Cookies from the Past: Timing Server-Side Request Processing Code for History Sniffing

ISKANDER SANCHEZ-ROLA, University of Deusto and NortonLifeLock Research Group
DAVIDE BALZAROTTI, EURECOM
IGOR SANTOS, University of Deusto and Mondragon University

Cookies were originally introduced as a way to provide state awareness to websites, and are now one of the backbones of the current web. However, their use is not limited to store the login information or to save the current state of user browsing. In several cases, third-party cookies are deliberately used for web tracking, user analytics, and for online advertisement, with the subsequent privacy loss for the end users.

However, cookies are not the only technique capable of retrieving the users' browsing history. In fact, *history sniffing* techniques are capable of tracking the users' browsing history without relying on any specific code in a third-party website, but only on code executed within the visited site. Many sniffing techniques have been proposed to date, but they usually have several limitations and they are not able to differentiate between multiple possible states within the target application.

We propose BakingTimer, a new history sniffing technique based on timing the execution of server-side request processing code. This method is capable of retrieving partial or complete user browsing history, it does not require any permission, and it can be performed through both first and third-party scripts. We studied the impact of our timing side-channel attack to detect prior visits to websites, and discovered that it was capable of detecting the users' state in more than half of the 10K websites analyzed, which is the largest test performed to date to test this type of techniques. We additionally performed a manual analysis to check the capabilities of the attack to differentiate between three states: never accessed, accessed and logged in. Moreover, we performed a set of stability tests, to verify that our time measurements are robust with respect to changes both in the network RTT and in the servers workload. This extended version additionally includes a comprehensive analysis of existing countermeasures, starting from its evolution/adoption, and finishing with a large-scale experiment to asset the repercussions on the presented technique.

CCS Concepts: • **Security and privacy → Browser security**.

Additional Key Words and Phrases: user privacy; browser cookies; history sniffing

ACM Reference Format:
Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. 2020. Cookies from the Past: Timing Server-Side Request Processing Code for History Sniffing. 1, 1 (August 2020), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The World Wide Web (WWW) is built on top of the HyperText Transfer Protocol (HTTP), a stateless request/response protocol in which each request is executed independently from any other received before. However, most web applications need to keep track of the user's progress from one page to another. For instance, after a user

Authors' addresses: Iskander Sanchez-Rola, University of Deusto, NortonLifeLock Research Group; Davide Balzarotti, EURECOM; Igor Santos, University of Deusto, Mondragon University.

XXXX-XXXX/2020/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

has successfully logged into a service, she expects the application to remember her authentication and allow her to perform other actions in the same website accordingly.

In order to solve this problem, in 1994 Netscape Mosaic introduced the use of Cookies [61]. HTTP cookies are small fragments of data that servers can send to the users inside their responses (by using the `Set-Cookie` header field) or by the use of JavaScript code executed in a webpage in order to create a cookie on the client-side (by invoking the `document.cookie` function). Either way, the browser stores the value of each cookie and includes them in every future request made to the same server. Today, cookies are used for a variety of different purposes, including to maintain the users' login status, to store different options that a user makes while browsing a website (such as the language preference or the acceptance/rejection of a specific consent), or to simply keep track of previous visits from the same user.

The cookies we described so far are called *first-party* cookies, as they are created by the website the user is visiting. However, these are not the only cookies that may be created in the browser. Websites load different types of resources to offer their services and the requests made to retrieve these resources may also trigger the creation of cookies. In many cases, these *third-party* cookies are used to track users among the different websites they visit. For instance, several studies have measured that a large percentage of websites on the Internet perform some form of user tracking [1, 17, 47, 53].

While tracking based on third-party cookies is one of the main techniques different companies use to offer personalized advertisements, other alternatives exist — for instance based on fingerprinting the browser or the user's machine by collecting either hardware or software information [8, 33, 38, 54]. These approaches can completely bypass all the browser protections regarding basic tracking, but their fingerprinting code needs to be executed in all the websites the user visits. Therefore, for websites that are not part of the main tracking networks, there is another option based on so-called *history sniffing*. History sniffing attacks can track the user without relying on any code executed in other third-party websites, just the one executed in the website accessed by the user. While many methods have been proposed in this line of research [7, 20, 32, 36, 69], most of them suffer from several severe restrictions. For instance, previous approaches only provide a coarse-grained classification (e.g., logged-in vs not logged-in), can be easily defeated by users without serious side-effects (e.g., by deleting the browser cache), and were all tested only in a handful of anecdotal cases. As an example, the two most recent works in this area, published in 2015 at the ACM CCS [69] and NDSS [36] conferences, were only evaluated on, respectively, five and eight target websites.

We present a new timing side-channel attack [51], called BakingTimer, that only relies on the presence of first party cookies set by the target websites (which are therefore considered third-party in the context of the attacker page). Our system is based on the analysis of the time spent by the server to process an HTTP request, and by using this information is able to detect both if the user previously visited the website and whether she is currently logged in into it. We then performed a set of experiments to measure how many websites are vulnerable to this attack. First, we checked if our method could detect website accesses, and found that our prototype was able to detect the access in more than half of the websites we analyzed. We tested our solution on over 10K websites belonging to different categories, resulting in the largest evaluation of a history sniffing technique performed to date. Second, we follow a similar approach as previous work, and manually tested the login detection capabilities in a small set.

This extended version also includes a comprehensive analysis of existing countermeasures. We started by analyzing the different possible countermeasures, both at the server- and client-side. After explaining why server-side solutions are not feasible on real-world deployments, we focused our analysis around an existing client-side solution, the `SameSite` cookie attribute. We first present the draft evolution, and explain the details of the different versions of the document. We then discuss the `SameSite` adoption on the three main browser families (i.e., Chrome, Safari, and Firefox), from the beginning of the draft in 2016, until the current year.

We also performed a new set of large-scale experiments to understand how cookies are created, and to which extendt the `SameSite` countermeasure is adopted in the real world. In particular, we visited multiple pages from each of the Top 1M most accessed websites, collecting and analyzing around 138M cookies. We discuss the different fields related to our attack (e.g., if first-party cookies are dynamically or statically created) and, finally, we studied which are the consequences for our history sniffing attack. This allowed us to differentiate the sites that are safe against our technique, from those that are more susceptible to be vulnerable to our timing attack.

2 BACKGROUND

To ease the comprehension of our contribution, in this section we review several aspects regarding current cookie management, as well as different attacks presented in the literature. We then discuss the threat model we consider for our history sniffing attacks.

2.1 Browser Cookies

Cookies were introduced by Lou Montulli [57] while working in Netscape, and are considered the first method to track users on the Web. Cookies allow services to remember a particular user by storing snippets of data on the users' computers, maintaining a concrete browsing state for a returning visitor. After cookies were first presented, they were rapidly embraced by the web community because of their flexibility and the increased usability they enabled in a broad range of websites. As a result, they are now playing a core role as part of the Internet technology [47].

