

Timing Attacks on Web Privacy

Edward W. Felten and Michael A. Schneider
Secure Internet Programming Laboratory
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

ABSTRACT

We describe a class of attacks that can compromise the privacy of users' Web-browsing histories. The attacks allow a malicious Web site to determine whether or not the user has recently visited some other, unrelated Web page. The malicious page can determine this information by measuring the time the user's browser requires to perform certain operations. Since browsers perform various forms of caching, the time required for operations depends on the user's browsing history; this paper shows that the resulting time variations convey enough information to compromise users' privacy. This attack method also allows other types of information gathering by Web sites, such as a more invasive form of Web "cookies". The attacks we describe can be carried out without the victim's knowledge, and most "anonymous browsing" tools fail to prevent them. Other simple countermeasures also fail to prevent these attacks. We describe a way of reengineering browsers to prevent most of them.

1. Introduction

This paper describes a class of attacks that allow the privacy of users' activities on the Web to be compromised. The attacks allow any Web site to determine whether or not each visitor to the site has recently visited some other site (or set of sites) on the Web. The attacker can do this without the knowledge or consent of either the user or the other site. For example, an insurance-company site could determine whether the user has recently visited Web sites relating to a particular medical condition; or an employer's Web site could determine whether an employee visiting it had recently visited the sites of various political organizations.

The attacks work by exploiting the fact that the time required by the user's browser to perform certain operations varies, depending on which Web sites the user has visited in the past. By measuring the time required by certain operations, a Web site can learn information about the user's past activities. These attacks are particularly worrisome, for several reasons:

- The attacks are possible because of basic properties of Web browsers, not because of fixable "bugs" in a browser.
- The attacks can be carried out without the victim's knowledge.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'00, Athens, Greece.

Copyright 2000 ACM 1-58113-203-4/00/0011 ..\$5.00

- Standard Web "anonymization" services do not prevent the attacks; in many cases they actually make the attacks worse.
- Disabling browser features such as Java, JavaScript, and client-side caching do not prevent the attacks.
- The only effective ways we know to prevent the attacks require either an unacceptable slowdown in Web access, or a modification to the design of the browser.
- Even modifying the browser design allows only a partial remedy; several attacks remain possible.

1.1 Why Web Privacy Matters

There is now widespread concern about the privacy of users' activities on the World-Wide Web. The list of Web locations visited by a user often conveys detailed information about the user's family, financial or health situation. Consequently, users often consider their Web-browsing history to be private information that they do not want unknown parties to learn. Of course, visiting a Web site necessarily leaks some information to that site; but users would like some assurance that information about their visits to a site is not available to arbitrary third parties.

Thus far in the short history of the Web, two types of problems have led to compromise of users' Web-browsing histories, and remedies are available for both types.

First, some Web sites gather information and then reveal it to third parties without the informed consent of users. (Alternatively, some sites cause users' browsers to reveal information directly to a third-party site.) These problems have been dealt with by the use of privacy policies and third party audits of Web sites. While these remedies leave much to be desired, they do give users a chance to guess where information will go after it is revealed to a law-abiding site.

Second, some implementation bugs in browsers have provided opportunities for unscrupulous third parties to gather information without the user's consent. While these bugs are part of an unfortunate pattern of security bugs in browsers, each bug by itself has been fixable.

The attacks we describe in this paper admit no such remedy. Because information about visits to a site is not controlled by that site, privacy policies, auditing, and trust in sites are not effective remedies. Because the attacks are not caused by browser bugs, they cannot easily be fixed.

2. Exploiting Web Caching

The first timing attack we will discuss exploits Web caching. We first review how Web caching works, and then discuss the attack.

2.1 Web Caching

Accessing Web documents often takes a long time, so Web browsers use *caching* — they save copies of recently-accessed files, so that

future accesses to those files can be satisfied locally, rather than requiring another time-consuming Web access. Caching is relatively effective in reducing perceived access times of Web pages.

The purpose of caching is to make accesses to recently-visited files faster. The attack exploits this by measuring the amount of time required to access a particular file. If that file is in the user's cache, the access will be fast; if not, it will be slow. By measuring the access time, an attacker can determine whether or not a file is in the user's cache, and hence whether or not the user has accessed the file recently.

2.2 An Example

To illustrate what we are talking about, we will now give a simple example. Readers should bear in mind that this is only one example; countermeasures that would prevent this simple example attack will not necessarily prevent more sophisticated versions of the attack.

Suppose that Alice is surfing the Web, and she visits Bob's Web site at `http://www.bob.com`. Bob wants to find out whether Alice has visited Charlie's Web site (`http://www.charlie.com`).

