Invasive Browser Sniffing and Countermeasures

Markus Jakobsson
Indiana University
Bloomington, Indiana USA
markus@indiana.edu

Sid Stamm
Indiana University
Bloomington, Indiana USA
sstamm@cs.indiana.edu

ABSTRACT

We describe the detrimental effects of browser cache/history sniffing in the context of phishing attacks, and detail an approach that neutralizes the threat by means of URL personalization; we report on an implementation performing such personalization *on the fly*, and analyze the costs of and security properties of our proposed solution.

Categories and Subject Descriptors

H.4.3 [**Information Systems**]: Communications Applications—*Information browsers*

General Terms

Security, Human Factors

Keywords

Browser cache, cascading style sheets, personalization, phishing, sniffing.

1. INTRODUCTION

It is commonly believed that phishing attacks increasingly will rely on contextual information about their victims, in order to increase their yield and lower the risk of detection. Browser caches are ripe with such contextual information, indicating whom a user is banking with; where he or she is doing business; and in general, what online services he or she relies on. As was shown in [2, 8, 4], such information can easily be "sniffed" by anybody whose site the victim visits. If victims are drawn to rogue sites by receiving emails with personalized URLs pointing to these sites, then phishers can create associations between email addresses and cache contents.

Phishers can make victims visit their sites by spoofing emails from users known by the victim, or within the same domain as the victim. Recent experiments by Jagatic et al. [3] indicate that over 80% of college students would visit a site appearing to be recommended by a friend of theirs. Over 70% of the subjects receiving emails appearing to come from a friend entered their login credentials at the site they were taken to. At the same time, it is worth noticing that around 15% of the subjects in a control group entered their credentials; subjects in the control group received an email appearing to come from an unknown person within the same domain as themselves. Even though the same statistics may not apply to the general population of computer users, it is clear that it is a reasonably successful technique of luring people to sites where their browsers silently will be interrogated and the contents of their caches sniffed.

Once a phisher has created an association between an email address and the contents of the browser cache/history, then this can be used to target the users in question with phishing emails that – by means of context – appear plausible to their respective recipients. For example, phishers can infer online banking relationships (as was done in [4]), and later send out emails appearing to come from the appropriate financial institutions. Similarly, phishers can detect possible online purchases and then send notifications stating that the payment did not go through, requesting that the recipient follow the included link to correct the credit card information and the billing address. The victims would be taken to a site looking just like the site they recently did perform a purchase at, and may have to start by entering their login information used with the real site. A wide variety of such tricks can be used to increase the yield of phishing attacks; all benefit from contextual information that can be extracted from the victim's browser.

There are several possible approaches that can be taken to address the above problem at the root – namely, at the information collection stage. First of all, users could be instructed to clear their browser cache and browser history frequently. However, many believe that any countermeasure that is based on (repeated) actions taken by users is doomed to fail. Moreover, the techniques used in [8, 4] will also detect bookmarks on some browsers (such as Safari version 1.2). These are not affected by the clearing of the history or the cache, and may be of equal or higher value to an attacker in comparison to the contents of the cache and history of a given user. A second approach would be to once and for all disable all caching and not keep any history data; this approach, however, is highly wasteful in that it eliminates the significant benefits associated with caching and history files. A third avenue to protect users against invasive browser sniffing is a client-side solution that *limits* (but does not eliminate) the use of the cache. This would be done based on a set of rules maintained by the user's browser or browser plug-in. Such an approach is taken in the concurrent work by Jackson et al. [1]. Finally, a fourth approach, and the one we propose herein, is a *server-side* solution that prevents cache contents from being verified by means of per-

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

sonalization. Our solution also allows such personalization to be performed by network proxies, such as Akamai.

It should be clear that client-side and server-side solutions not only address the problem from different angles, but also that these different approaches address slightly different versions of the problem. Namely, a client-side solution protects those users who have the appropriate protective software installed on their machines, while a server-side solution protect all users of a given service (but only against intrusions relating to their use of this service). The two are complimentary, in particular in that the server-side approach allows "blanket coverage" of large numbers of users that have not yet obtained client-side protection, while the client-side approach secures users in the face of potentially negligent service providers. Moreover, if a caching proxy is employed for a set of users within one organization, then this can be abused to reveal information about the behavioral patterns of users within the group *even if* these users were to employ client-side measures within their individual browsers; abuse of such information is stopped by a server-side solution, like the one we describe.

From a technical point of view, it is of interest to note that there are two very different ways in which one can hide the contents of a cache. According to a first approach, one makes it impossible to find references in the cache to a visited site, while according to a second approach, the cache is intentionally *polluted* with references to all sites of some class, thereby hiding the actual references to the visited sites among these. Our solution uses a combination of these two approaches: it makes it impossible to find references to all *internal* URLs (as well as all bookmarked URLs), while causing pollution of *entrance* URLs. Here, we use these terms to mean that an *entrance* URL corresponds to a URL a person would typically type to start accessing a site, while an *internal* URL is one that is accessed from an entrance URL by logging in, searching, or following links. For example, the URL `http://test-run.com` is an entrance URL since visitors are most likely to load that URL by typing it in or following a link from some other web site. The URL `http://test-run.com/logout.jsp`, however, is *internal*. This URL is far more interesting to a phisher than the entrance URL; knowing that a client C has been to this internal URL suggests that C logged *out* of the web site — and thus must have logged in. Our solution will make it infeasible for an attacker to guess the *internal* URLs while also providing some obscurity for the *entrance* URLs.