While cookies were not specifically designed to track users across websites, the abuse of their stateful nature started shortly after their first appearance. Since websites are composed of different resources that may be stored in the same domain hosting the page as well as in other third-party servers, these external resource providers have the capability of including cookies along with their provided resource. Therefore, if a third-party server provides resources to a large number of websites, it can certainly gather an accurate picture of a user's browsing history and profile her habits. For example, a website `site.com` includes an image from a third-party domain called `advertisement.com`. The server `advertisement.com` sends the image along with a `HTTP Set-Cookie` header, that will be stored on her machine. When the same user visits another website `site-two.com` that also utilizes images from `advertisement.com`, her browser will send the previously set cookie to `advertisement.com` alongside the request for the image. This allows the advertisement server to recognize the user and collect a list of the websites she regularly visits. This behavior, called *third-party cookies*, is arguably the most widespread technique for web tracking.

However, even the most privacy-invasive third-party cookies are an important part of the web. In fact, the most common usage of this specific type of cookies is web advertisement or analytics. Their web tracking capability is used to track users' browsing history and use this information to build a user profile for custom advertisements. In fact, advertising companies have already stated that web tracking is required for the web economy [58]. However, other techniques exist that can be used for analytics and targeting while preserving users' privacy (e.g., [2, 4, 5, 22, 24, 68]).

However, both third-party and first-party cookies are widely used and browsers even encourage their usage in order to ease usability. For example, when a user of the well-known Chrome browser decides to erase all the stored cookies, the browser warns the user that "*This will sign out of most websites*". Even more important are the settings regarding cookies. Chrome recommends to enable the permission for websites to read and write cookie data, and permits to block third-party cookies. Therefore, the importance of cookies is not only acknowledged by websites, but also by the browsers themselves albeit the privacy issues that may arise.

Even though it is not a common action among the average web users, *third-party cookies* can be blocked (as detailed in Section 2.1). However, browsers do not allow users to remove just these cookies, leaving as only option to manually remove them one by one.

2.2 History Sniffing

A large variety of history sniffing methods exists. We can group these techniques in two categories:

CSS-based attacks. A common trick a page can use to detect if a user has visited other websites (out of a predefined list of targets) is to check the CSS style used to render links. For instance, by checking the CSS `:visited` style of a particular link [15] it is possible to tell if that link had been visited before (typically as it is displayed in a different color). Similarly, it is possible to craft other CSS-based techniques [25, 30, 59, 72] even using filters to exploit the differences of the DOM trees [71] by using a timing side-channel attack.

Timing-based attacks. There is a broad range of history-stealing techniques based on timing information. These approaches were first introduced for this purpose in order to compromise users' private data by measuring the time differences in accessing different third-party content [20], by discovering if it had been cached by the browser. Extracting users' true geolocalization for the same purpose is also possible through web timing, due to the high customization present in current websites. It is also possible to detect if the user is currently logged in into certain websites by timing of specific requests [7], exploiting the AppCache [36], or by estimating the size of certain resources [69]. As explained in Section 3, our own attack belongs to the this timing-based attack category.

Countermeasures and Shortcomings Some of the attacks discussed above are already mitigated by browser vendors. For instance, for the CSS `:visited` style check, all the corresponding browser functions (e.g., `getComputedStyle` and `querySelector`) have been modified to always return that the user has never visited the link [40]. Despite these mitigations, recent work has shown that the attack is still possible using new features available in modern browsers. However, several possible defenses exist to avoid the problem, such as the ones proposed by Smith et al. [59]. In fact, one of these new techniques has already been blocked in Google Chrome [23].

In fact, all existing techniques fall in the classic "arms race" category, in which attacker and researchers constantly discover new tricks that are in turn mitigated by browser vendors, website developers, or even simply careful user settings. Therefore, we decided to investigate if it was possible to devise a new technique that would 1) rely only on server-side information, and 2) that could not be easily prevented without degrading the performance or functionalities of a web application.

2.3 Threat Model

In the timing attack presented in this paper, we adopt the same threat model used by previous work in the area [7, 69]. In particular, we assume an attacker can run JavaScript code on the client browser to perform cross-origin requests. This code can be either loaded directly by the first-party website, or by a third-party service (e.g., by an advertisement or analytics company).

The information collected by our technique allows an attacker to determine which websites were previously visited by the user and on which website the user is currently logged in. There are multiple usages for this data that can result in serious security and privacy implications. The most obvious is related to advertisement, as the usage of the browsing history allows an advertiser to display targeted advertisements. Moreover, an interested tracker could create a predefined list of websites and generate a temporal fingerprint of various users, indicating the user's state in each of them. Even if the fingerprint could not be used as an standalone fingerprinting solution, it will definitely improve the fingerprinting capabilities of other web tracking techniques. Finally, from a security point of view, this information can be used to perform targeted attacks against particular victims.

```php
<?php
$userID = "0bc63ecec05112d03544fde0b5a18c70";

if (isset($_COOKIE[["consent"]) {

    if (isset($_COOKIE[["userID"]) {

        if ($_POST["userID"] == $userID) {
            getUserData(); // Case C
        }

    } else {
        saveNavigation(); // Case B
    }

} else {
    askConsent(); // Case A
}
?>
```

Fig. 1. Example code of a PHP server presenting the three different possible cases of a cookie process schema.

3 BAKINGTIMER

Out of all the requests a web server receives, some contains cookies and some do not. The main observation behind our approach is that, when the browser sends a cookie along with an HTTP request, it is reasonable to assume that the server-side application will check its value, maybe use it to retrieve the associated user session and load additional data from the database, or that it will simply execute a different execution path with respect to a request that does not contain any cookie (in which case, for example, the application may execute the routines necessary to create new cookies).

For instance, Figure 1 shows a simple PHP skeleton that empathizes three different possible cases. First, the program checks if the user already accessed the website before, by testing for the presence of the `consent` cookie using the `isset` function. If the cookie is not found (either because it is the first time the user access the website or because it has been deleted), the program takes the path we named *Case A* that calls the `askConsent` function. Otherwise, the program performs additional checks by looking for some login information stored in the cookies. If userID is not indicated, *Case B* is followed, that calls function `saveNavigation`, else, the userID information is first validated, and then the application follows *Case C* by calling the `getUserData` function.

Figure 2 shows a simplified representation of the control-flow of our toy application, emphasizing the different paths. Our hypothesis is that the time difference among the three cases is sufficient, even across a network connection, to tell the three behaviors apart and therefore to accurately identify whether or not cookies are submitted alongside an HTTP request. In particular, there are two main tests that make detecting a timing difference possible: the first to verify if there are cookies at all and the second to analyze them and load session data. While the comparison themselves are too fast to be identified, the different functions invoked in the three cases may not be. In our toy scenario, these results in the request take x seconds to be processed if no cookies are found, y seconds if they exist but there is not a current active session, and z seconds if the user is currently logged in.

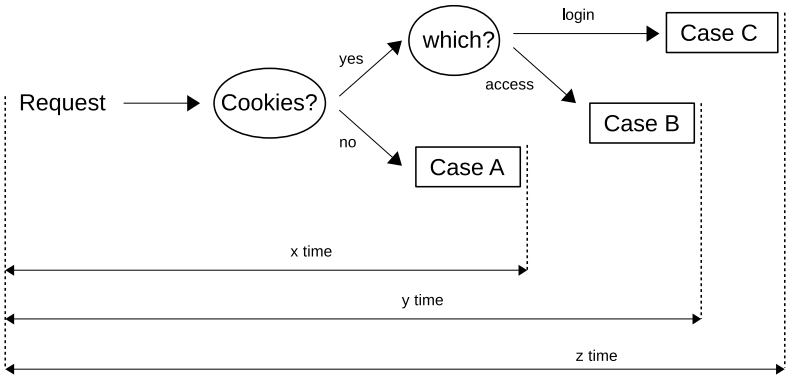


Fig. 2. Server cookie procress schema.