First, Bob looks at Charlie's site, and picks a file that any visitor to the site will have seen. Let's say Bob picks the file containing Charlie's corporate logo, at `http://www.charlie.com/logo.jpg`. Bob is going to determine whether the logo file is in Alice's Web cache. If the file is in her cache, then she must have visited Charlie's Web site recently.

Bob writes a Java applet that implements his attack, and embeds it in his home page. When Alice views the attack page, the Java applet is automatically downloaded and run in Alice's browser. The applet measures the time required to access `http://www.charlie.com/logo.jpg` on Alice's machine, and reports this time back to Bob. If the time is less than some threshold (say, 80 milliseconds), Bob concludes that Alice has been to Charlie's site recently; if it is greater than the threshold, he concludes that she has not been to the site recently.

2.3 Measuring Access Time

The main component of the attack is a measurement of the access time for a particular Web URL. There are several ways the attacker could measure this time. Java and JavaScript both have facilities for measuring the current time and for loading an arbitrary URL; these can be combined in the obvious way to measure the time required to load a URL.

Java and JavaScript provide the most accurate means of measuring access time, but the attacker can get a sufficiently accurate measurement even if Java and JavaScript are disabled. This is done by writing a Web page that loads three files in sequence:

1. a dummy file on the attacker's site,
2. the file whose access time is to be measured, and
3. another dummy file on the attacker's site.

The attacker's Web server can record the time at which it receives requests 1 and 3; subtracting yields an approximation to the time required to perform step 2. (We omit the straightforward but tedious details of how to write HTML that causes popular browsers to make serialized accesses to files.)

Any of these methods can be used to perform a measurement invisibly to the victim. The Java and JavaScript programs need not display anything, and the three-step method can also be implemented in a way that does not affect the appearance of the page doing the measuring.

To make the measurement as accurate as possible, the attacker can measure some control cases — known hits and known misses

— along with the main measurement. Known misses can be generated by accessing nonexistent (e.g. randomly generated) URLs on the same site as the file to be measured. Known hits can be generated by replicating the main measurement. The second and subsequent measurements will be known cache hits. By measuring several known hits and misses, the attacker can choose a threshold for discriminating between hits and misses. We show below that the attacker can pick a good threshold and make determinations with high accuracy, even when the control cases are restricted to only two known hits and two known misses.

Another way for the adversary to improve the accuracy of his measurement is to combine the results of several measurements. For example, there might be several static image files linked from Charlie's home page. If Bob measures the access time for each of these images individually, this gives him several estimates of the likelihood that Alice had visited Bob's site. By using standard statistical techniques, Bob can combine these estimates to obtain a more accurate estimate.

2.4 Delivering an Attack

The measurements described above are all performed by HTML pages that are viewed by the victim. In order to carry out the attack, the attacker must cause the victim to view an HTML page written by the attacker. There are many ways to do this.

1. An unscrupulous organization that runs a popular Web site could put measurement code into its site.
2. A Web advertising agency could add measurement code to the banner ads it distributes.
3. The attacker could create a Web site that is engineered to appear near the top of the list returned by a popular search engine when the user searches for a common search term.
4. The attacker could send the victim an email message referring to some enticing content, such as a low-price sale, or an award certificate, on the attacker's Web site.
5. The attacker could send the victim an email message with an HTML message body. If the victim uses an HTML-enabled mail reader (and most popular mail readers are now HTML-enabled), the measurement would be performed when the victim read the email message. The message could be disguised as an unwanted "spam" message, so that the victim did not notice anything unusual.

2.5 Accuracy of the Measurements in Practice

We performed a series of experiments to determine how accurately an attacker could distinguish cache hits from cache misses. In the experiments, we ran the Netscape Navigator 4.5 browser on a Windows NT 4.0 (Service Pack 4) PC with a 350 MHz Pentium II processor and 256 Megabytes of RAM. (Preliminary results indicate that our conclusions would have been unaffected had we used Internet Explorer rather than Netscape Navigator.) This PC was connected to our department's network via a 10BaseT link.

The experiments used our department's Web server, which is on the same switched departmental network as the client browser. Because the client and server are so close together in our experiments, cache miss times are much lower than they would be in practice, thereby making it artificially difficult to distinguish hits from misses. In practice, we expect miss times to be longer, giving the attacker even higher measurement accuracy than our experiments show.

2.5.1 Computing the Accuracy

The ultimate goal of our experiments will be to determine how accurately an attacker can distinguish cache hits from misses. We will characterize the accuracy as a percentage, which tells us what percentage of unknown references the attacker will be able to correctly characterize. We now consider how to compute this percentage.

