code resides a primitive which supports the majority of actions taking place. The cross-origin restriction policy, enforced with security in mind, prevents a page under domain A to spawn an iframe to a page under domain B and then access its elements, like HTML code or images; exactly the operations our node has to perform. Therefore, it became crucial to bypass this restriction. The idea is that a third-party page under domain C, capable of echoing back the content of foreign pages, is used to host both pages A and B under domain C. That way, a script (originally hosted under domain A, now domain C) is allowed to access HTML content (originally hosted under domain B, now domain C).

A plethora of Web-based proxies [2] exists and we have successfully employed them to get our code and a target page under the same domain; a user inputs a URL to the service and that URL's content is echoed back from inside the service's domain and so is our code. The case of code running in volunteering nodes employing a proxy at some point during its operation is different from client using

[2]e.g. 40K proxies on `http://proxy.org/`

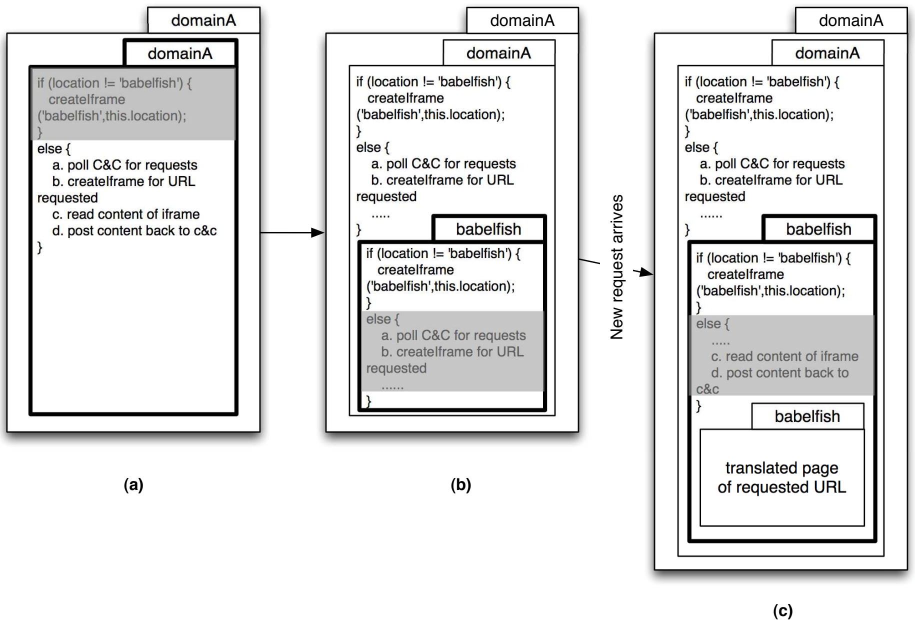

FIGURE 3.1: The cross-origin bypass mechanism employed by volunteering nodes. Code running at each step is highlighted with gray and the active frame is painted in bold border. Initially, the JavaScript code from domain A is loaded. At step (a) the code checks the domain, sees that it is not running inside the translation domain and spawns an iframe. This iframe asks from the translation engine to translate the code itself. Once the translation is finished, the code starts running again but now it resides in the Babelfish domain (step (b)). The code polls the C&C until a request is received. It then spawns a new iframe, also belonging to Babelfish domain, to translate the requested URL and retrieve the content from the translated version (step (c)).

standard proxies to access the Web; *volunteers reside outside the restricted zone and are not affected by an adversary's attempts to locate such public proxies and block them.*

Apart from such proxies, we explored the use of page translation services such as Yahoo's Babelfish or Google Translate. A translation service needs to fetch and host the content on its own space since it is supposed to present a version where elements from one language have been substituted by elements from another. We took advantage of this tool to place our JavaScript code under the same domain as a foreign page, by placing both under the domain of the translation service. At this point, one could argue that the foreign page would be altered due to the translation substitutions, an unwanted side effect of the same-origin bypass. This issue is mitigated by instructing the service to translate from and to languages that are not present in the remote page, for instance from simplified Chinese to traditional

Chinese for an English page. This will effectively prevent any substitutions and echo the original content.