Outline. We begin by reviewing the related work (section 2), after which we specify our goals (section 3). We then detail our solution and argue why it satisfies our security requirements (section 4). Finally, we report on practical details of a test implementation (section 5).

Preliminary numbers support our claims that the solution results in only a minimal overhead on the server side, and an almost unnoticeable overhead on the client side. Here, the *former* overhead is associated with computing one one-way function *per client and session*, and with a repeated mapping of URLs in all pages served. The *latter* overhead stems from a small number of "unnecessary" cache misses that may occur at the beginning of a new session. We provide evidence that our test implementation would scale well to large systems without resulting in a bottleneck – whether it is used as a server-side or proxy-side solution.

2. RELATED WORK

Browser caches. Caches are commonly used in various settings, both on a given computer, and within an entire network. One particular use of caches is for browsers, to avoid the repeated downloading of material that has been recently accessed. Browser caches typically reside on the individual computers, but the closely related caching proxies are also common; these reside on a local network to take advantage not only of repeated *individual* requests for data, but also of repeated requests within the group of users. The very goal of caching data is to avoid having to repeatedly fetch it; this results in significant speedups of activity – in the case of browser caches and caching proxies, these speedups result in higher apparent download speeds.

Felten and Schneider [2] described a timing-based attack that made it possible to determine (with some statistically quantifiable certainty) whether a given user had visited a given site or not – simply by determining the retrieval times of consecutive URL calls in a segment of HTTP code.

Browser history. In addition to having caches, common browsers also maintain a *history* file; this allows browsers to visually indicate previous browsing activity to their users, and permits users to backtrack through a sequence of sites he or she visited.

Securiteam [8] showed a history attack analogous to the timing attack described by Felten and Schneider. The history attack uses Cascading Style Sheets (CSS) to infer whether there is evidence of a given user having visited a given site or not. This is done by utilizing the `:visited` pseudoclass to determine whether a given site has been visited or not, and later to communicate this information by invoking calls to URLs associated with the different sites being detected; the data corresponding to these URLs is hosted by a computer controlled by the attacker, thereby allowing the attacker to determine whether a given site was visited or not. We note that it is not the *domain* that is detected, but whether the user has been to a given page or not; this has to match the queried site *verbatim* in order for a hit to occur. The same attack was recently re-crafted by Jakobsson et al. to show the impact of this vulnerability on phishing attacks; a demo is maintained at [4]. This demo illustrates how simple the attack is to perform and sniffs visitors' history in order to display one of the visitor's recently visited U.S. banking web sites.

Context-Aware Phishing. Browser-recon attacks can be used as a component of context-aware phishing [5], also known as spear phishing [6]. These are phishing attacks where the attacker uses some knowledge learned about each individual victim in order to fool more of his victims. (For a more complete view of the context-aware phishing problem, see [7].) For example, a visitor's history could be sniffed to determine which bank web site that specific visitor has loaded. The phisher's site in turn can be rendered with that specific bank's logo [4].

A client-side solution. In work concurrent with ours, Jackson et al. [1] have developed a client-side solution addressing the above-described problem. This works by making the browser follow a set of rules of when to force cache and his-

tory misses – even if a hit could have been generated. This, in turn, hides the contents of the browser cache and history file to prying eyes. It does not, however, hide the contents of local cache proxies – unless these are also equipped with similar but in all likelihood more complex rule sets.

Our server-side solution. We approach such history attacks from the opposite side from Jackson et al. [1], and make URLs served by a service provider employing our solution infeasible to guess. Though similar techniques (where a unique random string is present in the URL) are employed by many web sites, usually this is used to prevent session replay and can be hard to weave these URLs through a complex web site. We provide a simple plug-in solution where a service provider can simply install a new server, or new software application on a server, and have a protected web site without further site development.

Implementation issues. We make use of the robots exclusion standard [9]. In this unofficial standard, parts of a server's file space is deemed as "off limits" to clients with specific User-Agent values. For example, a client may present a User-Agent value of:
"`Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)`"
indicating that the browser in use is IE 6.0. Additionally, the user-agent could be:
"`Mozilla/5.0 (compatible; googlebot/2.1)`"
indicating that Google's web crawler is viewing the site. Web servers can use the User-Agent value to hide portions of their site from web crawlers. We use the these techniques in a different manner. Namely, in conjunction with a whitelist approach, we use the robot exclusion standard to give certain privileges to pre-approved robot processes – the identities and privileges of these are part of a security policy of each individual site.

Our implementation may rely on either browser cookies or an HTTP header called *referer* (sic). Cookies are small amounts of data that a server can store on a client. These bits of data are sent from the server to client in HTTP headers – content that is not displayed. When a client requests a document from a server S, it sends along with the request any information stored in cookies by S. This transfer is automatic, and so using cookies has negligible overhead. The HTTP-Referer header is an optional piece of information sent to a server by a client's browser. The value (if any) indicates where the client obtained the address for the requested document. In essence it is the location of the link that the client clicked. If a client either types in a URL or uses a bookmark, no value for HTTP-Referer is sent.

3. GOALS

Informal goal specification. Our goals are to make the fullest possible use of both browser caches and browser histories, without allowing third parties to determine the contents of the cache/history. We refer to such actions as *sniffing*. More in detail, our goals are:

1. A service provider $\mathcal{SP}$ should be able to prevent any sniffing of any data related to any of their clients, for data obtained from $\mathcal{SP}$, or referenced in documents served by $\mathcal{SP}$. This should hold even if the distribution of data is performed using network proxies. Here, we only consider sniffing of browsers of users not controlled by the adversary, as establishing control over a machine is a much more invasive attack, requiring a stronger effort.
2. The above requirement should hold even if caching proxies are used. Moreover, the requirement must hold even if the adversary controls one or more user machines within a group of users sharing a caching proxy.
3. Search engines must retain the ability to find data served by $\mathcal{SP}$ in the face of the augmentations performed to avoid sniffing; the search engines should not have to be aware of whether a given $\mathcal{SP}$ deploys our proposed solution or not, nor should they have to be augmented to continue to function as before.