The attacker will choose a threshold value T , such that, given an unknown reference r with access time t_r , he will decide r is a hit if $t_r < T$, and will decide r is a miss if $t_r > T$. If $t_r = T$, he will decide randomly, choosing “hit” with probability p . Given T and p , we can compute the attacker’s accuracy by seeing what percentage of references are mischaracterized.

To determine the accuracy, we first compute the accuracy separately for hits and misses. In other words, we compute the fraction of hits that are correctly characterized, and the fraction of misses that are correctly characterized. We then take the *minimum* of these two values, so that an accuracy of (say) 98% means that the attacker can get at least 98% of hits right, and at least 98% of misses right.

But how would the attacker choose T and p ? Different methods are called for, depending on whether or not the attacker knows the performance characteristics of the user’s system. We will consider two cases. In the first, the attacker knows the distributions of access times for hits and misses. In the second case, the attacker does not know these distributions.

2.5.2 Computing Accuracy from Known Time Distributions

To choose T and p optimally, the attacker needs to know two things:

1. the distribution of access times for references that hit in the cache, which we will represent with a function h such that $h(t)$ is the probability that a hit will have access time equal to t ; and
2. the distribution of access times for references that miss in the cache, which we will represent with a function m such that $m(t)$ is the probability that a miss will have access time equal to t .

From these, we calculate two additional functions: $H(t)$ is the probability that a hit will have access time less than t , and $M(t)$ is the probability that a miss will have access time greater than t .

Now, if we choose the parameters T and p , the accuracy is

$$A(T, p) = \min(H(T) + ph(T), M(T) + (1 - p)m(T)).$$

To find the optimal values for the parameters, we loop over all values of T . For each value of T we find the value of p that maximizes $A(T, p)$, by first finding the value p^* such that

$$H(T) + p^*h(T) = M(T) + (1 - p^*)m(T),$$

and then choosing p as follows:

1. If $p^* \leq 0$, choose $p = 0$.
2. If $p^* \geq 1$, choose $p = 1$.
3. If $0 < p^* < 1$, choose $p = p^*$.

Finding the optimal T and p is relatively efficient.

2.5.3 Computing Accuracy with Unknown Time Distributions

In practice, the attacker often does not know the distributions $h(t)$ and $m(t)$. To find a good value for T , the attacker can employ the following “four-sample” method:

1. Measure the time taken by the client to retrieve two known hits (H_1, H_2), and two known misses (M_1, M_2).

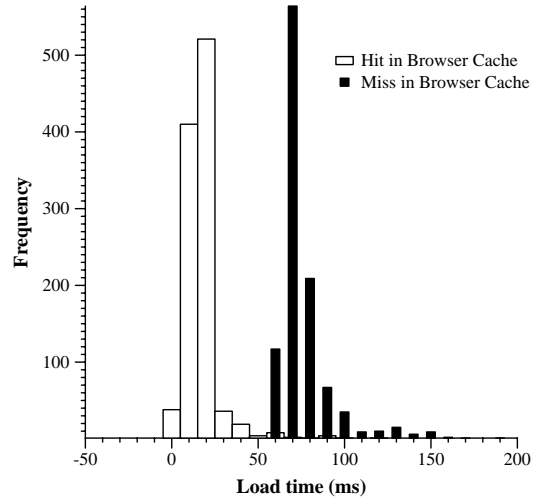


Figure 1: Distribution of access times for known cache hits and known cache misses, as measured by a JavaScript program embedded in the attacker’s Web page.

2. Choose T as the mean of $\max(H_1, H_2)$ and $\min(M_1, M_2)$.
3. Choose p , the probability of concluding a hit for measurements exactly equal to T , as $1/2$.

Intuitively, the four-sample method will achieve high accuracy if there is not much overlap between the distributions for hits and for misses. If almost all misses take longer than almost all hits, high accuracy will result, even if the attacker does not know the distributions of hit and miss times.

2.5.4 Measuring with Client-Side JavaScript

Figure 1 shows the distribution of hit times and miss times when the attacker measures access latency using a JavaScript program embedded in a Web page. The JavaScript program runs on the client browser and measures the time both before and after loading a file. Subtracting the two measured times yields the access latency, which is shown here.

Figure 1 shows that hit times are almost always considerably lower than miss times. An attacker who knows these distributions will set his threshold T to be 60 ms, his p to be 0.14, and will achieve 98.5% accuracy. An attacker who does not know the distribution, but uses the four-sample method, will achieve a mean accuracy of 97.7%.