The process of bypassing the cross-origin restriction is shown in Figure 3.1. With this mechanism at hand we proceed as follows;

1. once the JavaScript code is loaded in a volunteering node under the code-hosting domain (domain A), it spawns an iframe towards the Babelfish translation service, using the HTTP GET API to pass the necessary parameters to effectively translate its own page and bring its location under the Babelfish domain (phase a in Figure 3.1).
2. The translation service echoes back all HTML elements including our original JavaScript code for the node in its entirety. In other words, our code runs in the volunteer's browser while the designated domain is the one of Babelfish (phase b).
3. At any time the code wishes to poll the C&C or fetch the content for a request or generally reach remote pages, it will spawn an iframe towards the translation service, supplying the remote URL (phase c). That iframe, once loaded will contain the remote content and will also be located under the Babelfish domain, essentially co-located with the JavaScript code attempting to access its elements. In that case, the cross-origin restriction does not apply since our code, originating from the Babelfish domain, attempts to access an element located under the same domain.

Frame Busting. During the course of our experiments, we came across sites that tried to make sure they were not being accessed from within an iframe. For that matter, they check whether the top frame in the current browser window is equal to their page's address and if not, they reload themselves in the browser effectively shedding the iframe. Our JavaScript implementation relies on recursively spawning iframes and, thus, had to employ a countermeasure. For that matter, the `onbeforeunload` event of DOM element `window` was overwritten to call a function of our own. This event occurs right after the remote site B has replaced the window location and the window is about to reload. Our function, triggered right before the reload, effectively aborts this attempt.

Avoiding Caching Effects. The volunteering node's operation involves the creation and removal of many document elements. Browsers usually attempt to cache such elements when they appear to point to the same location or have identical properties. This is an undesirable effect, especially during the polling phase, as a node may miss new requests. To resolve this issue, at the end of every URL we include a random variable, an epoch number, making each URL unique while not interfering in any way with the Web server's response.

3.3 Exit Policies

Censorship resistant Web access is critical to enable unobstructed access to information. While a noble mission statement, such system may be abused to cloak access to illegal or frowned upon content. To protect the volunteers from serving such cases, we have implemented exit policies; a pop-up HTML text box in their Web browser gives the volunteers the option to define “allow-access” (and drop everything by default) or “deny-access” (and serve everything by default) directives using regular expressions (e.g. allow-access http://*/foo* or deny-access http://example.org/foo/bar.html). Their preferences are stored in the form of HTTP cookies [3]. The JavaScript code run by volunteers is clear-text and may be inspected by simply selecting the “view source” browser action on the site serving the code. Thereby, anyone can be sure that his preferences are being taken into consideration and that the node-enabling code does not perform differently than advertised.

[3]We believe that a single HTTP cookie will have enough capacity to hold the preferences of most users, as the maximum cookie size is 4KBytes. However, multiple cookies may be employed.

4
Measurements

In this section we measure the performance of our system. Our goal was not to build a low-latency system. Nevertheless, we compare the access time of our system with a set of open HTTP proxies and the Tor network to get a feel of its access times. We acknowledge that we are comparing two loaded systems with an unloaded one, but we believe that even so we are able to demonstrate that our system's performance is comparable to the other two, most commonly used today for similar purposes.

For the testbed, the role of a volunteering node was assumed by a PC at first in our lab (1Gbps symmetric Internet connection) and later in a residential environment (2Mbit/384Kbit DSL connection). The PC's browser visited a page, hosting our JavaScript code, in a test Web server we had set up. Clients of our system were simulated by 50 Planetlab nodes, scattered around the Internet. Each node ran a local instance of our system's local proxy and requested, through the *wget* tool, the contents of Alexa's top 10 pages. At any time a single volunteering node was active (Gigabit or DSL instance) and a single client (Planetlab node) placed requests towards the C&C in a serial way. For each request, we measured the time elapsed from the moment the user executed the *wget* tool to the moment it returned the content. Each client requested, ten times, the pages contained in Alexa's top 10 list. Overall, for each of the top 10 pages, we had 50 puppets fetching it 10 times, resulting in 500 measurements from which we calculated the average access time. In Figure 4.1 it is clearly shown that the access times of our system are directly comparable to those of Tor and open HTTP proxies. [1]

[1]Regarding the performance of open HTTP proxies, we polled a popular site featuring rapidly-updating proxy lists and tested more than 1500 proxies in a period of 5 days.