Our prototype tool, called BakingTimer, performs cross-origin requests towards a number of target websites. Each request is performed multiple times, with and without cookies. In fact, while JavaScript code cannot directly test for the existence of cookies belonging to other domains, it can issue HTTP requests in two different ways: (i) by letting the browser send cookies (if any exist for the target domain) or (ii) by forcing the browser to make a request without cookies. So, in one case the code knows that no cookie was present in the request, while in the other cookies may or may not be present depending whether they previously existing in the user browser. The time difference between the two requests, issued multiple times to account for small noise effects, is then used by our tool to detect if the browser already had cookies for that particular target. Even if some cookies can be created when performing these requests, our experiments show that in most cases either the number or type of these cookies differ from what it is set when the user visits the website. We will discuss this aspect in more details in Section 4 and Section 7.

3.1 Retrieval Phase

The first phase of our approach is the retrieval phase, in which the algorithm computes the time required by a server to execute different paths, according to hypothetical cookies sent by the browser. A simplified pseudo-code of the algorithm is presented in Figure 3. Note that the full implementation of the attacks is executed performing different asynchronous/non-blocking requests and using `onreadystatechange` event handlers. Moreover, even if security policies like *Same-Origin Policy* (SOP) or *Cross-Origin Resource Sharing* (CORS) may limit the capabilities of interacting with the response of cross-origin requests, the event handler would still be able to correctly catch when the response arrives to the browser – making our attack possible in these cases.

To measure the round-trip times (RTTs), we use the *User Timing API* [73] implemented in every major browser. The best solution to get a measure of time independent of the current CPU load of the server or the current speed and congestion of the network is to perform a comparison with two different requests sent at the same time. Both time information are obtained side to side, so the workload of the network will likely be roughly the same in the two cases (especially when the experiment is repeated multiple times). To make it more clear, let us consider a simple example. Imagine we are timing the execution of certain function of the server that directly depends on the existence of a specific cookie. Our system would execute a measurement by executing in parallel two requests, with and without the cookies. In a first measurement, when the network workload is low, the system can obtain a value of 30.5ms when cookies are sent and a value of 20.3ms without cookies — thus estimating the execution time of the function in 10.2ms. In a second repetition of the test, when maybe the network load is higher than before, the same experiment can return two different values, such as 45.1ms with cookies and 34.7ms

Input: n the number of comparisons to perform.
Output: cs an array of arrays of numbers representing the server request processing schema: each position are the result of timings with and without cookies.

```
1  Function bakingTimer (n)
2      i ←1;
3      cs ← float[][] of size n×2;
4      while i ≤ n do
5          j ← 1;
6          while j ≤ 2 do
7              startTime ← GetCurrentTime() ;
8              if j % 2 = 0 then
9                  Request(cookies);
10             else
11                 Request();
12             end
13             endTime ← GetCurrentTime();
14             logTime ← endTime − startTime;
15             cs[j][i] ← logTime;
16             j ← j + 1;
17         end
18         i ← i + 1;
19     end
20     return cs;
```

Fig. 3. BakingTimer simplified retrieval pseudo-code (implemented using asynchronous/non-blocking requests and onreadystatechange handlers).

without — leading to a similar (even if not exactly the same) time estimation. In Section 5 we present a number of *stability tests*, designed to verify to which extent our technique is robust against external factors that could tamper the results.

Following the schema presented in Figure 2, if a request requires x *seconds* to be processed (as opposed to y or z seconds), a simple comparison can be used to calculate the actual state of the user in relation with the website. The first request, the one we named as *Case A*, is issued without cookies by using:

```
xmlHttpRequest.withCredentials = false;
```

This guarantees that the server would not receive cookies, even if the browser would have normally sent some for that particular website. Then, our code repeat the request, this time by setting

```
xmlHttpRequest.withCredentials = true;.
```

It is important to remark that the cookies sent in this second request (if any) are completely invisible for our Javascript code. However, the code can still measure the time required to receive an answer, and use this

information to infer the previous relation between the user and the target website: if the cookies were created in a simple access, the server response time will fall in *Case B*, but if some login cookies are stored in the browser the time would be more consistent with *Case C*.

To summarize:

- **Not Accessed:** If the user never accessed the website, and we perform the timings described above, we will obtain a result close to zero. In fact, the first request will take x seconds, because we made the request without cookies. However, also the second request with cookies will take roughly the same amount of time, as no cookie were available in the browser for the target. In this situation we can infer than both calls were the same, indicating no previous access to the website under inspection.
- **Accessed/Logged:** If the user accessed the website in the past or if it is currently logged in into it, the result of the comparison of the time taken by the two request will be different from zero. As we are not using absolute values, but only time differences between consecutive requests, our technique can compensate for differences due to different network delays or workloads.
  A more fine-grained classification to distinguish between previous access and current login is also possible if the attacker has information about the different time required for specific actions. We will investigate this option and provide more details in the comparison phase section.

The algorithm (see Figure 3) takes one parameter n that indicates the number of comparisons to perform. As servers can have specific changes in their workload for many different reasons independent of our tests, we decided to obtain more than one simple comparison. Clearly, the more comparisons the attacker is able (or willing) to perform, the more precise is the model of the server-side computation she can obtain. Nevertheless, there is an obvious trade-off between the time spent in fingerprinting a singe website and the number of websites the attacker wants to test. We will investigate different options and the impact of the number of comparisons on the accuracy of the classification in Section 4.

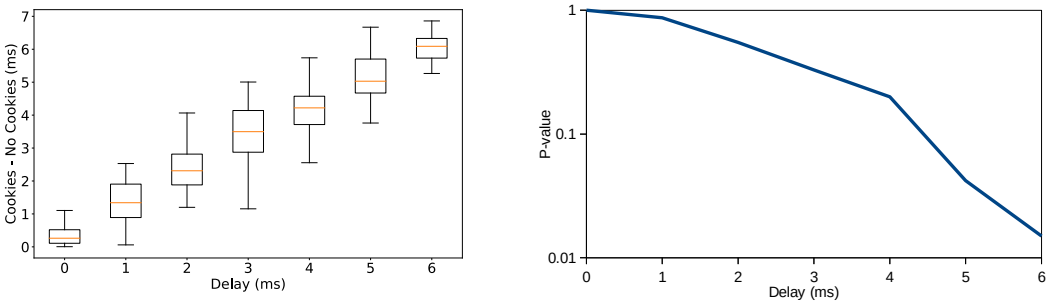
3.2 Comparison Phase

As explained in the previous section, our code performs a number of measurements by timings different HTTP requests for a target website. By subtracting each pair of requests, with and without cookie, we obtain an array of n time deltas.