2.5.5 Measuring on the Server Side

Using a Java or JavaScript program to measure access times is the most effective strategy for the attacker, but a few victims will make this impossible by turning off JavaScript. Even if JavaScript is turned off, the attacker can still measure the access time by creating an HTML page that hits a known URL on the attacker’s site, then loads the file that the attacker wants to test, then hits another known URL on the attacker’s site. The attacker’s Web server can measure the times at which it receives the two hits. Subtracting these two time measurements yields a measure of how long it took to load the file that the attacker wants to test.

This approach will typically lead to a higher variance in measured times, since the time for the first and third hits to reach the attacker’s Web server will vary. We performed an experiment to determine how well the attacker could distinguish cache hits from misses in this scenario.

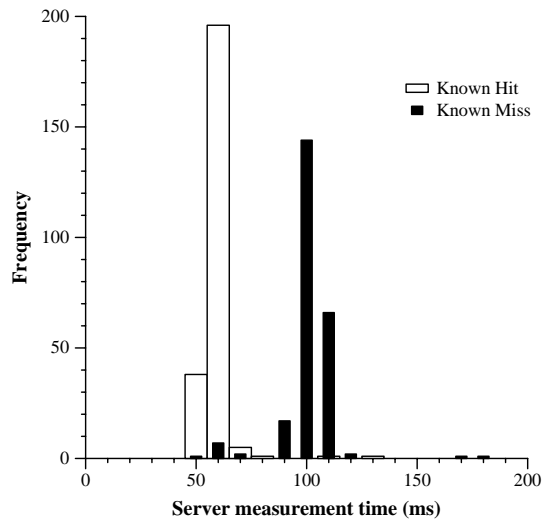


Figure 2: Distribution of access times for known cache hits and known cache misses, as measured by server-side time measurement.

Figure 2 shows the distribution of times we measured for known cache hits and known cache misses. The two distributions overlap a bit but are generally disjoint. To be precise, an attacker who knows the distributions will choose T to be 60 ms and $p = 1.0$, giving an accuracy of 96.7%. An attacker who does not know the distributions, but uses the four-sample method, will achieve a mean accuracy of 93.8%.

Summing up these two experiments, we can see that an attacker can determine with high accuracy whether or not a particular file is present in the victim’s cache.

2.6 Uncacheable Files

Not all files on the Web are cacheable. Some files are explicitly marked as uncacheable by the Web server supplying them, and some files are inherently uncacheable, for example because they are dynamically generated. A study [9] by Wolman and colleagues at the University of Washington found that about 60% of Web accesses are to cacheable files¹. The Washington group’s data also shows that the fraction of files that are cacheable is not changing appreciably over time [8].

Uncacheable files do not always prevent timing attacks. Pages that are dynamically generated often include embedded images that are cacheable — for example, a portal’s news page might include the portal’s corporate logo. To carry out a successful attack, the attacker need only find one cacheable file that is referenced by the page.

3. Anonymization Tools Do Not Help

Several tools exist to provide “anonymous” Web browsing; examples include anonymizer.com [2], Crowds [6], the Lucent Personal Web Assistant [3], and Zero Knowledge Freedom [10]. These tools all send Web requests through some kind of intermediary which shields the address of the original requester.

¹In their study, a file was considered cacheable if version 2 of the Squid proxy cache [7] would have been willing to cache it.

Since accesses made through these systems are all subject to caching, none of these systems prevents the timing attacks discussed above. Indeed, these systems may well make the attack easier — routing cache misses through an intermediary makes misses even slower, which makes it easier for the adversary to distinguish hits from misses.

Some of these systems attempt to block Java and JavaScript from reaching the browser, which would force the attacker to rely on the less accurate server-side timing method. However, this is unlikely to prevent the attack entirely, and in any case there is doubt as to whether it is possible for a firewall to block Java and JavaScript in all cases [4].

An anonymizing tool could also modify Web sessions to mark every incoming file as uncacheable. This would be equivalent to turning off caching at the Web browser, which is discussed in Section 7.

4. Exploiting DNS Caching

Attacks on Web caching can be prevented by turning off Web caching. This has a high performance penalty, since caching has a major effect on overall Web performance.

Even if Web caching is turned off, an attacker can exploit other types of caching, such as DNS caching, to learn about a user’s activities.

4.1 DNS Caching

The Domain Naming System (DNS) [1] is the mechanism that looks up human-readable “DNS names” like `www.whitehouse.gov` so that they can be translated into the numeric IP addresses like `192.168.13.4` that the Internet protocols use. Every DNS address must undergo this translation before it is used. This is done with the help of a set of server machines on the Internet.