Intuition. We achieve our goals using two techniques. First and foremost, we use a customization technique for URLs, in which each URL is "extended" using either a temporary or long-term pseudonym (discussed in Section 4.1). This prevents a third party from being able to interrogate the browser cache/history of a user having received customized data, given that all known techniques to do so require knowledge of the exact file name being queried for. A second technique is what we refer to as *cache pollution*; this allows a site to prevent meaningful information from being inferred from a cache by having spurious data entered.

One particularly aggressive attack (depicted in Figure 1) that we need to be concerned with is one in which the attacker obtains a valid pseudonym from the server, and then tricks a victim to use this pseudonym (e.g., by posing as the service provider in question.) Thus, the attacker would potentially *know* the pseudonym extension of URLs for his victim, and would therefore also be able to query the browser of the victim for what it has downloaded.

Hiding vs. obfuscating. As mentioned before, we will *hide* references to internal URLs and bookmarked URLs, and *obfuscate* references entrance URLs. The hiding of references will be done using a method that customizes URLs using pseudonyms that cannot be anticipated by a third party, while the obfuscation is done by polluting: adding references to all other entrance URLs in a given set of URLs. This set is referred to as the *anonymity set*.

Formal goal specification. Let S be a server, or a proxy acting on behalf of a server; here, S responds to requests according to some policy π_S. Further, let C be a client; here, C is associated with one user account, and one browser. The browser, in turn, is associated with a state σ_C, where the state consists of different categories (such as cache and history), and for each category of the state, a set of URLs or other identifiers is stored. Furthermore, we let P be a caching proxy associated with a set of clients $\mathcal{C}$, $C \in \mathcal{C}$, and σ_P be the state of P; we let that be organized[1] into segments

[1]This organization is assumed simply for denotational simplicity, and does not have to be performed in an actual implementation.

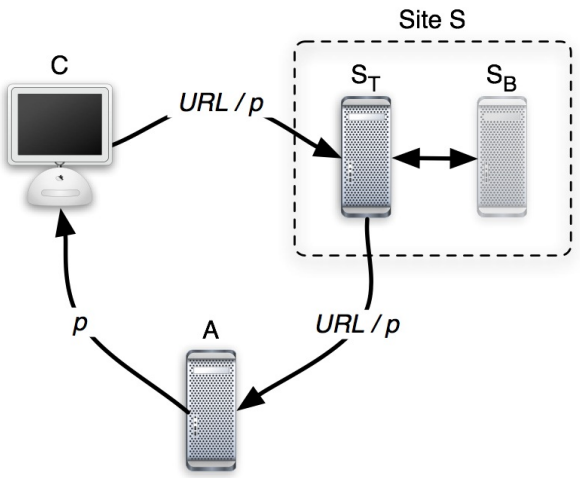

Figure 1: An attack in which A obtains a valid pseudonym p from the translator S_T of a site S with back-end S_B, and coerces a client C to attempt to use p for his next session. This is performed with the goal of being able to query C's history/cache files for what pages within the corresponding domain that C visited. Our solution disables such an attack.

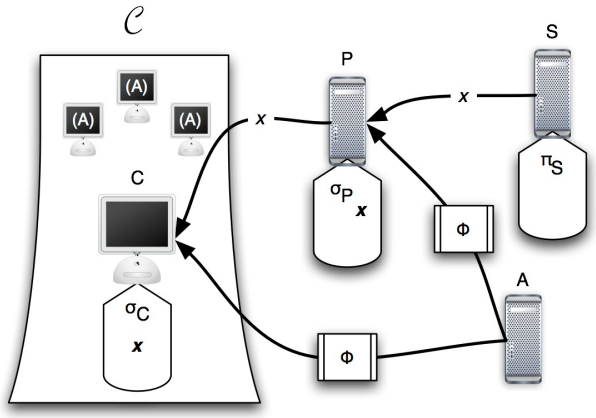

Figure 2: Formalization of a server S, caching proxy P, client C, attacker A, and attack message Φ (that is sent either through the proxy or directly to C). A controls many members of $\mathcal{C}$, allowing it – in a worst case scenario – to generate and coordinate the requests from these members. This allows the attacker to determine what components of the caching proxy P are likely to be associated with C.

σ_{Pi}, each one of which corresponds to a client $i \in \mathcal{C}$. These segments, in turn, are each structured in the same manner as σ_C is organized. When C retrieves data corresponding to some URL x from a document served by S, then x is entered in both σ_C and σ_{PC} (see Figure 2; contents of σ_C and σ_{PC} are deleted according to some set of rules that are not of importance herein. We let $HIT_C(x)$ be a predicate that is true if and only if σ_C or σ_{PC} contains x. We say that S and x are *associated* if documents served by S contain references to x; we note that this allows x to be maintained by a server other than S. Further, we say that an entrance S is n-indicated by x if there are at least n independent domains with entrances associated with x. (Thus, n corresponds to the size of the anonymity set of S.)