In order to classify this data, we need to obtain a ground truth dataset for the selected websites. To this end, we need to retrieve the computing time information for each of the cases we are interested to detect. By using this information, we can then statistically test whether a set of measurements belong to one group or another, simply by computing a T-test over the two data samples. A T-test is a two-sided test for the null hypothesis that two samples have identical average values. The result tells us which of the three presented cases does a certain experiment belongs to.

3.3 BakingTimer's Resolution

Before we started our large scale experiments, we wanted to experimentally verify that our hypothesis is correct and that the time difference among different server functionalities can be successfully distinguished over a long-distance network connection. For this reason we implemented a toy service based on `web.py` [62], a simple but powerful Python framework that has been used by famous websites such as Reddit and Yandex. In our example, we controlled the time spent in each path. All HTTP requests sent to the service were issued from a browser located in a different country of where the server was located (Spain and France respectively), to avoid any possible bias introduced by short-distance connections. The average ping time among the two machines was 13.6 milliseconds.



(a) Time deltas at different delays introduced in the server. (b) P-value (T-test) at different delays in the server.

Fig. 4. BakingTimer resolution test.

The server-side application consist in less than 10 lines of code with just a single Python function. In this function, we receive the GET requests, and using the `web.cookies` method, we are able to check if the request includes any cookie, and its value. The service was designed to introduce no delay for requests without cookies, and a configurable delay (ranging from 1 to 6 milliseconds in our tests) to the processing of any request containing cookies. The specific delay was indicated using that same cookie, which was created with JavaScript thought `document.cookie`. We were able to control the introduced delay invoking the function `sleep` from the Python `time` library.

The results of our experiments are summarized in Figure 4. The graphs show that it is possible to detect the time the server spent processing each request quite precisely (see Figure 4a), despite the network load. However, with delays below four or five milliseconds, the difference among the two sets of requests measured by the browser is not statistically significant and therefore an attacker could not conclude whether or not cookies were present in the browser for the target website (see Figure 4b). Instead, if the difference between two paths was equal or above five milliseconds, then even over a long distance connection, it was possible to reliably tell the difference between the two cases. This indicates that it is not necessary for a website to perform complex computations to be vulnerable to our technique and even looking up some data from a database may be sufficient to make it vulnerable to our attack. Obviously, less optimized servers are more prone to incur into larger delays that are easier to observe remotely, while high-end servers may make the path detection more difficult and error prone. We will analyze all these situations in the following section.

4 EXPERIMENTS

In most of the previous studies on history sniffing, the authors limited their experiments to just few selected websites. This poor coverage is mainly due to the fact that existing solutions could generally only distinguish between two states: currently logged in or not logged in. As a result, experiments required the authors to manually create accounts for different services, thus reducing the number of websites that could be tested.

Our technique allows instead to distinguish among multiple states, and in particular to differentiate between websites that have been previously visited by the victim from those that have not. Therefore, on top of performing a login detection case study (detailed in Section 6), we also conducted a large scale experiment to measure the percentage of different websites that are vulnerable to our attack.

The dataset used in our tests consists of two different groups: (i) highly accessed websites and (ii) websites related to sensitive information that users may want to keep private. For the first group we selected websites

from the Alexa [3] Top5K list, to verify whether extremely popular services are also vulnerable to our timing side-channel attack. The second group is composed instead by websites that can be used to detect private personal information of the user, such as medical or religious websites. We used categories defined as sensitive in various data protection laws [18, 19, 34]. Since many of the entries in this group are not included in the Alexa Top1M, we could also check if not highly accessed websites are more or less vulnerable than highly accessed ones.

4.1 Domain Selection

We first populated our list of personal information websites by performing various queries obtained through the auto-complete option offered by different search engines (e.g., *"cancer treatment side effects"*). We made five different queries for the following six categories: medical, legal, financial, sexual identity, political, and religion. Our tool issues each query on four search engines (Google, Bing, Yandex, and DuckDuckGO) and retrieved the top 100 results from each of them.

To avoid an overlapping between the two lists, we removed from this group domains that also belonged to the Alexa Top10K list. We also removed any website that appeared in multiple categories, as we wanted to focus only on specific areas in isolation. Finally, we obtained a set of 5,243 unique personal information websites. In order to balance the two groups in the dataset, we selected the same number of websites from the Alexa top list. The combination of the two groups resulted in a final dataset of 10,486 websites.

4.2 Methodology

We implemented our BakingTimer proof-of-concept by using a custom crawler based on the well-known web browser Chrome, without using any flag that may influence the network connectivity in any way. We can perform multiple actions using the Chrome debugging protocol (with the `remote-debugging-port` parameter), which allows developers to control the browser [9]. For instance, we can access different websites using the `navigate` command, or run JavaScript code on the current website once loaded with the `loadEventFired` event and the `evaluate` command. The executed code uses `xmlHttpRequest` to perform the corresponding requests of the presented technique. Even if it would have been possible to implement the experiments by using a simple Python script, we decided to rely on a real-world browser to obtain the most realistic results possible. Our crawler follows two different phases in order to obtain the data that we will later used to check if a target server is actually vulnerable to the attack.

- **Never Visited:** First, the tool cleans every previous access information stored in the browser. Then, it starts making the different requests described in Section 3 (i.e., both with and without cookies, which in this case are none) from a third-party website (blank tab). This data will be later used as a baseline for requests performed when no previous access was done.
- **Previously Visited** In this case, the browser first accesses the website under inspection. No cleaning process is performed, so all cookies automatically created by the website are saved in the browser and sent to the server in the following requests. After that, it goes to a third-party website (blank tab) and starts making the different requests described in Section 3 to that same website under inspection.

Once all the data was retrieved, we performed the statistical tests described in Section 3 in order to identify whether the time information in the two groups of requests are statistically different or not. We also repeated the experiment with different number of requests in order to check their influence on the final result. To be practical, we tested with a minimum of 10 and a maximum of 50 comparisons (therefore ranging from 20 to 100 HTTP request per target website). The higher the number the more stable is the measurement, but so is the amount of time required to run the test. Therefore, the choice boils down to a trade-off between precision and scalability. We believe that if the attacker is only interested in a handful of websites, then it would be possible to perform even more than 50 comparisons.

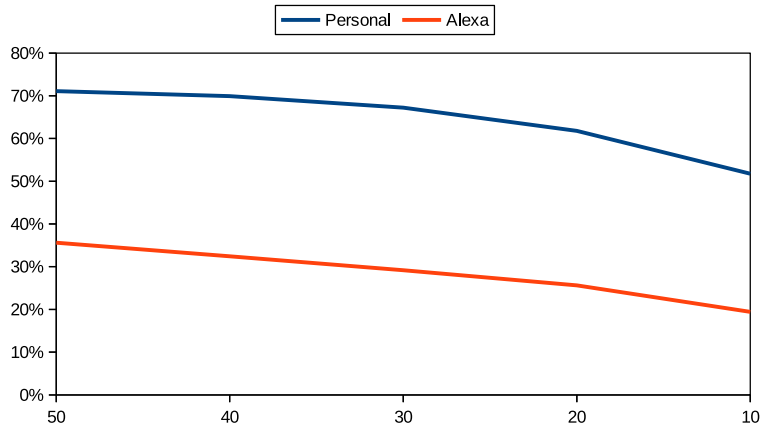


Fig. 5. Percentage of vulnerable websites depending in the number of comparisons performed.