Because a DNS lookup requires potentially expensive network communication, and because machines typically look up the same DNS addresses repeatedly, most machines keep a *DNS cache* that holds the results of recent DNS lookups. Subsequent lookups of the same address can be completed quickly by using the cached result.

As with a Web cache, a DNS cache records information about a user’s recent activities. In particular, if the user visits a Web server at `www.whitehouse.gov`, this will cause a DNS lookup for `www.whitehouse.gov`, so an entry for this address will appear in the DNS cache of the user’s PC. Subsequent attempts to look up `www.whitehouse.gov` will complete quickly because of the cache entry.

There are several ways for an HTML page to cause a DNS lookup. For example, a Java applet can directly ask to have a DNS lookup done; a JavaScript program can indirectly cause a DNS lookup to occur. By measuring the time required to look up a particular DNS address, an HTML page can determine whether or not a particular address appears in a user’s DNS cache, and can therefore determine whether the user accessed that address recently.

Such an attack can be delivered by any of the same methods as the Web caching attacks described above, via either a Web page or an email message.

4.2 Accuracy of the Measurements

Figure 3 shows the distribution of DNS lookup time for known hits and known misses to three different Internet sites, as determined by a Java applet. The three sites `cs.princeton.edu` (our department), `rutgers.edu` (about thirty kilometers from us), and `berkeley.edu` (about 4000 kilometers from us) were chosen as examples of DNS servers with different network latencies for DNS queries. The test client experienced a small latency

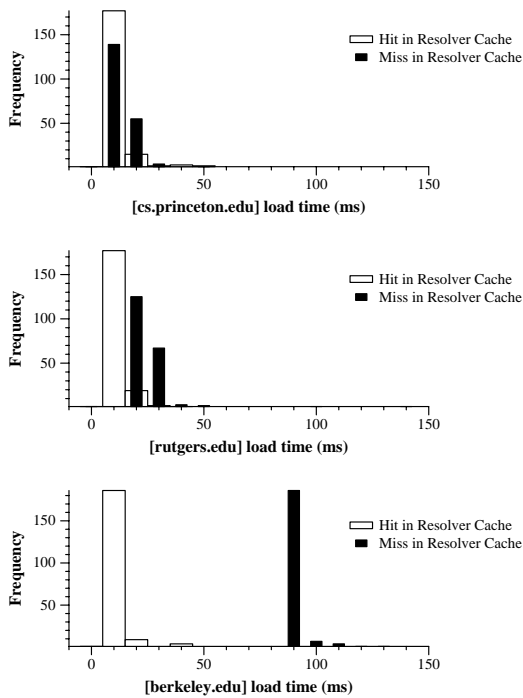


Figure 3: Distribution of DNS lookup time for known DNS hits and known DNS misses for three sites, as measured by a Java applet embedded in the attacker’s Web page.

for queries to `cs.princeton.edu`, a slightly larger latency for queries to `rutgers.edu`, and a somewhat larger latency in queries to `berkeley.edu`.

Computing the accuracy as described above in Section 2.5.2 for each site, we find that, for the three servers, an attacker would know the distributions would choose (T, p) to be (10 ms, 0.63), (20 ms, 0.15), and (60 ms, 0), respectively, giving optimal accuracies of 53.5%, 90.4%, and 100.0%, respectively. Using the four-sample method, an attacker who does not know the distributions will achieve mean accuracies of 53.5%, 87.7%, and 100.0% respectively.

The differences in these accuracy measurements demonstrate that these tests do not perform well when queries to the DNS server being tested experience a very small cache miss penalty. However, when the miss penalty is large enough to detect (as it is for almost all sites on the net) the tests lead to very high accuracies.

Thus, when the attacker chooses a DNS server with a measurable latency between the server and the victim, the attacker can effectively determine whether or not a particular entry is in the victim’s DNS cache.

5. Attacking Multi-Level Caches

So far we have described how attackers can learn the contents of caches on client workstations. Often, client caches are backed by additional caches that are shared by many clients. For example, a client PC’s DNS cache is typically backed by another cache that is shared at a departmental level. If an address is not found in the client’s DNS cache, a request is sent to the departmental cache. If the address is in the departmental cache, the result is provided to the client; if not, the departmental cache does a DNS lookup, which may involve lookups in additional caches. Similar forms

of multilevel caching are often used for Web content, backing a client browser cache with a departmental or institutional Web proxy which performs caching.

Multilevel caches are presumably effective in reducing average access time; that is the main reason for their existence. In principle, then, if an attacker determines that an access misses in the first-level cache, he may be able to tell whether it hits or misses in the second-level cache. This may allow attackers to gather even more information than is available from measuring first-level cache contents.