We let A be an adversary controlling any member of $\mathcal{C}$ but C, and interacting with both S and C some polynomial number of times in the length of a security parameter k. When interacting with S, A may post arbitrary requests x and observe the responses; when interacting with C, it may send any document X to C, forcing C to attempt to resolve this by performing the associated queries. Here, X may contain any polynomial number of URLs x_j of A's choice. A first goal of A is to output a pair (S, x) such that $HIT_C(x)$ is true, and where x and S are associated. A second goal of A is to output a pair (S, x) such that $HIT_C(x)$ is true, and where S is n-indicated by x.

We say that π_S is *perfectly* privacy-preserving if A will not attain the first goal but with a negligible probability in the length of the security parameter k; the probability is taken over the random coin tosses made by A, S, P and C. Similarly, we say that π_S is n privacy-preserving if A will not attain the second goal but with a negligible probability.

Furthermore, we let E be a search engine; this is allowed to interact with C some polynomial number of times in k. For each interaction, E may post an arbitrary request x and observe the response. The strategy used by E is independent of π_S, i.e., E is oblivious of the policy used by S to respond to requests. Thereafter, E receives a query q from C, and has to output a response. We say that π_S is *searchable* if and only if E can generate a valid response x to the query, where x is considered valid if and only if it can be successfully resolved by S.

In the next section, we describe a solution that corresponds to a policy π_S that is searchable, and which is perfectly privacy-preserving with respect to internal URLs and bookmarked URLs, and n-privacy-preserving with respect to entrance URLs, for a value n corresponding to the maximum anonymity set of the service offered.

4. A SERVER-SIDE SOLUTION

At the heart of our solution is a filter associated with a server whose resources and users are to be protected. Similar to how middleware is used to filter calls between application layer and lower-level layers, our proposed filter modifies communication between users/browsers and servers – whether the servers are the actual originators of information, or simply act on behalf of these, as is the case for network proxies.

When interacting with a client (in the form of a web browser), the filter customizes the names of all files (and the corresponding links) in a manner that is unique for the session, and which cannot be anticipated by a third party. Thus, such a third party is unable to verify the contents of the cache/history of a chosen victim; this can only be done by somebody with knowledge of the name of the visited pages.

4.1 Pseudonyms

Establishing a pseudonym. When a client first visits a site protected by our translator, he accesses an *entrance* such as the index page. The translator catches this request's absence of personalization, and thus generates a pseudonym extension for the client.

Pseudonyms and temporary pseudonyms are selected from a sufficiently large space, e.g., of 64-128 bits length. Temporary pseudonyms includes redundancy, allowing verification of validity by parties who know the appropriate secret key; pseudonyms do not need such redundancy, but can be verified to be valid using techniques to be detailed below.

Pseudonyms are generated pseudorandomly each time any visitor starts browsing at a web site. Once a pseudonym has been established, the requested page is sent to the client using the translation methods described next.

Using a pseudonym. All the links, form URLs, and image references on translated pages (those sent to the client through the translator) are modified in two ways. First, any occurrence of the server's domain is changed to that of the translator[2]. This way requests will go to the translator, instead of the server. Second, a querystring-style argument is added to the URLs served by the translator (for the server). This makes all the links on a page look different depending on who and when the site is visited.

Pseudonym validity check. If an attacker A were able to first obtain valid pseudonyms from a site S, and later were able to convince a victim client C to use these same pseudonyms with S, then this would allow A to successfully determine what pages of S that C requested. To avoid such an attack, we need to authenticate pseudonyms, which can be done as follows:

1. Cookies: A cookie (which is accessible to only the client and the protected server) can be established on the client C when a pseudonym is first established for C. The cookie value could include the value of the pseudonym. Later, if the pseudonym used in a requested URL is found to match the cookie of the corresponding client C, then the pseudonym is considered valid. Traditional cookies as well as cache cookies (see, e.g., [2, 8]) may be used for this purpose.

2. HTTP-Referer: The HTTP-Referer (sic) header in a client's request contains the location of a referring page: in essence, this is the page on which a followed link was housed. If the referrer is a URL on the site associated with the server S, then the pseudonym is considered valid.

3. Message Authentication Codes: Temporary pseudonyms may be authenticated using message authentication codes, where the key in question is shared by the referring site and the site S. Such pseudonyms may consist of a counter and the MAC on the counter, and would be found valid if and only if the MAC on the counter is valid.

[2]The requested domain can be that which is normally associated with the service, while the translated domain is an internal address. It would be transparent to users whether the translator is part of the server or not.

A site may use more than one type of pseudonym authentication, e.g., to avoid replacing pseudonyms for users who have disabled cookies or who do not provide appropriate HTTP-referrers (but not both.) It is a policy matter to determine what to do if a pseudonym or temporary pseudonym cannot be established to be valid. One possible approach is to refuse the connection, and another is to replace the invalid pseudonym with a freshly generated pseudonym. (We note that the unnecessary replacement of pseudonyms does not constitute a security vulnerability, but merely subverts the usefulness of the client's cache.)

HTTP-Referer is an optional header field. Most modern browsers provide it (IE, Mozilla, Firefox, Safari) but it will not necessarily be present in case of a bookmark or manually typed in link. This means that the referer will be within server S's domain if the link that was clicked appeared on an one of the pages served by S. This lets us determine whether we can skip the pseudonym generation phase. Thus, one approach to determine the validity of a pseudonym may be as follows:

- S looks for an HTTP referer header. If the referer is from S's domain, the associated pseudonym is considered valid.
- Otherwise, S checks for the proper pseudonym cookie. If it's there and the cookie's value matches the pseudonym given, then the associated pseudonym is considered valid.
- Otherwise, disallow access with the given pseudonym to prevent the related URL from entering C's cache or history.

