Patching Browsers and DNS Clients to foil Timing Attacks

Prashant Dewan
Computer Science and Engineering
Arizona State University
Tempe AZ, U.S.A.

Partha Dasgupta
Computer Science and Engineering
Arizona State University
Tempe AZ, U.S.A.

Abstract

Timing attacks exploit a loophole that allows any website to deduce confidential information fom a web user visiting the website by peeping into the user's Internet cache or DNS resolver cache. Any unscrupulous attacker can also insert 'cache cookies' into the user's Internet cache.

The solutions, which have been proposed so far, are disabling the caches, the java script, java and domain tagging. Disabling the caches will reduce the surfing speed of the user and will exerting an extra load on the network. It is very unrealistic to disable java and/or javascript since they are widely used. 'Domain Tagging' will modify the browser architecture, which can only be done if the source code for the browser is available and someone is ready to bear the overheads associated with the modification needed.

We present a technique and its implementation to foil timing attacks. This technique allows a restricted access to the Internet cache and the DNS resolver cache to the browser. It uses API interception and Browser Helper Objects to alter the default browser behavior. The proposed technique is implemented without peeping into the source code of the browser. As a result, this technique blocks any suspicious requests to the cache. This technique foils all forms of timing attacks.

Keywords: Internet Security, API Interception, Operating Systems, Timing Attacks, Web Privacy.

1. Introduction

The number of people using the web is doubling every year and by the year 2003, 476 million people are expected to be online [8][5]. Due to an extensive use of the Internet and the stakes involved, the security of a user on the net has become a critical issue. Users want to be protected from unscrupulous attackers who are trying to invade their privacy and possibly harm them.

This paper presents a technique and its implementation to prevent a class of attacks called 'Timing Attacks'. *Felten and Schneider,* have described this class of attacks in their paper '*Timing Attacks on Web Privacy*' [3]. These attacks allow an intruder to gather a browser user's web surfing history, without his knowledge or consent, and compromise his privacy on the World Wide Web (WWW). This attack exploits the design of the browser cache and the DNS caches. Fundamentally, the cache 'hits' are faster than the cache 'misses'. Hence, an attacker can find out if the user has visited any website (or a set of sites) by measuring the time consumed in accessing a particular web object [3]. The previously suggested solutions (explained in [3] and summarized in Section 4) either need a major modification in the design of the web browser or restrict a user from accessing very useful features of most of the web browsers.

As of July 2001, 88% of the online users browse the web with Internet Explorer (IE) and 9% use the Netscape Navigator (NN) [9]. Most of the browsers cache the content rendered by them, in their Internet Cache in order to make the subsequent access of the same content faster and reduce the network traffic. The high level of dependence on the above-mentioned browsers makes a large section of the online population susceptible to Timing Attacks.

The technique proposed in this paper foils all forms of timing attacks without modifying the browser architecture or forcing the browser user to switch off caching, java or java script. It is implemented without looking into the source code of the IE. IE is a 'closed source' browser and even Netscape has not released the source code of its latest browser (Version 6.2). Any user who is worried about his privacy on the WWW is dependent on the browser manufacturer to release a patch to fix the loophole. The browser manufacturer may or may not plug the loophole, depending upon the cost involved and the percentage of users who are aware of this vulnerability. The technique proposed in this paper obviates any such dependencies.

2. Motivation and Example

Internet users routinely report that protection of their privacy on the web is one of their greatest concerns [11].

The browsing history of the user can expose his family, health, and psychological and financial information. The information collected by this attack can be used against the user.

If an insurance company finds out that an insurance applicant has been visiting the sites of different hospitals in the town, it can reject his application on the grounds of suspicion. An advertising company can find out the preferences' of users by continuously monitoring their browsing history, and can modify the advertisements, which the user sees when he browses across websites, in order to influence the user. An attacker can also find the browsing history of the employees of a company or any other organization and hence compromise the sensitive issues of that company or organization. We illustrate this process in Section 3.

In addition, an attacker can put a cache cookie into the user's cache. Then, other sites can use this cookie to ascertain if the user has visited a particular site and deduce user behavior.

For example consider two web sites. One owned by the cardiac department of a hospital and the second owned by an insurance company. The first site is having the URL *cardiac.hospital.com* and the second is second is having the URL *insurance.health.com*. The first site has its main logo *cardiaclogo.gif* embedded in the website. Mr. Brown has applied for a life insurance and has not declared any disease(s). When Mr. Brown visits *insurance.health.com* an applet embedded in the site makes a HTTP request for *cardiaclogo.gif*. As the file lies in the Internet Cache of the browser, its retrieval is quicker (hit) and hence *insurance.health.com* deduces that the file has been retrieved from the Internet Cache on the client and hence, the user has been visiting *cardiac.hospital.com* lately.