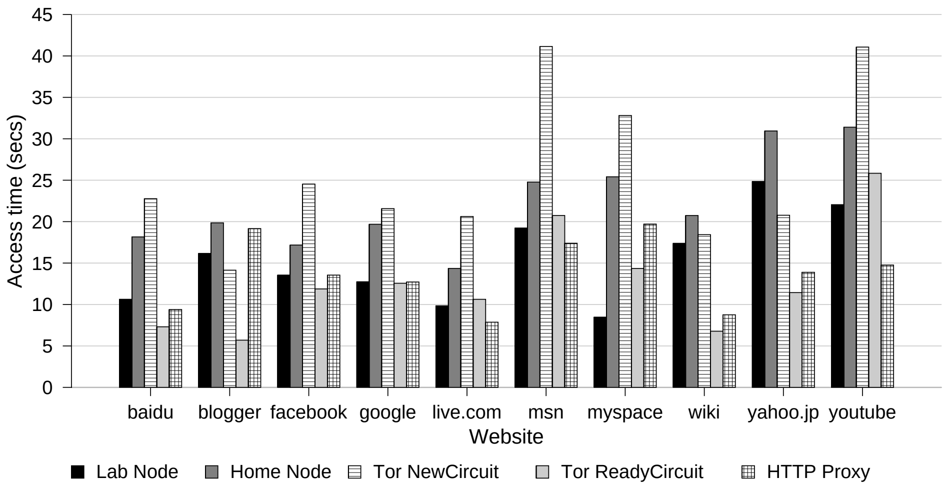

FIGURE 4.1: Access time for visiting the top 10 sites of Alexa using nodes of our system located in our lab and a residential connection and using Tor with and without a pre-established path.

5 Attacks

5.1 Security of the Clients

Considering our goal of highly-resilient to black-listing, censorship-resistant access to Web content, we are interested in the *availability*, *unobservability* and *correctness* of our system. We discuss how these properties hold under different scenarios; when the censorship adversary is *close to a client*, *close to the infrastructure* or *close to both*. In the first case, the adversary is in the same network as the client and monitors his communication. This is the default scenario where users reside inside a restricted zone which is controlled by a person or organization. In the next case, the adversary has joined our system as a volunteer and is thereby able to know the location of the C&C site and Object Stores. That way, he monitors incoming requests from clients and outgoing replies. Finally, if these two cases are combined, the adversary is able to monitor the network traffic of a client and gain insider's knowledge of our system at the same time.

5.1.1 Blocking our System

An adversary may try to black-list and, thereby, block our system from communicating with his network. Such a ossibility comes in conflict with the requirement of **availability** which states that our system should be able to serve clients despite the efforts of a censorship authority to block it.

1. *Adversary close to client.* We employ non-censored Web services for the purpose of coordination (C&C) and the purpose of storing (Object Stores) the Web content to be accessed by the clients. At the same time, access-request serving nodes reside outside the censored zone. As a result, our

system remains available despite an adversary's filter which blocks a set of Web sites and perhaps known anti-censorship proxies.

2. *Adversary close to our infrastructure.* Although an adversary may be aware of the Web sites that comprise the infrastructure of our system, we argue that he is unable or unwilling to block them all from his network without greatly impairing the Web browsing experience of his users, as he will have disabled a great part of the Web. At the same time, our design is dynamic so that it may incorporate a plethora of available Web sites and services for its purposes. As a result, our system provides resiliency in the face of blocking attempts by an adversary.

5.1.2 Observing and Identifying the Traffic of our System

An adversary may try to identify the communications of our system on the network and associate a user or volunteer with it. However, we acknowledge the need for **unobservability** so as to not exhibit any such distinguishable activity.

1. *Adversary close to client.*
   - If the adversary does not perform deep packet inspection, our client's communication does not raise any red flags as it is directed towards a non-censored site. Moreover, the traffic patterns of the communication (e.g. polling rate) do not deviate from standard behavior. For instance, Twitter desktop applications poll the service for new messages every minute and read or write in bursts. As long as our client does the same to read or write to the C&C queue, there is no reason why this would differentiate him from other Twitter users in the same network.
   - If the adversary does in fact perform deep packet inspection (one of our assumptions in the threat model), we argue that there is a trend [13] for more and more Web services to support HTTP over SSL. This practice is justified by the fact that the overhead of a secure communication can be optimized [7] and at the same time a series of attacks [11] target the unencrypted communication of users of popular Web services such as Facebook and Twitter. Therefore, as our client communicates with the C&C and Object Stores over a native encrypted channel (Facebook, Twitter and other major sites already support that), the adversary is unable to inspect the payload.
   - If the adversary does perform deep packet inspection and no available Web service supports secure HTTP, we consider the use of payload packing in a self-decrypting obfuscated JavaScript block that offers some protection so that it is very difficult for meaningful content to be recovered by the adversary in real time. User requests and returned content are both wrapped in a JavaScript echo function, which is then compressed