Robot policies. The same policies do not necessarily apply to robots and to clients representing human users. In particular, when interacting with a *robot* [9] (or *agent*), then one may do not want to customize names of files and links, or customize them using pseudonyms that will be replaced when they are used.

Namely, one could – using a whitelist approach – allow certain types of robot processes to obtain data that is not pseudonymized; an example of a process with such permission would be a crawler for a search engine. As an alternative, any search engine may be served data that is customized using *temporary pseudonyms* – these will be replaced with a fresh pseudonym each time they are accessed. All other processes are served URLs with pseudo-randomly chosen (and then static) pseudonym, where the exact choice of pseudonym is not possible to anticipate for a third party.

More in particular, if there is a privacy agreement between the server S and the search engine E, then S may allow E to index its site in a non-customized state; upon generating responses to queries, E would customize the corresponding URLs using pseudo-randomly selected pseudonyms. These can be selected in a manner that allows S to detect that they were externally generated, allowing S to immediately replace them with freshly generated pseudonyms. In the absence of such arrangements, the indexed site may serve the search engine URLs with temporary pseudonyms (generated and authenticated by itself) instead of non-customized URLs or URLs with (non-temporary) pseudonyms. Note that in this case we have that all users receiving a URL with a temporary pseudonym from the search engine would

receive the *same* pseudonym. This corresponds to a degradation of privacy in comparison to the situation in which there is an arrangement between the search engine and the indexed site, but an improvement compared to a situation in which non-customized URLs are served by the search engine. We note that in either case, we have that the search engine does is unable to determine what internal pages on an indexed site a referred user has visited.

The case in which a *client-side* robot is accessing data corresponds to another interesting situation. Such a robot will *not* alter the browser history of the client (assuming it is not part of the browser), but *will* impact the client cache. Thus, such robots should be not be excepted from customization, and should be treated in the same way as search engines without privacy arrangements, as described above.

In the implementation section, we describe these (server-side) policies in greater detail. We also note that these issues are orthogonal to the issue of how robots are handled on a given site, *were our security enhancement not to be deployed.* In other words, at some sites, where robots are not permitted whatsoever, the issue of when to perform personalization (and when not to) becomes moot.

Pollution policy. A client C can arrive at a web site through four means: typing in the URL, following a bookmark, following a link from a search engine, and by following a link from an external site. A bookmark may contain a pseudonym established by S, and so already the URL entered into the C's history (and cache) will be privacy-preserving. When a server's S obtains a request for an entrance URL not containing a valid pseudonym, S must pollute the cache of C in a way such that analysis of C's state will not make it clear which site was the intended target.

When C's cache is polluted, the entries must be either chosen at random or be a list sites that all provide the same pollutants. Say when Alice accesses S, her cache is polluted with sites X, Y, and Z. If these are the chosen pollutants each time, the presence of these three sites in Alice's cache is enough to determine that she has visited S. However, if all four sites S, X, Y, and Z pollute with the same list of sites, no such determination can be made.

If S cannot guarantee that all of the sites in its pollutants list will provide the same list, it must randomize which pollutants it provides. Taken from a large list of valid sites, a random set of pollutants essentially acts as a bulky pseudonym that preserves the privacy of C – which of these randomly provided sites was actually targeted cannot be determined by an attacker.

4.2 Translation

Off-site references. The translator, in effect, begins acting as a proxy for the actual web server – but the web pages could contain references to off-site (external) images, such as advertisements. An attacker could still learn that a victim has been to a web site based on the external images or other resources that it loads, or even the URLs that are referenced by the web site. Because of this, the translator should also act as an intermediary to forward external references as well or forward the client to these sites through a standard redirection URL; many web sites such as Google's GMail employ a technique like this to anonymize the referring page.

It is important to note that the translator *should not ever* translate pages off-site pages; this could cause the translator software to start acting as an open proxy. The external URLs that it is allowed to serve should be a small number to prevent this.

Redirection may not be necessary, depending on the trust relationships between the external sites and the protected server, although for optimal privacy either redirection should be implemented or off-site images and URLs should be removed from *internal* pages. Assuming that redirection is implemented, the translator has to modify off-site URLs to redirect through itself, except in cases in which two domains collaborate and agree to pseudonyms set by the other, in which case we may consider them the same domain, for the purposes considered herein. This allows the opportunity to put a pseudonym in URLs that point to off-site data. This is also more work for the translator and could lead to serving unnecessary pages. Because of this, it is up to the administrator of the translator (and probably the owner of the server) to set a policy of what should be directed through the translator S_T. We refer to this as an *off-site redirection policy.* It is worth noting that many sites with a potential interest in our proposed measure (such as financial institutions) may never access external pages unless these belong to partners; such sites would therefore not require off-site redirection policies.

Similarly, a policy must be set to determine what types of files get translated by S_T. The scanned types should be set by an administrator and is called the *data replacement policy.*

Example. A client Alice navigates to a requested domain `http://test-run.com` (this site is what we previously described as S) that is protected by a translator S_T. In this case, the translator is really what is located at that address, and the server is hidden to the public at an internal address (10.0.0.1 or S_B) that only the translator can see. The S_T recognizes her User-Agent (provided in an HTTP header) as not being a robot, and so proceeds to preserve her privacy. A pseudonym is calculated for her (say, 38fa029f234fadc3) and then the S_T queries the actual server for the page. The translator receives a page described in Figure 4.2.

The translator notices the pseudonym on the end of the request, so it removes it, verifies that it is valid (e.g., using cookies or HTTP Referer), and then forwards the request to the server. When a response is given by the server, the translator re-translates the page (using the steps mentioned above) using the same pseudonym, which is obtained from the request.