Felten and *Schneider* have suggested *'Domain tagging'* in [3], which is explained in Section 4. It necessitates the modification of the source code of the browser. This modification can be an enormous task considering the fact that it would need modification of at least one module of the browser and testing of possible ramifications in other modules of the browser.

It is an accepted fact that fixing one bug can introduce more bugs due to human error or unobvious dependencies between parts of a big application. As Internet browsers have magnanimous source codes the probability of this happening is even higher if the browser is modified as compared to modification in a DLL. In addition detection and fixing of bugs in components is much easier than tracing and fixing the bugs in big applications.

The API used by IE to access the Internet Cache is in *Wininet.dll*, and for foiling this attack; it will be necessary to modify the DLL. This modification will mandate testing of all commercial and user specific applications, which use that DLL to access the Internet Cache. The cost of testing and possible subsequent modifications can be too high to be borne for plugging this loophole.

The technique described in this paper modifies the browser behavior non-intrusively, and it does not have any impact on the other modules of the browser. In addition only the calls from the browser to *Wininet.dll are* intercepted. As a result, the behavior of the DLL for all other applications does not change.

In some scenarios, the other applications might also need the modified behavior of the DLL. A Configuration file is maintained. It contains the names of the executables of all the applications, which need to be intercepted.

A persistent layer of abstraction is injected between the browser and its cache, using popular tools like the API Interception [6] and the Browser Helper Objects [7]. This layer separates the content downloaded by each web site by keeping a record of the web site from where the request originated. As a result, every object in the Internet Cache is owned by a web site or by multiple web sites. To own an object in the Internet Cache, a website has to download it from the web at least once. A website can only access the objects owned by it, exclusively or shared with any other website. In this way, no website can see the content downloaded by any other website. Hence, the Timing Attacks are foiled.

3. Timing Attacks

The Timing Attacks can be classified into three kinds. They are summarized below and a detailed description can be found in [3].

3.1 DNS Attacks

An attacker can measure the time needed to retrieve a web object, and hence distinguish a cache hit from a cache miss using java and/or java script or at the server[3] . Subsequently he can deduce, if the user has visited a particular web site [3].

An attacker needs to calculate the threshold value T, which is the average time required to access any object. He can assume that the time needed to access a particular web object O is t_r. If $t_r < T$ then he can safely conclude that it is a hit. If $t_r > T$ he can conclude that it is a miss. If t_r =T, then the attacker can randomly choose a hit with probability P; the probability of concluding a hit for a case when t_r is exactly equal to T.

The attacker chooses P to be exactly equal to T as ½. The attacker can determine the value of T by a "four-sample" method [3]. He prepares four test cases. Two of which are surely going to be hits (H1, H2) and the other two, which are going to be, misses (M1, M2). T can be calculated by using the following equation:

$$T=\{\text{Max}(H1, H2) + \text{Min}(M1, M2)\} / 2$$

He can compute the accuracy separately for hits and misses and take the lower value. Therefore an accuracy of 98% would mean that an attacker could get at least 98% hits and 98% misses right [3].

3.2 DNS Attacks

The DNS servers and the resolver cache previously requested DNS translations in the server cache and the resolver cache respectively. These translations can be used to gather information about the users' previous activity. An attacker can initiate a DNS lookup by using java or java script to measure the time taken for the lookup. As illustrated in Section 2, he can easily determine if a lookup has been performed recently and deduce that the user has recently visited a particular website.

3.3 Cache Cookies

A cookie is a small data file, which is stored by the server into the client's cache as part of a HTTP response. Cache cookies can be stored in the client's cache without the knowledge or consent of the client. Any HTML page can download a cookie by using the <IMG> tag. A server can easily detect the presence or the absence of a cookie using the method explained in Section 3.1. In addition a cache cookie stored by one website can be easily read by any other website (unlike traditional cookies). All these attributes make cache cookies dangerous to the privacy of the web user [3].