It is also important to note that since we need to repeat the requests multiple times, it is possible that the first request "pollutes" the browser by setting cookies that are later sent in the following tests. In other words, if i) the target website sets cookies on cross-origin requests, and ii) those cookies are the same (in number and nature) of those set when the website is loaded in the browser, then our timing attack would not work.

However, this does not seem to be very common, as more than half of the 10,486 websites we tested are vulnerable to our technique. The actual percentage varies between 40%, when the minimum number of comparisons is performed, and 53.34% if our attack performs 50 comparisons. Figure 5 shows the success rate at different numbers of comparison for the two groups separately.

4.3 Highly Popular Websites

This group includes 5,243 websites from the Alexa top websites list. As websites in these categories are visited by a large number of users, most of their servers and the code they run are likely more optimized, thus making more difficult to measure the timing side channel. This is confirmed by the fact that our attack worked on 35.61% of the websites in this category. This result is still quite severe, as it means than a webpage could still reliably identify if its users had previously visited more than one third of the most popular pages on the Internet.

In order to get a deeper analysis of the obtained results, we clustered the websites in different categories and we computed the percentage of each of them that were vulnerable to our attacks. To determine the category, we used three services: Cloudacl [14], Blocksi [6], and Fortiguard [21]. Their category names are similar, and, after a normalization process, we settled for 78 different category names. Table 1 shows that the top 6 categories vulnerable to the attacks include around 40% of the websites, with a peak of 43.75% in the case of sport-related websites in the Alexa top list.

4.4 Privacy-Sensitive Websites

This group includes 5,243 websites from six different categories related to private personal information — i.e., medical, legal, financial, sexual identity, political, and religion. The results from these websites allows us to understand two different aspects. First, we can verify the amount of websites directly related to sensitive information that are vulnerable to our attack. Second, it gives us an opportunity to test less popular websites (as

Table 1. Websites vulnerable to our attack (top six private-sensitive on the top and top six highly popular on the bottom).

Category	% Vulnerable
Medical	72.63
Religion	71.66
Financial	71.63
Political	70.73
Sexual Identity	70.38
Legal	69.39
Sports	43.75
Search/Portal	42.25
Government	40.63
Travel	40.40
Gaming	39.39
Adult/Pornography	39.06

85% of the vulnerable sites in this category are ranked below the Alexa Top500K) to observe whether smaller server infrastructures can result in a higher accuracy of our technique.

The results of our experiments show that a stunning 71.07% of all the analyzed websites in this group are vulnerable to the BakingTimer attack. If we break down the result by category, we see that all have similar percentages and there is no clear difference between them (see Table 1). This result is much higher than the one obtained in the top Alexa group, but the difference is not due to the number of cookies. In fact, we compared the mean and standard deviation of the number of cookies in privacy-sensitive websites and highly popular websites. Unsurprisingly, the results show that highly accessed websites have a higher mean number of cookies (9.03±7.87) compared to the number of cookies in private personal information websites (5.83±5.49). This means that the main reason behind the difference is not linked to the number of cookies, but more likely to the slower servers or the less optimized code responsible to process the incoming requests.

5 STABILITY TEST

Our attack relies on a time-based side channel used to distinguish among different execution paths on the server-side application code. The fact that BakingTimer does not look at absolute times but at the difference between two consecutive requests minimizes the effects of network delays on our computation. However, even if the time difference between the two request is minimal, it is possible that small non-deterministic variations such as the jitter, bandwidth, or network congestion and routing can introduce a certain level of noise in the measurement. To account for these small fluctuations, BakingTimer needs to repeat each test multiple times. As shown in Section 4, ten comparisons are sufficient to detect nearly 40% of all the analyzed websites, which increases to over 50% if we perform 50 comparisons.

In order to obtain a clear view of the specific effect the network or the server load can have in our measurements, we decided to perform a dedicated stability test. For this experiment we randomly picked 25 website detected as not vulnerable and 25 websites detected as vulnerable to our attack. Following the same approach presented in the general experiment, we now repeated our test every hour for a period of 24h for each website. Moreover, to be more realistic, we computed the ground truth on one day, and performed the attacks on the following day. This resulted in a total of 48 checks per websites, and 2,400 tests globally.

From the group of the 25 websites not vulnerable to our attack, all our tests returned the same result (which confirmed the absence of the side channel). This results proves the stability of the presented method from a

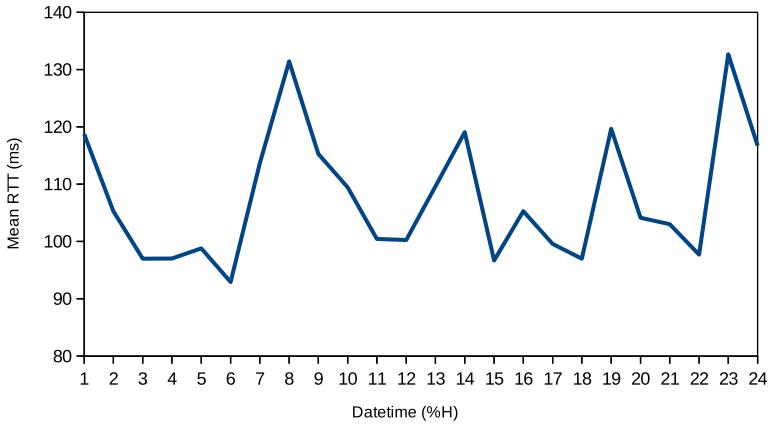


Fig. 6. Mean RTT of one website never visited before, during a full day (with data every hour).

network perspective. More concretely, regarding fluctuations, Figure 6 shows, for one of the websites analyzed, the different mean RTTs we registered each hour. Even if there were considerable fluctuations on the network speed during the day, we were still able to perform the correct classification.

From the group of the 25 vulnerable websites, we were able to correctly identify when each website was previously visited – in all our tests. Instead, in the case in which we did not previously visited the websites, there was one case in which (at 9 and 10 pm), we incorrectly identified a single website as visited. Nevertheless, in total from the 2,400 checks performed in this experiment, we only incurred in two false positives and no false negatives, indicating the high reliability of the presented history sniffing technique.

6 LOGIN DETECTION

In this section we look at performing a more fine-grained classification of the user history, not just by telling if a victim has visited a target website in the past, but also to distinguish a simple visit from owning an account or being currently logged in into the service.

Since this analysis requires a considerable manual effort to set up the accounts and operate the websites, we will limit our study to few examples taken from the two groups in our dataset. This approach is similar to the one regularly adopted by other related work on history stealing techniques [36, 69]. It is important to remark that we did no test websites that required a registration fee or that had a complex registration procedures (e.g., requiring a phone number verification or a bank account).

This cases, a third-party website can check if the user ever accessed any of the websites under attack, and can even check if the user is logged in. This type of information could be extremely useful for a malicious attacker. For instance, it could be used to perform targeted phishing attacks against users, to steal the login credentials of those affected websites. Moreover, it will also be beneficial for other types of attacks like Cross-Site Request Forgery (CSRF), in which the attacker could use the state of the user in a website vulnerable to this type of attacks to perform privileged actions on the users accounts, such as password changes or information retrieval [12, 48]. Actually, the attacker does not need to control the third-party website, just have his code executed on them, for example via an advertisement (as explained in Section 2).