and obfuscated. That way, any kind of static analysis is unable to recover meaningful content. Furthermore, there are techniques that render the use of content signatures inefficient [8], any attempts for lightweight dynamic analysis ineffective [9] and any analysis in a virtualized environment detectable [3]. Finally, (obfuscated) JavaScript in Web packets is not suspicious but nowadays rather mainstream. Therefore, we argue that even if the adversary is aware of our obfuscated format, it is very hard to identify and uncover the actual message in real time. However, an adversary can capture such packets and unwrap them offline which could have future repercussions for the client. Preventing the adversary from doing so at a later time is beyond the scope of our system.

2. *Adversary close to our infrastructure*. An adversary is not in a position to identify individual clients by simply inspecting their communication as it arrives in a Web service employed by our infrastructure. Client messages to the C&C, asking for Web access, do not carry any identifying information, unless for instance the client's name or username is part of the URL he wishes to request access to. Moreover, as volunteering nodes never contact a client directly but indirectly respond to his access request, an adversary will not be able to trace the network (e.g. IP) address of a client by pretending to be a volunteer in our system.

3. *Adversary close both to client and infrastructure*. It is very hard for the adversary to discover the content of a client's communication, as already discussed. The adversary may rely on correlating network activity from a client towards a Web service and changes in the C&C queue or Object Stores of our system. Such correlation relies on the timing of events and we argue that there is too much noise from unrelated events on a Web service for an adversary to do so. For instance, in Twitter the noise not only comes from user access patterns but also from content patterns; considering that 25% of all Twitter messages contain URLs, the URL-access requests being uploaded as messages by our system are able to blend-in.

5.1.3 Poisoning Served Content

Theoretically, an adversary could join our system as a volunteer and serve censored (poisoned) responses. It is important, in order to achieve **correctness** in such system, that the information served is not tampered with or if tampered, such incidents can be detected.

We deal with such a scenario and achieve *correctness* like so; by design, multiple volunteering nodes will post (redundant) replies to a client's Web access request. From these replicated replies, the client's proxy selects N at random, where $N >= 3$. Then their content is stripped from dynamic elements such as cross-domain iframes and scripts and compared [20] to identify possible deviations. If a majority of these replies agrees on the content, the situation is resolved by rejecting the different views. If not possible, the client is notified by including a warning frame in the content loading in his Web browser. Thus, the probability that a majority of poisoned replies is selected by the client's proxy is $\frac{poisoned\_replies}{all\_replies-g} * \{\frac{poisoned\_replies-b}{all\_replies-g-b}\}(N/2)$ [1]. However, as clients are not tied to specific volunteers (and vise versa), even if an adversary manages to poison the majority of replies for a specific access request, the next request from the same or other clients is an independent event that may or not result in favor of the adversary. We acknowledge that the security of this scheme holds only as long as $poisoned\_replies << all\_replies$. For that matter, as volunteers subscribe to the C&C Web service with individual accounts, so as to upload their replies, a client's proxy accepts a single reply from any account in the C&C and no account is selected twice for the same reply. That way an adversary would have to create a large number of accounts (must be larger than the average number of active volunteers) so as to influence the majority vote on content with good probability. However, such act is not trivial. The issue of automatic account creation is not new to Web services. CAPTCHA mechanisms [**?**] are employed by roughly 100,000 Web services [**?**] and that number if constantly increasing.

Overall, we assume a usage model where it is trivial for an adversary to join our system as a volunteer through a public site. Under that model we have argued how our system is able to withstand attacks against its *availability*, *unobservability* and *correctness*. For increased security, we consider a reputation-based information dissemination mechanism such as Proximax [27] for offering a more efficient way for volunteers and clients to discover and join our system, instead of publishing the information for anyone, including the adversaries, to see.

5.2 Security of the Volunteers

5.2.1 Drive-by Downloads

Drive-by downloads [30] [29] are a general form of attacking Web users through their Web browser; an HTML page contains code, typically written in the JavaScript language, that exploits vulnerabilities in the user's browser or plugins. If successful, the attack downloads malicious software (malware) on the victim's machine. When a volunteer attempts to serve a client's Web access request, during the rendering of the corresponding page, his browser also executes any JavaScript or other code, potentially malicious, present in the page.