4. Existing Countermeasures

- *Turn off caching*: This solves the problem, but WWW cannot withstand the bandwidth demand imposed by switching caching off. In addition surfing the web can become excruciatingly slow.
- *Turn off user level caching*: This solves the problem at the user level, but it cannot stop the attacker from reading second level caches. In addition it will overload the proxy server as well as the network.
- *Altering hit or miss performance*: This is of limited value; theoretically, the hits can be made to be as slow as the misses, and hence become indistinguishable from the misses. This leads to degradation in the performance.
- *Turn off java and javascript*: Java and javascript are so widely used that it is very unrealistic to ask any user to turn it off. In addition, even if they are turned off, the timing can be measured on the server side as explained in [3].
- *Domain tagging:* The Domain Tagging method is introduced in [3]. Currently, all cache objects are tagged with their access URLs when stored in the Internet Cache. This method proposes the addition of another tag called the domain-tag, which is the domain name of the page that the user is viewing. In the existing

mechanism, for a cache hit both the tags must match else an access is a cache miss. However, Domain Tagging does not solve the problem of cache cookies. Moreover, its implementation is not cost effective. This leads to doubts about the benefits accrued by the implementation of this approach in existing browsers.

5. Proposed Solution

5.1 Terminology

The following terminology is used in the remaining part of this paper to refer to entities involved in a consistent manner.

- *Web Object*: Any object requested by the browser from the web server or the cache .It can be an HTML page, an image file or any other object which can be rendered by the browser
- *Source*: The source (S) for a given web object (O) will be <address bar>, if this object has been requested by typing the address in the address bar. If O has been requested by using a hyperlink or an applet or a java script, then the URL of the page containing the hyperlink or the applet or the javascript code is the source. If an object is requested because it is embedded in a web page using the (<IMG> tag), then the URL of this web page will be the source of this object.
- *Target*: The requested web object or the DNS record is the target irrespective of how it is requested.
- *Persistent Storage:* A depository for storing the Source-Target relationships. We are using a flat file as a persistent storage, but it can be implemented by using any other form of Persistent Storage available.
- *Configuration File***:** This file is used for storing the applications from which the calls need to be intercepted.

5.2 Technique

Consider a scenario in which a user goes to the web page B by clicking a hyperlink on web page A. The web page B becomes the target and A becomes the source. The Browser Helper Object (BHO) intercepts the *InternetExplorer* events. The BHO captures the requested URL before it is passed on to *Wininet* APIs. It stores this URL in the Persistent Storage and returns the control to IE. This URL forms the source for subsequent requests. IE is made to call the new APIs instead of the APIs of the operating system using API interception (explained in Section 6.3). The web page being currently rendered becomes the source for this URL.

If the source-target record is present in the Persistent Storage, the new APIs call the Windows' APIs and the browser follows its default course of action. If the record is absent and the requested object does not belong to the domain of the source, the input parameters of the request are modified by the new APIs, and passed on to the Windows' APIs. The new set of parameters force retrieval from the network even if that object is present in the Internet Cache. Once the object has been successfully retrieved, a new source-target relationship is inserted in the Persistent Storage for subsequent requests that might be made for the same target from the same source. This strategy is followed for all web objects and DNS requests.

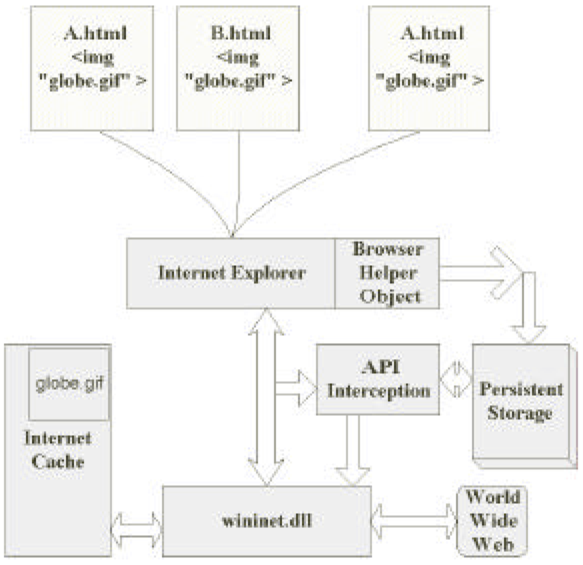

Figure 1: Internet Explorer with an injected BHO & API Interception

6. System Mechanisms

This Section describes API interception and the Browser Helper Objects (BHO) and Windows 2000 DNS. API interception and BHO has been used to unobtrusively modify the default behavior of the browser.