4.3 Translation Policies

Offsite redirection policy. Links to external sites are classified based on the sensitivity of the site. Which sites are redirected through the translator S_T should be carefully considered. Links to site a from the server's site should be redirected through S_T only if an attacker can deduce something about the relationship between C and S based on C visiting site a. This leads to a classification of external sites into two categories: *safe* and *unsafe*.

Distinguishing safe from unsafe sites can be difficult depending on the content and structure of the server's web site. Redirecting all URLs that are referenced from the domain of

```
<a href='http://www.google.com/'>Go to google</a>
<a href='http://10.0.0.1/login.jsp'>Log in</a>
<img src='/images/welcome.gif'>
```

The translator replaces any occurrences of the S_B's address with its own.

```
<a href='http://www.google.com/'>Go to google</a>
<a href='http://test-run.com/login.jsp'>Log in</a>
<img src='/images/welcome.gif'>
```

Then, based on S_T's off-site redirection policy, it changes any off-site (external) URLs to redirect through itself:

```
<a href='http://test-run.com/redir?www.google.com'> Go to google</a>
<a href='http://test-run.com/login.jsp'>Log in</a>
<img src='/images/welcome.gif'>
```

Next, it updates all on-site references to use the pseudonym. This makes all the URLs unique:

```
<a href='http://test-run.com/redir?www.google.com?38fa029f234fadc3'> Go to google</a>
<a href='http://test-run.com/login.jsp?38fa029f234fadc3'>Log in</a>
<img src='/images/welcome.gif?38fa029f234fadc3'>
```

All these steps are of course performed in one round of processing, and are only separated herein for reasons of legibility. If Alice clicks the second link on the page (Log in) the following request is sent to the translator:

```
GET /login.jsp?38fa029f234fadc3
```

Figure 3: A sample translation of some URLs

S will ensure good privacy, but this places a larger burden on the translator. Servers that do not reference offsite URLs from "sensitive" portions of their site could minimize redirections while those that do should rely on the translator to privatize the clients' URLs.

Data replacement policy. URLs are present in more than just web pages: CSS style sheets, JavaScript files, and Java applets are a few. Although each of these files has the potential to affect a client's browser history, not all of them actually will. For example, an interactive plug-in based media file such as Macromedia Flash may incorporate links that direct users to other sites; a JPEG image, however most likely will not. These different types of data could be classified in the same manner: *safe* or *unsafe*. Then when the translator forwards data to the client, it will only search for and replace URLs in those files defined by the policy.

Since the types of data served by the back-end server S_B are controlled by its administrators (who are in charge of S_T as well), the data types that are translated can easily be set. The people in charge of S's content can ensure that sensitive URLs are only placed in certain types of files (such as HTML and CSS) – then the translator only has to process those files.

Client robot distinction. We note that the case in which a client-side robot (running on a client's computer) is accessing data is a special case. Such a robot *will not* alter the browser history of the client (assuming it is not part of the browser), but *will* impact the client cache. Thus, such robots should not be excepted from personalization. In the implementation section, we describe this (server-side) policy in greater detail.

4.4 Special Cases

Akamai. It could prove more difficult to implement a translator for web sites that use a distributed content delivery system such as Akamai. There are two methods that could be used to adapt the translation technique: First, the service provider could offer the service to all customers – thus essentially building the option for the translation into their system. Second, the translator could be built into the web site being served. This technique does not require that the translation be separate from the content distribution – in fact, some web sites implement pseudonym-like behaviors in URLs for their session tracking needs.

Shared/transfer pseudonyms. Following links without added pseudonyms causes the translator to pollute the cache. A better alternative may be that of shared pseudonyms (between sites with a trust relationship) or transfer pseudonyms (between collaborating sites without a trust relationship.) Namely, administrators of two translated web sites A and B could agree to pass clients back and forth using pseudonyms. This would remove the need for A to redirect links to B through A's translator, and likewise for B. If these pseudonyms are adopted at the receiving site, we refer to them as shared pseudonyms, while if they are replaced upon arrival, we refer to them as transfer pseudonyms. We note that the latter type of pseudonym would be chosen for the sole purpose of inter-domain transfers – the pseudonyms used within the referring site would not be used for transfers, as this would expose these pseudonym values to the site that is referred to.

Cache pollution reciprocity. A large group of site administrators could agree to pollute, with the same set of un-targeted URLs, caches of people who view their respective sites without a pseudonym. This removes the need to generate a random list of URLs to provide as pollutants and could speed up the pollution method. Additionally, such agreements could prevent possibly unsolicited traffic to each of these group-members' sites.

4.5 Security Argument

Herein, we argue why our proposed solution satisfies the previously stated security requirements. This analysis is rather straight-forward, and only involves a few cases.

Perfect privacy of internal pages. Our solution does not expose pseudonyms associated with a given user/browser to third parties, except in the situation where temporary pseudonyms are used (this only exposes the fact that the user visited that very page) and where shared pseudonyms are used (in which case the referring site is trusted.) Further, a site replaces any pseudonyms not generated by itself or trusted collaborators. Thus, assuming no intentional disclosure of URLs by the user, and given the pseudo-random selection of pseudonyms, we have that the pseudonyms associated with a given user/browser can not be inferred by a third party. Similarly, it is not possible for a third party to cause a victim to use a pseudonym given to the server by the attacker, as this would cause the pseudonym to become invalid (which will be detected.) It follows that the solution offers perfect privacy of internal pages.

n-privacy of entrance pages. Assuming pollution of n entrance points from a set $\mathcal{X}$ by any member of a set of domains corresponding to $\mathcal{X}$, we have that access of one of these entrance points cannot be distinguished from the access of another – from cache/history data alone – by a third party.