The problem of drive-by downloads is outside the scope of this paper but we consider that volunteers have the option to specify exit policies, thereby white-listing or black-listing domains and pages they may fetch on behalf of clients. Therefore, they can protect themselves, e.g., by using an automated process to

[1]where g is the number of legitimate (correct) replies chosen so far for this access request and b is the number of poisoned replies chosen so far. Also $all\_replies > N$ and $N >= 3$.

populate their "deny-access" exit policy through an existing black-list, such as `http://www.malwaredomains.com/`, of domains known to serve malware. An alternative approach is to employ a security plugin, such as NoScript [2], which by default prevents the browser from executing arbitrary code.

5.2.2 Web Privacy

As mentioned earlier, volunteers are encouraged to use an isolated instance of their Web browser or a separate browser when participating in our system. Such isolated instance may be parallel to the volunteer's regular browser, so he may simultaneously surf without problem. The reason behind this is that our system may overwrite some of the user's cookies or other browser state and subsequently cause him to be logged out of a Web site, e.g. overwrite his Twitter credentials with the Twitter account that has been set up for writting to Twitter as a C&C queue.

One could point out the case of a volunteer participating in our system through his default Web browser and then serving client Web access requests using the volunteer's own Web credentials. A Web browser will send any (authenticating) cookies to a site upon visit. Therefore, one could argue that a volunteer, fetching a Web site to serve an access request, will use his own cookies to render and return a private view of the site as available only to him.

However, this is not possible and the volunteer is safe. As already described in section 3, the node-enabling code executed by the volunteer is bounded by the same-origin policy and cannot access a Web page under domain B directly. We employ our *cross-origin bypass mechanism* to bring the target page under the same domain C as our code. As a result, the Web browser will never send the volunteer's original cookies for domain B and his privacy is preserved.

5.2.3 Laundering Web Access

There is the case of a client, malicious or not, requesting access to pages of illegal or disputable content. As volunteers will try to serve that request, it will be their Web browser and network address asking for that content. This could have legal or other implications.

However, volunteers are protected through the use of exit policies, discussed in section 3.3. Volunteers are able to specify, through regular expressions, access request rules and thereby refrain from serving or serve only specific Web pages.

5.3 Security of the System

5.3.1 Preventing Abuse

We consider the scenario where an attacker leverages our system to carry out Denial-Of-Service attacks against Web sites; he may flood the C&C with access-

[2]http://noscript.net

requests that the volunteers will try to serve, resulting in excessive traffic towards a particular target Web site.

While mitigating such attack is beyond the scope of this paper, we consider approaches like Hashcach [**?**] where a CPU-intensive function, executed by the client, computes a token which can be used as a proof-of-work. The server is in a position to verify the validity of such token. In our system, client's would calculate such token and it would be verified by volunteering nodes prior to serving an access request. To prevent an adversary from pre-computing a large amount of tokens, a short-lived public seed is employed so as to automatically invalidate tokens pre-computed in the past.

6
Related

Mix Networks [23–25, 31, 32] consist of an overlay of nodes that create hard-to-trace anonymous communications. The client uses such system as a proxy to indirectly reach Web servers and download information, therefore bypassing restrictions at the local network or at the server's end. An other alternative are (open) Web proxies [21].

However, such networks traditionally use a publicly listed directory of nodes comprising its infrastructure (so that users may connect to them) and, as a result, these nodes can easily be blocked at the network level [19]. Even unstructured peer-to-peer networks rely on a bootstrapping phase with some hard-coded super-nodes or known Web caches (which is the same problem as before) or employ random peer discovery via flooding or network path walks at known ports that can also be filtered out. Tor has introduced Tor Bridges [17], which are relays not listed in the public directory and run on non Tor-standard ports, commonly over TCP 80 or 443. To discover such relays, users may contact the address distribution mechanism hosted on the Tor Web site or send an e-mail to the Tor project. The mechanism is designed to prevent attackers from acquiring the entire list of bridges, by serving three IP addresses per user's network address, which is logged for a period of time. We observed that all hosts that belong to the same /24 subnet and query the site within a certain time window will receive the same list of bridges. Apart from blocking access to the Tor Bridges site, an administrator, with a large enough number of unique source IP addresses, could eventually acquire a great portion of the list of bridges. As a matter of fact, by fetching the Tor Bridges site through open HTTP proxies and a distributed system we have managed to acquire all 30% of the alleged 700 [18] available bridges within the first hour and almost all of them in the next couple of days.