6.1 Internet Explorer Controls and DLLs

The Internet Explorer (IE) or the *InternetExplorer* object serves as a container for the *WebBrowser* control [7]. The *WebBrowser* control is both an ActiveX control and an Active Document host.IE is just a wrapper to the functionality implemented in the system components: *Shdocvw.dll,*

. *Shdocvw.dll* contains the *WebBrowser* control and provides the browsing capability for the host. *Wininet* is a DLL containing a set of APIs, which abstract the functionality provided by Winsock for making HTTP, FTP or Gopher requests.

6.2 Domain Name System (DNS) Cache

Windows 2000 provides a caching DNS service. The resolver caches the DNS queries on the client machine. Any application, which needs to perform a DNS query, uses the *DnsQuery* API in *dnsapi.dll*. When a name is submitted to the resolver, the latter first checks the cache. If the name is present in the resolver cache, the data is returned to the user. If the name is absent from the resolver cache, the resolver fetches the Resource.

6.3 API Interception

In the Windows 2000 scheme, all executables maintain an Import Address Table (IAT). The IAT contains references to the DLLs and the Methods/APIs contained therein. This table is generated at run time. When needed, the application calls the methods/APIs in the DLL by using the addresses in IAT. In order to intercept an API, an application is forced to call a different method, which is present in a secondary DLL, by replacing the address of the original DLL and the original APIs with the addresses of the secondary DLL and secondary APIs respectively. The APIs in the secondary can call APIs in the primary DLL. This secondary DLL is called the injected DLL. [6].

6.4 Browser Helper Object (BHO)

Browser Helper Objects are in-process Component Object Model (COM) objects loaded by the Internet Explorer every time it starts up [7]. A BHO is notified of browser's typical events, such as *GoBack, GoForward, DocumentComplete.* BHOs can also intercept and manipulate events of *InternetExplorer* using standard COM interfaces.

Each time a new instance of IE starts, it looks for Class Id's (*CLSIDs*) in the registry. For each *CLSID* listed in the registry, every instance of IE starts an instance of a helper object as an in-proc server in the address space of IE. The *CLSID* can be created automatically or manually for any helper object. In this paper, a BHO is used for logging the URLs visited by the user in a particular instance or across instances of IE.

7. Implementation

In the following sections we present the algorithms, which are used to implement the proposed technique. These algorithms have been carved around the above system mechanisms.

7.1 Timing Attack Prevention

IE makes the following API calls to retrieve any web object from the web, in the given order. These methods are implemented in the *wininet.dll* available in Microsoft Windows 2000**.**

1. *InternetOpen*	8. *HttpSendRequest*
2. *InternetConnect*	9. *InternetReadFile*
3. *HttpOpenRequest*	10. *InternetCloseHandle*

The following algorithm is used to prevent Timing Attacks.

The *BEFORENAVIGATE2* event of the Internet explorer is intercepted by the BHO. This BHO extracts the URL of the requested web page and stores it in the persistent storage and returns the control to the browser. Any other URL that is requested by the browser due to the re-direction by the web server or due to an alias of the URLs is not stored by the BHO into Persistent Storage. The URL that is stored, serves as the Source URL for the request that is made by the browser due to a particular embedded image tag or any applet or java script or a hyperlink embedded in the website that is currently being rendered. The browser calls Wininet APIs in the sequence mentioned above

The Wininet API *HttpOpenRequest* is intercepted. *InternetQueryOption* returns the requested URL when called with the

HINTERNET handle. This URL becomes the target URL.

A search is made in the Persistent Storage for the source-target combination by using the source URL, which is determined by the BHO, and the target URL that is determined in step 1 of this algorithm.

If the source-target pair is found in the Persistent Storage or if both the source and the target are from the same domain the *HttpOpenRequest* in wininet.dll is called with the original parameters passed by the browser.

If the source-target pair is not found in the Persistent Storage, then the injected DLL forces the request to be resolved by the web server by Oring the dwFlags parameter with *INTERNET_FLAG_PRAGMA_NOCACHE*, even if a cached copy exists in the Internet Cache.

If the web object is retrieved successfully an entry is made in the Persistent Storage. The requested URL and its corresponding source URL that is found by the BHO make a source-target pair. The source for everything else embedded in the webpage that is retrieved with the webpage is the URL of the webpage. All these pairs are written in the Persistent Storage for future reference. If the web object is not retrieved successfully, then the request fails and the error code is returned.

7.2 Countering DNS Attacks

The following algorithm is used for countering the DNS attacks. This algorithm branches out of the above algorithm after step 2.