6.1 Highly Accessed Websites

From this category we selected two popular websites, related to gaming and clothing, that were detected to be vulnerable to our BakingTimer attack (Section 4), and that have been the target of different phishing attacks — in one case for the huge number of users and in the other case for the high economic value of their customers. More concretely, World of Warcraft (WoW) and Gucci.

In both cases, after the user logs in, a number of cookies are created in the user's browser to store this new status (e.g., `wow-session` and `gucci-cart` respectively). The presence of these cookies make the server take a different execution path, resulting in a different computation time. We followed the same analysis principles as the ones used when we analyzed the websites in Section 4, and detected both websites fall in our simplified three-paths model presented in Figure 2. The results show than each of the different states (e.g., not accessed, accessed, and logged in), does not match any of the other two states when performing the statistical test of the comparison phase (see Section 3).

6.2 Private Personal Information Websites

Detecting if a user has previously visited a website linked to specific information such as religion or sexual identity can leak some private information to third-parties the user may not even be aware of. However, if the third party could actually detect that the user is currently logged in into one of those websites, the link between the user and the website becomes much stronger.

From this category we picked a religious website (Dynamic Catholic) and a sexual related chat/forum (LGBTchat.net). Again, the presence of login cookies (i.e., `xf_user` and `frontend_cid`) made the two applications take different execution paths, whose computation time was sufficiently different to be reliably fingerprinted by our solution.

6.3 Persistent Login Information

In all previous cases, when the user logs out from the website, the different cookies related to the login status are deleted by the server. In this situation, a third-party website would still be able to detect that the user has visited the website in the past, but it would not be able to distinguish if she had an account and she ever logged into the service.

While this may seem obvious, there are also websites for which it is not true. In fact, some sites do not properly (or at least not completely) delete all cookies they created in relation to the login process. This could be either because the cookie deleting process was not performed correctly, or because the website developers explicitly decided to maintain part of the login information stored in the cookies even when the user is not logged in. However, the presence of these cookies can be sufficient to trigger a different code execution in the application that can be detected by our technique. This allows third-party to be able to differentiate users that just accessed the websites from those who own an account, even if they are not logged in at the time the test is performed.

For instance both Microsoft/MSN and Openload fall into this category because of the presence of, respectively, a `MUID` cookie and a cookie with a `MD5` hash as name. In both cases, when the user logs out of the service, some cookies are deleted but some other are maintained in the browser. In this two specific cases, we first classified the websites following the same three-state schema as in previous cases. Then, we logged out of the service and checked if this state would be classified as logged or accessed. Our results show that in both cases, the comparison phases classified this state as logged. This indicates that even if the user logged out, if the websites does not correctly delete all related cookies, it would be possible to detect a previous logged state.

Table 2. A comparison of current state-of-the-art timing methods for history sniffing.

Approach	Type	Login Status	Difficult Clean	Previous Access	Websites Analyzed
Timing Attacks on Web Privacy [20]	Web & DNS Caching	✗	✗	✓	<10
Exposing private information... [7]	Server Boolean Values	✓	✓	✗	<10
Cross-origin Pixel Stealing [32]	Cross-origin DOM Structure	✓	✓	✗	<10
Identifying Cross-origin... [36]	HTML5 Application Cache	✓	✗	✗	<10
The Clock is Still Ticking [69]	Cross-origin Resource Size	✓	✓	✗	<10
Our Approach (BakingTimer)	Cookie-based Request Timing	✓	✓	✓	10,486

7 DISCUSSION

Even if a user trusts all the websites she visits, many websites include a large number of third-party services and resources to improve their usage or to monetize their traffic. All these scripts loaded on the website, including simple advertisement banners, can track users and perform many different attacks, including the one we presented in this paper.

7.1 Result Interpretation

In order to obtain the most reliable results, it is important to perform the experiments against multiple real-world websites. In fact, synthetic websites or small sample sets may not correctly capture all the implementation patterns encountered in the wild. Our tests show that more than half of the websites we analyzed are vulnerable to our attack. This means two important things. First, that there is a measurable, statistically significant difference between the time the server spend processing the two classes of requests (with, and without the access cookies). Second, that either the website does not set cookies on cross-origin requests, or that those cookies are different from the one created on a regular access.

7.2 Comparison with Similar Techniques

Table 2 shows the different state-of-the-art timing methods and their different characteristics, both in terms of adopted technique and on the type and size of the experiments performed to validate.

The majority of previous works allowed an attacker to detect the login status of a victim in other websites. Only one [20], apart form ours, allows also to detect the access state (but this same technique is unable to detect the login status). The only technique able to detect both access and login state is the one presented in this paper.

Most of existing attacks, including our own, do not rely on any other browser resource than the cookies. This makes the technique resilient to generic browsing history cleaning processes, as browsers explicitly discourages user to delete cookies in their settings (see Section 2). Two techniques are instead based on different types of browser caching that, on the contrary of cookies, are even deleted by default, and therefore users can easily and without any major consequence delete them when needed.

Regarding the number of websites analyzed, as nearly all the techniques are only able to detect the login status, the manual effort needed to perform a big scale analysis made large scale experiments unfeasible. We also presented a similar set of experiments in Section 6, but we also performed an automatic analysis of over 10k websites divided in different categories to provide a general overview of how effective this attack can be.

8 COUNTERMEASURES ANALYSIS

As the problem happens in the servers' request processing logic, the most reasonable solutions may seem those that involve modifications on how the servers respond to requests. For example, including a random delay in the

response time. However, this would just increase the noise and a larger number of comparisons may compensate for small random changes in the processing time. Another possibility could be changing the web application to have fixed response times for sensitive requests. On top of being very difficult to be properly implemented, this solution would also have a large performance impact — considerably reducing the number of requests per second a web site can sustain. Moreover, as the fixed time would need to be exactly the same for all the sensitive request, they should be as slow as the slowest response the server can make. For all these reasons, we believe none of these server-side mitigations are practical and feasible to implement in a real-world deployment.

Nevertheless, there is another possible approach to counteract our attack through a client-side solution. RFC6265 is an Internet Engineering Task Force (IETF) document [29] that defines the corresponding HTTP Cookie and Set-Cookie header fields (e.g., `HttpOnly` and `Secure`). In January 2016, an Internet-Draft was published to update it, with the definition of an opt-in feature indicated though the new attribute called `SameSite`. The main reason behind this addition, is to allow to control when the created cookie should be sent in cross-site requests. This solution could allow to mitigate cross-origin information leakages (as the one presented in this paper) and provide some protection against cross-site request forgery (CSRF) attacks.