At the same time, in our approach, we leverage Web sites as coordination and storage points and disseminate their location for volunteers to join and clients to discover our system. However, unlike traditional proxy nodes, there is no way to block volunteers as they reside outside the censored zone and do not contact our clients directly. Furthermore, we argue that the vast number of Web services and their popularity makes a censorship authority unwilling or unable to block them all as that would greatly impair user connectivity to the Internet.

Feamster et al. in [28] propose Infranet, a system where cooperating Web servers provide clients with access to censored sites while continuing to host normal uncensored content. Concurrently and independently from our work, Burnett et al. in [22] have extended the concept of Infranet to a system where volunteers, running specialized software at their end, act as intermediaries in the communication of users with Web servers. They coordinate and exchange content by steganographically encoding information in images and videos and uploading them to user-content hosting sites.

In both of these systems a dedicated distributed infrastructure is required in which participating nodes (volunteers) need to explicitly run specialized software that handles the steganography-based communication and serves the Web access requests of clients. Moreover, such infrastructure must be relatively stable as a non-trivial rendezvous protocol must take place between a volunteer and a client for the latter to discover the first.

On the other hand, our approach is lightweight since nodes, serving access requests, operate within Web browsers executing carefully crafted JavaScript code. As a result, volunteers do not need to incur any software installation or configuration overhead and simply contribute during their every day surfing activities, for instance by keeping an extra tab open in their Web browser. Moreover, since a client using our system is not tied to a specific volunteer, nor is he required to perform some form of protocol handshake, but his requests are processed independently by the first available node, our system can cope with a high churn rate of participating nodes.

Puppetnets [26] rely on websites that *coerce* Web browsers to (unknowingly) participate in malicious activities. Puppetnets exploit the high degree of flexibility granted to the mechanisms comprising the Web's architecture, such as HTML and JavaScript. A Web site under the control of an attacker can thereby transform a collection of Web browsers into an *impromptu* distributed system that is effectively controlled by the attacker.

Our work is based on the puppetnet concept, that is to run JavaScript code snippets in a population of user Web browsers so as to form a distributed system. However, there are three fundamental differences. First, our approach is used as a system for unobstructed Web access and not to perform any kind of attacks or other malicious activities. Second, we employ a body of volunteers who knowingly visit the sites that contain our crafted code so as to contribute to the unobstructed Web access of restricted users. And third, while puppetnets rely on a single piece of hidden code to perform their task, our approach is a more sophisticated architecture that combines multiple Web services to achieve its goals.

7 Future Work

The capabilities of Web browsers are increasing every day both in depth and breadth; JavaScript engines are becoming faster and more efficient and the new HTML 5 specification [6] will bring new features in the hands of Web programmers. For instance, one will be able to execute the hidden JavaScript code as a web worker, a form of thread that runs in the background of the browser. Or, one will be able to store Web application data in a contained local store in the user's hard disk. Finally, emerging technologies [1] will soon allow applications in Web browsers to communicate directly with each other in a peer-to-peer fashion. Our system could benefit so as to have volunteers communicate directly with clients, a capability that could facilitate end-to-end encryption. Moreover, faster JavaScript will mean better performance for our system and the ability to store data outside HTTP cookies will allow caching of content at the volunteers.

8 Discussion

We acknowledge that we are short of providing the bi-directional browsing experience of an open HTTP proxy or Tor. However, we consider our system highly resilient against blacklisting by an adversary. We argue that our proposal offers a "good enough" censorship resistant Web access mechanism were there is no other means available, i.e. solutions like Tor, open Proxies, VPNs have been blocked or are not available. Moreover, since our design is highly flexible in terms of its infrastructure, as it may employ a plethora and variety of Web services to operate, we feel that our system will last longer against an adversary than traditional anti-censorship systems.

9 Conclusion

In the traditional model, censorship-resistant Web access is based on a set of known proxy hosts which can be easily black-listed. In this paper, we propose a new paradigm where unobstructed access is based on a continuous supply of volunteering nodes and Web services, which cannot be censored using traditional filtering approaches.