The typical difference between a first-level cache and a second-level cache is that second-level caches are usually shared between multiple client machines. For example, a departmental DNS cache might be shared by all of the client machines in the department. In general, a shared departmental cache aggregates information about the access patterns of individuals in the department: it records which files have been accessed by members of the department, but does not record which specific individual accessed which file. Depending on the circumstances, leaking aggregated information may be either more or less dangerous than leaking individual information.

5.1 Testing for Cache Sharing

If an adversary can test for the presence of files in second-level caches, he can also determine whether two clients share a second-level cache. This would be done by causing the first client to access a unique name (in a part of the namespace controlled by the attacker), and then determining whether that name is present in the second-level cache of the other client. If it is present, then the two clients must share a second-level cache.

An attacker can use this attack in several ways. He might use it to probe the organizational structure of an enterprise, exploiting the fact that employees who work in the same group, department, or location tend to share second-level caches. Alternatively, he might use it to find the location of an unknown client, by checking whether that client shares a second-level cache with any of a set of clients whose locations are known.

5.2 Accuracy of Measurements

We performed an experiment to determine how accurately an attacker could distinguish hits from misses in a shared departmental HTTP proxy cache. To do this, we disabled caching in a client browser and configured the client to use our department’s caching HTTP proxy. We then measured the access time for known hits and known misses in the proxy cache.

Figure 4 shows the results. The hit distribution and the miss distribution are nearly disjoint. Following the methodology of Section 2.5.2, we find that an attacker who knows the distributions will choose his threshold T to be 50 ms, and his p to be 0.58, achieving an accuracy of 96.7%. Using the four-sample method, an attacker who does not know the distributions can attain an accuracy of 95.7%. In other words, the attacker can effectively test for the existence of a file in a shared second-level cache.

6. Cache Cookies

Traditional web cookies are one way for sites to maintain persistent state on the clients who visit their pages. The cookie itself is a relatively small amount of data, “written” by the server and stored by the client as part of a normal HTTP request. Clients volunteer the contents of the cookie back to the same server on subsequent HTTP accesses as part of the HTTP request. Clients can store cookies across multiple browsing sessions; a cookie may have an expiry time associated with it beyond which it will be discarded by the client. [5]

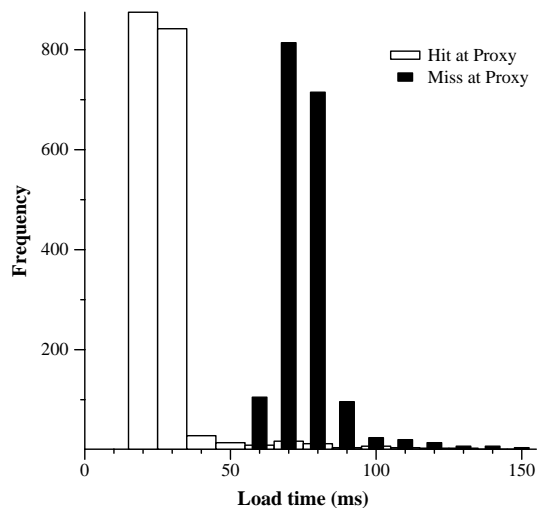


Figure 4: Distribution of access times, with client-side caching disabled, for known hits and known misses in a shared HTTP proxy cache, as measured by a JavaScript program embedded in the attacker’s Web page.

Sites might use the persistent state stored on each client, for example, to retain the site preferences of individual users. Cookies can also be used to tie multiple site visits to the same user. Most web browsers provide a method for accepting (storing) or rejecting (ignoring) web cookies on the client. This method allows clients to opt-out of the persistent state, perhaps for personal privacy reasons.

6.1 Cache Cookies

Using the methods described above, it is possible to develop a new, more intrusive, form of web cookies, which we call “cache cookies”. Servers can store cache cookies on clients who visit their pages, without the client’s knowledge or approval. Cache cookies are stored in the form of entries in the client’s web cache. By forcing a client to retrieve a specific URL (using an IMG tag, for example), the server can effectively write entries into the client’s cache, thus storing the cookie. To read the cookie, the server can use any of the measurement techniques described above to measure the retrieval time for the same URL. A hit indicates that the cookie is present, otherwise the cookie is not present.

Although cache cookies can be used to emulate traditional cookies, they differ in two important respects. First, they require no client side support. A server can use cache cookies to store information on a client without the user’s knowledge or approval. We know of no way that a browser can unconditionally block these sorts of cookies without an unacceptable performance loss.