Searchability. We note that any search engine that is excepted from the customization of indexed pages (by means of techniques used in the robots exclusion standard) will be oblivious of the translation that is otherwise imposed on accesses, unless in agreement to apply temporary pseudonyms. Similarly, a search engine that is served already customized data will be able to remain oblivious of this, given that users will be given the same URLs, which will then be re-customized.

It is worth noting that while clients can easily manipulate the pseudonyms, there is no benefit associated with doing this, and what is more, it may have detrimental effects on the security of the client. Thus, we do not need to worry about such modifications since they are irrational.

5. IMPLEMENTATION DETAILS

We implemented a rough prototype translator to estimate ease of use as well as determine approximate efficiency and accuracy. Our translator was written as a Java application that sat between a client C and protected site S. The translator performed user-agent detection (for identifying robots); pseudonym generation and assignment; translation (as described in Section 4.2); and redirection of external (off-site) URLs. We placed the translator on a separate machine from S in order to get an idea of the worst-case timing and interaction requirements, although they were on the same local network. The remote client was set up on the Internet outside that local network.

In an ideal situation, a web site could be augmented with a translator easily: the software serving the site is changed to serve data on the computer's loopback interface (`127.0.0.1`) instead of through the external network interface. Second, the translator is installed and listens on the external network interface and forwards to the server on the loopback interface. It seems to the outside world that nothing has changed: the translator now listens closest to the clients at the same address where the server listened before. Additionally, extensions to a web server may make implementing a translator very easy.[3]

5.1 Pseudonyms and Translation

Pseudonyms were calculated in our prototype that use the `java.security.SecureRandom` pseudo-random-number generator to create a 64-bit random string in hexadecimal. Pseudonyms could easily be generated to any length using this method, but 64-bit was deemed adequate for our test.

A client sent requests to our prototype and the URL was scanned for an instance of the pseudonym. If the pseudonym was not present, it was generated for the client as described and then stored only until the response from the server was translated and sent back to the client.

Most of the parsing was done in the header of the HTTP requests and responses. We implemented a simple data replacement policy for our prototype: any value for User-Agent that was not "robot" or "wget" was assumed to be a human client. This allowed us to easily write a script using the command-line wget tool in order to pretend to be a robot. Any content would simply be served in basic proxy mode if the User-Agent was identified as one of these two.

Additionally, if the content type was not `text/html`, then the associated data in the data stream was simply forwarded back and forth between client and server in a basic proxy fashion. HTML data was intercepted and parsed to replace URLs in common context locations:

- Links (`<a href='`$\langle URL \rangle$`'>...</a>`)
- Media (`<`$\langle tag \rangle$ `src='`$\langle URL \rangle$`'>`)
- Forms (`<form action='`$\langle URL \rangle$`'>`)

More contexts could easily be added, as the prototype used Java regular expressions for search and replace.[4] The process of finding and replacing URLs is not very interesting because the owner of the translator most likely owns the server too and can customize the server's content to be "translator-friendly" – easily parsed by a translator.

[3]The Apache web server can be extended with `mod_rewrite` to rewrite requested URLs on the fly — with very little overhead. Using this in combination with another custom module (that would translate web pages) could provide a full-featured translator "proxy" without requiring a second server or web service program.

[4]Our prototype did not contain any optimizations because it was a simple proof-of-concept model and we wanted to calculate worst-case timings.

Redirection policy. The prototype also implemented a very conservative redirection policy: for all pages p served by the web site hosted by the back-end server S_B, any external URLs on p were replaced with a redirection for p through S_T. Any pages q *not* served by S_B were not translated at all and simply forwarded; the URLs on q were left alone.

Timing. The prototype translator did not provide significant overhead when translating documents. Since only HTML documents were translated, the bulk of the content (images) were simply forwarded. Because of this, we did not include in our results the time taken to transfer any file other than HTML. Essentially our test web site served only HTML pages and no other content. Because of this, all content passing through the translator had to be translated. This situation represents the *absolute worst case scenario* for the translator. As a result, our data may be a conservative representation of the speed of a translator.

Set up	Avg.	StdDev.	Min	Max
No Translator	0.1882s	0.0478s	0.1171s	1.243s
Basic Proxy	0.2529s	0.0971s	0.1609s	1.991s
Full Translation	0.2669s	0.0885s	0.1833s	1.975s

Table 1: Seconds delay in prototype translator

We measured the amount of time it took to completely send the client's request and receive the entire response. This was measured for eight differently sized HTML documents 1000 times each. We set up the client to only load single HTML pages as a conservative estimate – in reality fewer pages will be translated since many requests for images will be sent through the translator. Because of this we can conclude that the actual impact of the translator on a robust web-site will be less significant than our findings.

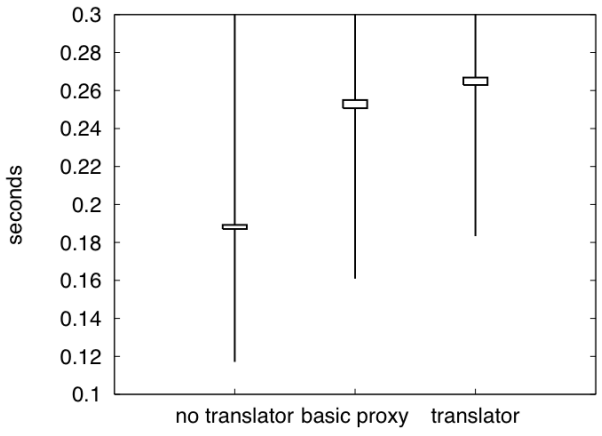

Figure 4: Confidence intervals for three tests — based on our sample, the mean of any future tests will appear within the confidence intervals (boxes) shown above with a 95% probability. The lines show the range of our data (truncated at the top to emphasize the confidence intervals).