8.1 Draft Evolution

The first version referencing this technique [26] just indicated that if detected (case-insensitive match), the browser must append an attribute called `samesite-flag` to the newly created cookie with an empty value. The following version [27], included two possible values: `Strict`, and `Lax`. If the value is set to Lax, it indicates that the cookie should only be sent on requests within the same site, or through top-level navigation to that specific site from other different sites. A value of Strict limits the cookie inclusion on request only to those originated from the same site. If none of those two options is indicated (i.e., the field is either malformed or left empty), it would be enforced as Strict. It is important to note that for instance, if *www.example.com* requests an image from *static.example.com*, it would be considered like a same-site request. Browsers should use an up-to-date public suffix list (such as the one maintained by Mozilla [39]) to detect these situations. A new version of the draft was published in April 2016 [28], which includes one main modification to the previous one: if the SameSite attribute is included, but no case-sensitive match is found for Strict or Lax, the browser should ignore it. This differs from the preceding version, which suggested to use a Strict enforcement on this same situation. Additionally, the revised version included the value `None`, which should only be set in cases where the SameSite attribute was not indicated (defined as "monkey-patching" in the draft).

In May 2019 a completely new Internet-Draft was published titled Incrementally Better Cookies [45]. In this document, there are two main changes inspired by the HTTP State Tokens draft [44]: (i) cookies are treated as Lax by default (breaking backwards-compatibility), and (ii) cookies that require cross-site delivery can explicitly opt-into that behavior indicating the None value in the SameSite attribute when created. There is an additional important point in the second case, those cookies need the Secure attribute, indicating that they must be transmitted over HTTPS. The last version [46], published in March 2020, included a temporal option, where unsafe top-level requests (e.g., `POST`) can include recent cookies (e.g., created less than 2 minutes before) that did not indicate the attribute.

8.2 Browser Adoption

The SameSite attribute was implemented in all three major browser families, but each family followed a different strategy in the process. We will explain how each of them adopted this countermeasure and the market share of each of them (based on the data of March 2020 of Stetic [60]), in order to asses the impact.

- **Google Chrome** (49.82% of the market): Started implementing the SameSite attribute in version 51 [10], published mid 2016, following the draft published in April of that same year [28]. In this case, only

cookies with valid values would be used, and no default enforcement was implemented. This approach was adopted until version 79, all together accounting for 13.49% of the total web users (27.08% of the Google Chrome share). The more recent behaviors presented in the last draft [46] (Lax by default), started to be implemented in version 80 [65]. However, they are currently only deployed for a limited population, in order to monitor and evaluate the impact on the ecosystem, with the plan of gradually increasing the distribution in subsequent rollouts.
Nowadays, Google indicates that it is still targeting an overall limited population of users with the last Chrome version. Taking into account that 35.33% of the web users browse with this version, and that only a small percentage of them implement have it implemented, we can expect that in total less than 2% of the total web user are running this restricted policy in their browsers. It is interesting to note that this more restrictive SameSite behavior will not be enforced on Android WebView or Chrome for iOS for now, reducing even more the number of users that run this functionality. Moreover, Chrome has implemented new policies to avoid possible problems regarding this change in the cookies behavior (e.g., `LegacySameSiteCookieBehaviorEnabled`), which will allow browsers with the restrictive behavior implemented to revert to the legacy SameSite behavior [66].
Due to the extraordinary global circumstances (COVID-19), at the time of writing Chrome is temporarily rolling back the enforcement of this SameSite restrictions, starting from April 2020 to the end of the year [11]. According to Google, this is done because *"we want to ensure stability for websites providing essential services including banking, online groceries, government services and healthcare that facilitate our daily life during this time"*.

- **Apple Safari** (30.97% of the market): Apple implemented the SameSite attribute at the end of 2018 in iOS 12 and macOS 10.14 (Mojave) [67]. The developers followed the second version of the draft about Samesite [27], indicating that cookies will be enforced to Strict by default. At the time of its implementation, a newer version of the draft already existed [28], where there is not default enforcement for cookies, only the general Strict or Lax. This was reported as a bug [70], and fixed in Safari 13 (17.33% of the browser share). However, the Safari development team indicated that they are not planning to backport this fix in older iOS or macOS versions, and recommend developers to check the user-agent value before creating the cookie, to act accordingly.

- **Mozilla Firefox** (15.28% of the market): Mozilla implemented the SameSite attribute in mid 2018, in Firefox 60 [43]. They followed the last version of the original draft [28], cookies do not have a default enforcement and there are the Strict and Lax options if opted in. However the developers are currently making some tests in Nightly [41], with the first version of the final draft [45], where Lax is the default. However, they do not plan to include it in the stable version (*"We are not planning to ship this feature yet. We are testing in nightly only to see the level of breakage."*). Moreover, Mozilla is running additional tests with the last version [42], where the developers included the option of unsafe top-level requests when cookies are newly created (<2minutes).

To summarize, most of the browser are implementing the SameSite attribute without any default enforcement, and just a small percentage of the global web population (maybe 1 to 3%), will have access to the restricted version in a near future.

8.3 Experimental Study

Now that we better understand the adoption of the SameSite attribute on the browser side, we wanted to also measure to which extent website developers implemented the SameSite attribute on the server side. Moreover, we are interested in analyzing how this new feature could affect the attack presented in this paper. For this

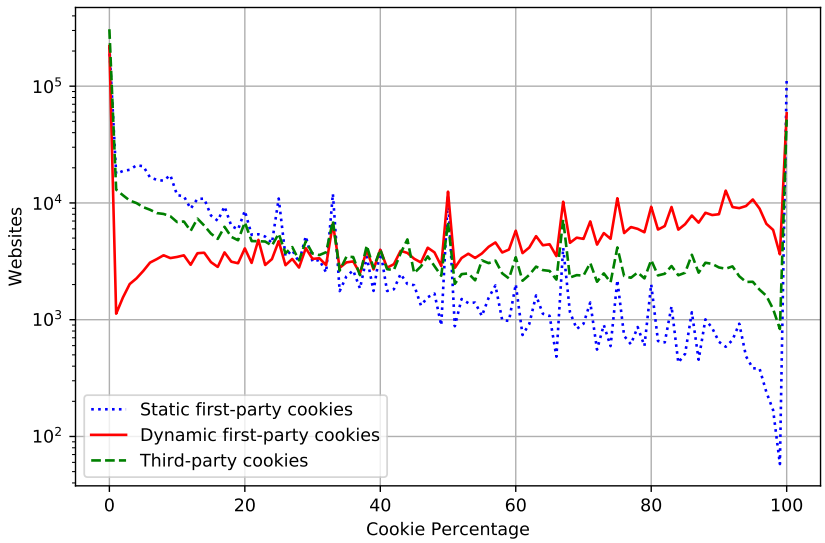


Fig. 7. Percentage of each type of cookies in websites (log scale).

purpose we used the Tranco 1M most accessed domains list [35], which is a combined ranking of Alexa [3], Cisco Umbrella [13], Majestic [37] and Quantcast [49], to offer better confidence and stability.

We used a custom crawler based on the open-source web browser Chromium, in order to access the corresponding websites, and we collect all the information regarding cookies (e.g., headers and creator) using a custom instrumentation developed using the Chrome debugging protocol (CDP) [9]. Our extension loads the home page of the domains, and then recursively visits three random pages from each website, up to a distance of three starting from the home page (up to 13 pages per website), following the approach presented in previous studies [50]. In order to avoid the detection of our crawler, we also implemented the most recent methods proposed to mimic real users [16, 55, 56].