Second, cache cookies can be written and read by different servers. The reading of traditional web cookies is restricted to the site which originally wrote them by web browser security. Implementing this type security for cache cookies would most likely require reengineering of the operation of the client’s web browser.

6.2 Emulating “Traditional” Web Cookies

A single cache file can be only present or absent, so it can store only a single bit of state. Traditional Web cookies store multi-bit values, so several files (one per bit, plus a few extra for error correction) must be put in the cache to emulate one traditional cookie. Since traditional cookies are used only to store very small amounts

of information (a unique identifier, for example), the number of cache files required is small enough to be manageable.

6.3 Cookie Reading Caveats

While the process of writing a cache cookie into the user’s cache is straightforward, checking for the existence of a cache cookie is slightly more complex. Checking for the presence of a file requires reading the file; and reading the file normally causes the file to be loaded into the cache, so it might appear that checking for the existence of a cache cookie has the side-effect of putting that cookie into the user’s cache. An additional technique is necessary to prevent cookie checking from destroying the information contained in the cookies.

To make reads nondestructive, the server must know, when serving a URL corresponding to a cache cookie, whether the client is reading or writing. One method of discrimination is to use the `Referer: HTTP` header. For example, suppose we have a cookie `URL cookie.jpg`. The cookie might be read by an html page `reader.html`, or written by another html page `writer.html`. Since the enclosing page is included in the `Referer: HTTP` header, the server can examine this header to determine whether to read or write.

When writing, the server should return the file with a very long expiry time. This will help to make sure that the cookie does not expire from the client’s web cache. When reading, the server should return the file with a very short expiry time. After a string of cookies has been read, the ones that were present already (which will have long expiry times) will persist in the cache. The cookies which were absent (and hit during the reading process) will expire quickly, returning them to the absent state. In this way, servers can read client-side cache cookies nondestructively.

6.4 Using Cache Cookies

Cache cookies have the advantage (from the attacker’s point of view) that they require no client-side support. In most cases the client would be completely unaware that cache cookies are in use. Worse yet, cache cookies can be used across sites (i.e. written by one site and read by another independent site). These qualities make cache cookies very dangerous to the privacy of web users.

7. Countermeasures

We now turn to the question of how to counter these attacks. Several types of countermeasures exist, but all are unattractive.

7.1 Turning Off Caching

The first countermeasure is simply to turn off caching. This will certainly prevent the attacks, but it imposes a big performance penalty. Indeed, it seems likely that the Web and the DNS infrastructure could not meet the bandwidth demands imposed by the removal of caching.

7.1.1 Turning Off First-Level Caching

Rather than turning off caching altogether, we could try to turn off first-level caching and rely on department-scale second-level caching. This has significant performance implications, and it still allows attackers to gather aggregated information from the second-level caches.

7.1.2 Implications of Transparent Caching

Recently, systems employing *transparent* Web caching have been devised. Transparent caching does not require clients to explicitly designate a cache. Rather, a transparent cache observes network traffic over some network segment, detects which traffic encodes Web requests, and automatically provides a Web cache for those requests, without the knowledge or consent of the requestors. Trans-

parent caching can be embedded in network components such as routers, and it offers improved performance for Web clients without requiring any reconfiguration at the client. Transparent caching cannot be turned off at the client, so one client, acting alone, may be unable to avoid using caching.

7.2 Altering Hit or Miss Performance

Another class of countermeasures relies on altering the performance of cache hits, in an attempt to make them harder to distinguish from misses.

Simply slowing down hits is of limited value. As long as hits are appreciably faster than misses, attackers will be able to extract useful information about cache contents. Of course, if we make hits as slow as misses, then attackers will be handcuffed; but in that case we might as well have turned off caching.

Another approach is to increase the variance in hit times in a randomized fashion. This also has only limited value. If a particular hit is noticeably faster than a miss, then the attacker will still be able to categorize it as a hit. If it is not noticeably faster than a miss, then it might as well have been a miss. In other words, this approach essentially converts some randomly chosen hits into misses, with a consequent loss of performance. The more hits we wish to disguise, the more performance will suffer.

Alternatively, we could try increasing the variance in miss times. This fails utterly. We cannot make misses faster, since they are already as fast as they can be. However, if we make a miss slower, we just make it easier to classify as a miss. Thus increasing miss variance can only make things worse.

7.3 Turning Off Java and JavaScript

Our final countermeasure is to turn off Java and JavaScript. This takes away the attacker's most accurate measurement tools, and hence lowers the accuracy of the attacker's measurements.

There are two problems with this approach. First, the attacker can still do reasonably accurate measurements, as the data presented above in Section 2.5.5 demonstrates. Second, Java and JavaScript are now so widely used on Web sites that it seems unrealistic to ask ordinary users to turn them off.