1 After Step 2 the following algorithm is executed for foiling of DNS attacks

The input parameter *lpstrName* is the domain name whose resolution is requested. The API *DnsQuery* contained in *dnsapi.dll* is intercepted. The source URL is again taken from the Persistent Storage where it is stored by the BHO. The injected DLL searches the Persistent Storage for the source-target record by using the source URL and the target URL determined in step 1 of this algorithm.

If the Source-Target record is found in the Persistent Storage, the *DnsQuery* API in *dnsapi.dll* is called without modifying the original parameters passed by the browser. The output of the above method is subsequently returned to the browser.

If the Source-Target pair is not found in the Persistent Storage then the injected DLL forces the request to be resolved by the original server, by *OR-ing* the *fOptions* parameter with *DNS_QUERY_BYPASS_CACHE,* even if a cached copy exists in the resolver Cache.

If the DNS record is retrieved successfully, a source-target record is written on the Persistent Storage. The Domain Name Record requested, the source found by the BHO and this URL make the source-target pair.

If the DNS record is not retrieved successfully then the request fails and the error code is returned.

In a similar fashion, a request for *B.html* is made and the retrieval of the *<AddressBar>: B.html* record from the Persistent Storage fails; therefore, the request is sent to the web server. In addition, *B.html* is downloaded and another source-target record is written on the Persistent Storage. *B.html* also has an embedded *<IMG>* tag and needs *globe.gif*. Hence, the injected DLL searches the Persistent Storage for the second time and finds that the corresponding record does not exist, so it forwards the request for *globe.gif* to the web server inspite of the fact that a copy of *globe.gif* is present in the Internet Cache. This file is downloaded from the web and a source-target record is written on the Persistent Storage.

A.html is requested again from the browser of the same client. The injected DLL looks for the source-target entry in the Persistent Storage and finds it. It retrieves *A.html* and similarly *globe.gif* from the Internet Cache and IE renders the page along with the image.

7.3 Example of a DNS/Timing Attack Prevention

Consider a scenario: IE is rendering a page *B.html* which requests DNS resolution / retrieval for / of *www.GO.com*. The injected DLL checks for the presence of a source-target record of the form *B.html: www.Go.com* in Persistent Storage. It does not find the source-target record, hence it calls the original API, which in turn forwards the request to the DNS server/Web Server hosting *B.html:www.Go.com*. If the resolution/ retrieval is successful a source-target record entry is made

Persistent Storage. Later, when a request is made for the same target from a different source, the request will be forwarded to the DNS/Web servers, and not serviced from the cache, as the source-target record will be absent.

7.4 Countering cache cookies

The technique proposed in this paper prevents referencing of cache cookies across websites. Unless the source of the cookie matches to the source of the requestor the search for the cookie will result in a miss. This is also analogous to web cookies where only the host which has created a cookie at the client can read it. In this way no website will be able to track the user's history by using cache cookies.

The algorithm in Section 7.1 foils any attempt, to access the cookie by any non-originating web site. The cookies downloaded by the browser also have a corresponding entry in the persistent storage as explained in step 8. Any subsequent attempt to access the cookie is routed to the new APIs, which verify the source of the cookie in step 5. If this source URL matches the requestor URL, then the cookie is retrieved from the cache. Otherwise, a network request is made to download the cookie in step 7. The only difference is that instead of *globe.gif*, the <*IMG*> tag will contain *cookie.txt*.

8. Performance and Overheads

The above system was implemented and tested using Internet Explorer 6.0 on a 2 GHz Pentium IV machine in the LAN whose gateway is connected at 10 MBPS and 512 MB RAM. Windows 2000 (Service Pack 2) is used for calculating the performance.

The objective of the experiments was to estimate the overheads associated with the system described in this paper. Two experiments were performed to calculate the time taken for Internet Cache hits and misses and the DNS cache hits and misses

8.1 Simulation of a Cache Attack.

The Internet Cache for IE and the Persistent Storage are flushed. As a result, there are no files in the cache and no records in the Persistent Storage. IE is forced to retrieve images across domain. An HTML file in the domain *www.public.asu.edu*, containing a java script capable of rendering N number of images (11KB each), from the domain *calypso.eas.asu.edu* is downloaded by Internet Explorer while the 'hit' and the 'miss' times corresponding to N are recorded. The following table shows the timings in seconds.