We employ popular Web services, such as social networks and user-content hosting services, and utilize a swarm of lightweight nodes, operated by volunteers executing our JavaScript code inside their Web browser. The basic idea behind our approach is that clients place requests for URLs in Web services that are used as command and control (C&C) sites and, subsequently, a set of Web volunteers that act as nodes fetch the respective content and store it in an uncensored user-content hosting service for the restricted client to download.

As clients never communicate directly with the volunteers, there is no point for an adversary to try to discover and block the volunteers. At the same time, the design of our system is dynamic so that is may integrate many popular and diverse Web services as components. We argue that a censorship authority will be unwilling or unable to monitor and block all of these services without greatly impairing Internet access for users.

We've built a prototype using Twitter as the C&C mechanism and Pastebin as the primary Object Store. We present the implementation details along with ways to overcome technical challenges, such as bypassing the cross-origin policy. In terms of performance, the time to access pages through our system is comparable to that of Tor and other HTTP proxies.

Finally, we perform an extensive evaluation of the security of our approach against a plethora of censorship scenarios and attacks and show that in any case we are able to hold the properties of availability, unobservability and correctness so as to protect clients and volunteers of our system.

Bibliography

[1] Adobe Real Time Media Flow Protocol. http://labs.adobe.com/technologies/cirrus/.

[2] Arbor Networks Blog - Twitter-based Botnet Command Channel. http://asert.arbornetworks.com /2009/08/twitter-based-botnet- command-channel/.

[3] Carnal0wnage Blog - Detecting VMware with JavaScript. http://carnal0wnage.blogspot.com /2009/04/detecting-vmware- with-javascript-or-how.html.

[4] Future of the Internet Blog - Could Iran Shut Down Twitter? http://futureoftheinternet.org/could-iran-shut-down-twitter.

[5] GreatFirewall - Keep track of what's blocked in China. http://www.greatfirewall.biz/.

[6] HTML 5 differences from HTML 4. http://www.w3.org/TR/html5-diff/.

[7] ImperialViolet - Overclocking SSL. http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html.

[8] Internet Storm Center - Dynamic JavaScript Obfuscation. http://isc.sans.edu/ diary.html?storyid=3219.

[9] Internet Storm Center - Obfuscated JavaScript Analysis. http://isc.sans.edu/ diary.html?storyid=4246.

[10] OpenNet Initiative. http://opennet.net/.

[11] PCMagazine - With Firesheep All Your HTTP Sessions Belong To Us. http://blogs.pcmag.com/ securitywatch/2010/10/with_firesheep_all_your_http_s.php.

[12] So you still think the Internet is free. http://yuxiyou.net/open/.

[13] Symantec - Authentication Business. http://www.symantec.com/connect/blogs/authentication-business.

[14] TechCrunch - Twitter Seeing 90 Million Tweets Per Day, 25 Percent Contain Links. `http://techcrunch.com/2010/09/14/twitter-seeing-90-million-tweets-per-day/`.

[15] The Chromium Projects - Multiple Profiles. `http://www.chromium.org/user-experience/multi-profiles`.

[16] The Inquirer - South Korea mulls Web watch, June 2008. `http://www.theinquirer.net/inquirer/news/1042091/south-korea-mulls-web-watch`.

[17] Tor Bridges. `http://www.torproject.org/bridges`.

[18] Tor Metrics Portal: Network. `http://metrics.torproject.org/network.html`.

[19] Tor Project Blog - Tor partially blocked in China, Sept. 2009. `http://blog.torproject.org/blog/tor-partially-blocked-china`.

[20] W3C - HtmlDiff. `http://www.w3.org/wiki/HtmlDiff`.

[21] J. Boyan. Anonymizer: Protecting user privacy on the web. *Computer-Mediated Communication Magazine*, Sept. 1997.

[22] S. Burnett, N. Feamster, and S. Vempala. Chipping away at censorship firewalls with user-generated content. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

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

[24] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[25] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[26] V. Lam, S. Antonatos, P. Akritidis, and K. Anagnostakis. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[27] K. Levchenko, J. A. Morales, and D. McCoy. Proximax: Fighting censorship with an adaptive system for distribution of open proxies. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, 2010.

[28] N. F. Magdalena, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing web censorship and surveillance. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[29] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th USENIX Security symposium*, 2008.

[30] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, 2007.

[31] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1998.

[32] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: resilient anonymous routing. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, 2005.