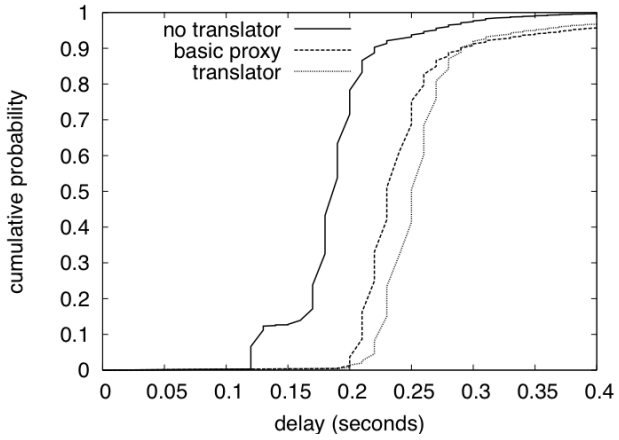

Figure 5: Cumulative distribution of our data. The vast majority of the results from each of the three test cases appears in a very short range of times, indicating cohesive results. Additionally, the delay for translation is only about 90ms more than for standard web traffic (with no translator).

Our data (Figures 4 and 5) shows that the translation of pages does not create noticeable overhead on top of what it takes for the translator to act as a basic proxy. Moreover, acting as a basic proxy creates so little overhead that delays in transmission via the Internet completely shadow any performance hit caused by our translator (Table 1)[5]. We conclude that the use of a translator in the fashion we describe will not cause a major performance hit on a web site.

5.2 General Considerations

Forwarding user-agent. It is necessary that the User-Agent attribute of HTTP requests be forwarded from the translator to the server. This way the server is aware what type of end client is asking for content. Some of the server's pages may rely on this: perhaps serving different content to different browsers or platforms. If the User-Agent were not forwarded, the server would always see the agent of the translator and would not be able to tell anything about the end clients – so it is forwarded to maintain maximum flexibility.

Cookies to be translated. When a client sends cookies, it *only* sends the cookies to the server that set them. This means if the requested domain is not the same as the hidden domain (that is, the translator is running on a machine other than the protected server) then the translator will have to alter the domain of the cookies as they travel back and forth between the client and server (Figure 6). This is clearly unnecessary if the translator is simply another process running in the same domain as the privacy-preserving server – the domain does not have to change.

[5]A small quantity of outliers with much longer delays (more than four seconds) was removed from our data since it was most likely due to temporary delays in the Internet infrastructure.

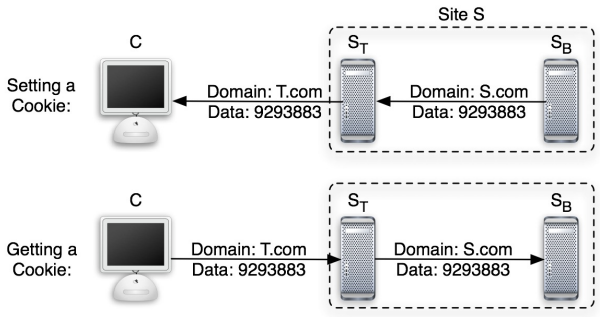

Figure 6: The translation of cookies when transferred between C and S_B through a translator S_T.

Cookies that are set or retrieved by external sites (not the translated server) will not be translated by the translator. This is because the translator in effect *only represents its server* and not any external sites.

Translation optimization. Since the administrator of the server is most likely in control of the translator too, she has the opportunity to speed up the translation of static content. When a static HTML page is served, the pseudonym will always be placed in the same locations no matter what the value of the pseudonym. This means that the locations where pseudonyms should be inserted can be stored along side of the content – then the translator can easily plop in pseudonyms without having to search through and parse the data files.

Acknowledgments

Many thanks to Tom Jagatic for stimulating discussions and Mike McLeish for assistance with timing measurements.

6. REFERENCES

[1] C. Jackson, A. Bortz, D. Boneh, J. C. Mitchell, "Web Privacy Attacks on a Unified Same-Origin Browser," in submission.

[2] E. W. Felten and M. A. Schneider, "Timing Attacks on Web Privacy," In Jajodia, S. and Samarati, P., editors, 7th ACM Conference in Computer and Communication Security 2000, pp. 25–32.

[3] T. Jagatic, N. Johnson, M. Jakobsson, F. Menczer: Social Phishing. 2006

[4] M. Jakobsson, T. Jagatic, S. Stamm, "Phishing for Clues," `www.browser-recon.info`

[5] M. Jakobsson "Modeling and Preventing Phishing Attacks." Phishing Panel in Financial Cryptography '05. 2005.

[6] B. Grow, "Spear-Phishers are Sneaking in." BusinessWeek, July 11 2005. No. 3942, P. 13

[7] M. Jakobsson and S. Myers, "Phishing and Counter-Measures: Understanding the Increasing Problem of Electronic Identity Theft." Wiley-Interscience (July 7, 2006), ISBN 0-4717-8245-9.

[8] `www.securiteam.com/securityreviews/5GP020A6LG.html`

[9] `www.robotstxt.org/`