7.4 Countermeasures Summary

None of the countermeasures discussed here is attractive. At present, we would have to turn off caching, paying a high performance price, to prevent these attacks.

8. Domain Tagging

As discussed above, no practical countermeasure against our timing attacks is available today. We now describe a countermeasure, based on changing the browser's cache implementation to implement "domain tagging," that prevents some timing attacks.

Currently, each item in the browser's cache is tagged with the URL of the file of which it is a copy. We add another tag, called the domain-tag, which gives the domain name of the page that the user is viewing. A cache access is treated as a hit only if *both* tags match.

For example, suppose the user is currently viewing the page `http://search.yahoo.com/search/options`. All accesses to the cache while rendering this page (and any images and other material embedded in it) are given the domain-tag `yahoo.com`. For example, while fetching the image `http://us.yimg.com/images/yahoo.gif` (which is embedded in the search options page), the domain-tag `yahoo.com` is used.

If the user later visited `www.adversary.com`, this page could embed the `yahoo.gif` image, in an attempt to determine whether

it is in the user's cache. However, this attempt would come with domain-tag `adversary.com`, which is different from the domain-tag of the cached copy, so the result would be a cache miss. In short, the adversary's access to the page would miss in the cache, regardless of whether the user had visited Yahoo.

Domain tagging does transform some hits into misses, but this only occurs when the same file is accessed while viewing multiple domains. This situation is sufficiently rare that it should not pose much of a performance problem in practice.

Domain tagging would be effective in preventing some timing attacks on browser caches, but it fails to protect against other attacks. It will not prevent timing attacks that learn shared proxy cache contents (unless we augment the proxy-http protocol to carry domain-tag information). It will not prevent attacks on DNS caches. It also fails to prevent the use of cache cookies, although it does prevent a cache cookie set by one site from being seen by another site. Given the limited value provided by domain tagging, it is not clear whether it is worth implementing in real browsers.

9. Conclusion

Cache-based timing attacks are possible in many situations where references to data are cached. By measuring the time required to access a data element, a malicious program can determine whether that element is in a nearby cache, thereby allowing the program to learn whether that element has been accessed recently on the same machine (for local caches) or in the same group of machines (for shared caches). We expect that this type of attack will be possible in many situations besides the ones we have described.

Web technologies allow an attacker to control the sequence of data accesses on a remote machine, and hence to carry out cache-based timing attacks. An attack could be delivered by a Web page, or in an email message if the victim uses an HTML-enabled mailer.

We have described attacks that probe the contents of Web browser file caches, to learn a user's Web browsing history, and attacks that probe DNS caches, to learn which network addresses a machine has connected to recently.

We are not aware of any practical countermeasures to these attacks. There seems to be little hope that effective countermeasures will be developed and deployed any time soon.

Acknowledgments

Thanks to Drew Dean, Gary McGraw, Avi Rubin, Dan Wallach, Randy Wang, and the anonymous referees for their comments and suggestions.

10. REFERENCES

- [1] ALBITZ, P., AND LIU, C. *DNS and BIND*, third ed. O'Reilly and Associates, 1998.
- [2] ANONYMIZER. Web site at <http://www.anonymizer.com>.
- [3] GABBER, E., GIBBONS, P., MATIAS, Y., AND MAYER, A. How to make personalized web browsing simple, secure, and anonymous. In *Proc. of Financial Cryptography '97* (1997).
- [4] MARTIN JR., D. M., RAJAGOPALAN, S., AND RUBIN, A. D. Blocking Java applets at the firewall. In *Proc. of Internet Society Symposium on Network and Distributed System Security* (1997).
- [5] NETSCAPE COMMUNICATIONS. Persistent client state: HTTP cookies. Web site at <http://home.netscape.com/newsref/std/cookie.spec.html>.

- [6] REITER, M. K., AND RUBIN, A. D. Crowds: Anonymity for web transactions. *ACM Trans. on Information and System Security* 1, 1 (June 1998).
- [7] The squid web proxy cache. Web site at <http://squid.nlanr.net>.
- [8] VOELKER, G. Personal communication.
- [9] WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., BROWN, M., LANDRAY, T., PINNELL, D., KARLIN, A., AND LEVY, H. Organization-based analysis of web-object sharing and caching. In *Proc. of Second USENIX Symp. on Internet Technologies and Systems* (Oct. 1999), pp. 25–36.
- [10] ZERO KNOWLEDGE SYSTEMS. Freedom 1.0. Web site at <http://www.freedom.net>.