Table 1: Hit and Miss times for IE Cache

Average Timings (Seconds)				
	Without the Technique		With the Technique	
N	Miss	Hit	Miss	Hit
25	1	<1	1	<1
50	3	< 1	5	1
75	4	1	6	1
100	5	1	7	1
125	6	1	9	1
150	7	1	11	1

The above table shows that with this technique the timing of a hit increases by less than a second when a web page having 25 images of 11KB each is rendered by the browser. In addition, the maximum difference in the miss times is 4 seconds for a web page having 150 images. It amounts to 26.6 milliseconds for each image. Considering that an average web page will have 10 images, the extra time consumed will be less than half a second.

8.2 Simulation of a DNS Attack

The resolver cache and the Persistent Storage are flushed. IE is forced to do DNS lookups, and the 'miss' times and the 'hit' times are recorded. The following table shows timings in microseconds.

Table 2: Hit and Miss times for a DNS Resolver Cache

Average Timings (Microseconds)			
Without the Technique		With the Technique	
Miss	Hit	Miss	Hit
4780	976	6060	3000

It is indicated in the above table that the miss timings increase by just over 1 millisecond and the hit timings increase by just over 2 milliseconds for each DNS lookup. This is attributed to the time needed to update Persistent Storage for maintaining Source-Target records. This overhead is needed to provide higher degree of privacy to the user.

8.3 Additional misses in the Cache

This system induces the overhead of additional misses, which in the default case would have been hits. Users repeatedly follow specific paths or subsequences while browsing. In [10] it is shown that there are a high number of repetitions for a given subsequence of size less than 5. A possible hit will be converted to a miss only the first time the subsequence is traversed. Hence the above number will depend on the number of subsequences traversed by a given user. The overhead will vary across users who can decide between the classic trade off of security and speed.

9. Future Work

Besides the vulnerabilities discussed above, a couple of other need to be plugged.

9.1 Attack on Cache Engines

Cache engines are hardware-based caches, which are attached to the routers serving clients with high volume of Internet traffic. These cache engines are susceptible to intrusion by the method explained in Section 7.1 Configuring the parameters of the cache engine and modifying the request parameters in *HttpOpenRequest* will plug this loophole.

9.2 Attack on DNS Server Caches

DNS Server cache can be targeted to execute Timing Attacks. Introducing a delay in the retrieval from a cache whenever a DNS Lookup request is made can foil this attack. As illustrated in Section 8.2, this delay will be in the range of microseconds, and it will not be a big deterrent to speed.

10. Conclusion

As the browsers become complex in a short span of time, the manufacturers of browsers will find it hard to apply antidotes to various approaches of intrusion that are being discovered rapidly. Any other user will not be able to fix the loopholes because of the 'closed source code' for most of the browsers. The technique illustrated in this paper not only foils timing attacks but also presents a new strategy for users to safeguard their privacy without being dependent on the browser manufacturer to come out with patches.

11. References

[1] *Graeme Bennett.1998*.PC Buyers Guide

[2] Jia Wang, A Survey of Web Caching Schemes for the Internet, Cornell Network Research Group (C/NRG), ACM Computer Communication Review, 29(5): Pages 36-46, October 1999.

[3] Edward W. Felten, Michael A. Schneider.2000.Timing Attacks on Web Privacy. *Proc. of 7th ACM Conference on Computer and Communications Security*

[4] *C. Yoshikawa*.1997.Microbrowser: An Extensible Web Browser Architecture. *Master's Report, University of California, Berkeley*

[5] Matrix Information and Directory Services (MIDS).

[6] Jeffrey Richter.1999. *Programming Applications for Microsoft Windows, Fourth Edition*. Seattle: Microsoft Press.

[7] Scott Roberts.1999.*Programming Internet Explorer, Fourth Edition* 1999.Seattle: Microsoft Press.

[8] Global Reach: 2001 Global Internet Statistics. Available from World Wide Web:(http://www.glreach.com/globstats/)

[9]) Browser Statistics 1999-2002. Refsnes Data. Available from World Wide Web:(http://www.w3schools.com/browsers/browsers_stats.asp/)

[10] Linda M Tauscher. 1996. Evaluating History Mechanisms: An Empirical Study of Reuse Patterns in WWW Navigation. *Master's Report, University of Calgary*.

[11] EPIC.2000. *Surfer Beware: Personal Privacy and the Internet, Electronic Privacy Information Center, Washington DC*.

[12] Microsoft Product Team. 2000. *Windows 2000 Professional Resource Kit and the Windows 2000 Server Resource Kit.* Seattle: Microsoft Press.

Sponsor Acknowledgment: Effort sponsored by the Defense Advanced Research Projects Agency, under agreement numbers F30602-99-1-0517 and N66001-00-1-8920. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

Sponsor Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, or the U.S. Government.