8.4 General Results

Out of the 1M websites we visited, 76% created cookies. In total, we detected the creation of 138M cookies. However, the attack presented in this paper, does not use all the different cookies that are created in an access, but only those first-party cookies that were dynamically generated (using JavaScript or the HTTP `Set-Cookie` header). For example, if all cookies created on the website are first-party cookie generated statically (i.e., in the main page request), the attack would not work, as those cookies would be polluted in subsequent request performed to detect the users' state (as previously indicated in Section 4).

Therefore, we started by checking what is the percentage of cookies that are interesting for our history sniffing attack, and found that 56% of all the cookie creation we detected were dynamic first-party cookies, 37% were third-party cookies, and only 7% were actually static first-party cookies. This shows that the cookies required by our attack represent, by far, the biggest group of cookie type found on websites nowadays.

Even if indicating percentages of global numbers can help to comprehend the scale of the phenomenon, those percentages could be misleading if only a small group of websites generate many cookies of one specific type.

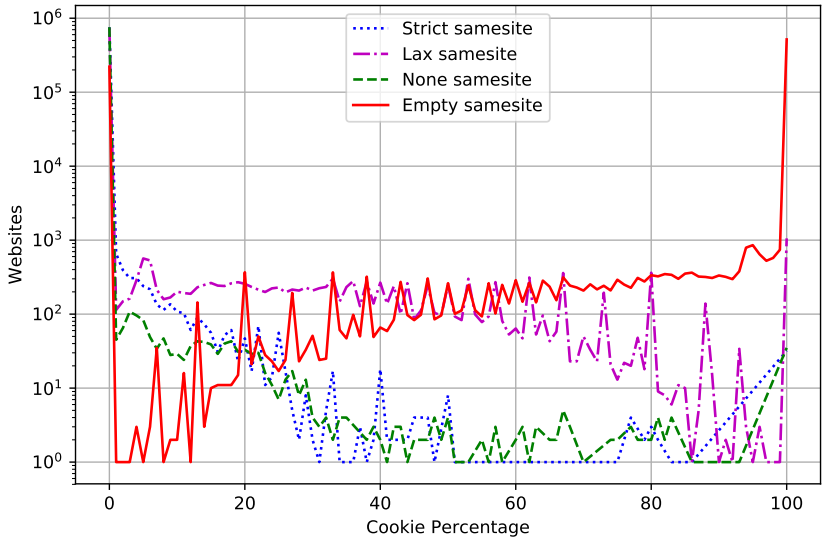


Fig. 8. Percentage of each type of samesite in dynamic first-party cookies (log scale).

In order to give a more accurate view of the current state, we performed an additional analysis in which we calculated the percentage of each of those cookies types per website, and then we represented the occurrence of each of them. Figure 7 shows three curves, representing respectively static first-party, dynamic third-party, and third party cookies. The Y axis shows the number of websites, while the X axis shows the percentage of each type of cookies. So, for instance, we see that around 10K sites have 90% of their cookies belonging to the dynamic, first-party category—but less than 1K have 90% of static, first-party cookies. From the figure we can draw two conclusions. First, that static cookies are a minority in most of the websites (left part of the graph). Second, that dynamic, first-party cookies are a vast majority in many websites (right side of the graph). This indicates that the cookies used for our attack exist in most websites, and that they represent the largest percentage of all cookie on each of them.

After detecting that dynamically created first-party cookies are the most common type of cookie, we wanted to analyze how those cookies are protected against the history sniffing method presented in this paper. In total, around 99% of the cookies did not indicate any type of SameSite attribute, but as we indicated before, this may misrepresent reality, because some small number of website may create a disproportionate amount of cookies. By following the same approach described above, we calculated the percentage of dynamic first-party cookies with the different possible SameSite attribute values, and reported the occurrence of of each of them in Figure 8 (log scale). The figure shows that `Strict` and `None` values are not very common, indicating that developers do not normally implement neither the extreme restrictive nor the open options. The two most common values we found are `Empty` and `Lax`, which respectively provide a milder form of protection and no explicitly selected choice. However, the distribution resembles the one seen in Figure 7 regarding cookie type:

Lax is generally more common is low percentage cases (<30%), but Empty is the prominent choice for a very large amount of websites. Again, this clearly indicates that dynamic first-party cookies with empty SameSite attributes are present in most websites, and that they represent the largest percentage of all cookie on them.

Table 3. Top and bottom six categories susceptible to be vulnerable (over 10k websites).

Category	Websites	Susceptible	Improbable	Safe
Shopping	82,960	83%	3%	13%
News/Media	44,251	81%	5%	14%
Vehicles	30,005	80%	4%	16%
Forums	28,602	79%	3%	18%
Restaurants/Food	24,081	79%	4%	17%
Real Estate	11,549	79%	4%	18%
Games	23,404	65%	6%	30%
Legal	32,948	62%	11%	27%
Personal Sites	11,421	62%	7%	31%
Malicious Sources	13,035	58%	5%	37%
Suspicious	42,580	50%	6%	44%
Web Ads/Analytics	26,089	24%	1%	74%

If we now take into account the browser adoption for the SameSite attribute, we can clearly see that today the protection offered by the SameSite attribute against our attacks is very small. Only a very small percentage of websites are protecting their cookies (Strict or Lax), and a very small percentage of the total web user population, as calculated in the previous subsection, is using (or will be using in the near future) a browser that implements restrictive enforcements (e.g., Lax by default) [46].

8.5 Attack Susceptibility

We wanted to go one step further, so we calculated which is the percentage of websites that, due to the little implementation of the SameSite attribute described in the previous subsection, are susceptible to be vulnerable to our technique. In order to be conservative, we will separate those websites that have unprotected identifier cookies from those that do not, assuming that is more likely that a cookie that stores an identifier can trigger a different path on the server request processing code. To identify possible identifiers we pre-filter cookies by using `zxcvbn` method proposed in a recent GDPR related work [52]. To be as precise as possible, we divided website in three different groups:

- **Susceptible**: Created dynamic first-party cookies that contain possible identifiers, without using the SameSite attribute, or with `None` value.
- **Improbable**: Similar to susceptible, but in this case the cookies did not contain a likely identifier value.
- **Safe**: Did not create dynamic first-party cookies that could make the site vulnerable to our attack.

Overall, 65% of all the websites with cookies are susceptible of being vulnerable, 7% were improbable to be vulnerable, and only 29% were certainly safe. It is important to note that this numbers do not indicate that those website are in fact vulnerable, as their server-side code may not depend on certain cookies to trigger different code paths. However, we can say that those websites can be vulnerable based on the adoption of the countermeasures currently offered.

Finally, we wanted to check if there are differences between website categories (classified using website-to-category data from a commercial engine [63, 64]). We performed this same classification based on that data, and reported the results of the top and bottom six categories in Table 3. Results hints that website with high user interaction rate (such as shopping, news, and forums) are more prone to be vulnerable, while security/privacy dubious website belonging to the malicious, suspicious, and analytics categories, are less prone. One of the possible

reasons for this is that highly interactive website create multiple dynamic first-party cookies to communicate with the user, and the second group of websites rely more often on third-party cookies.

This analysis demonstrates that in order to be completely protected from the technique presented in this paper, all cookies must set the SameSite attribute with the Strict or Lax value. As long as one of the cookies involved does not specify this option, the attack would still work. Due to the sensitive nature of login cookies, they could be more prone to use this option. Nevertheless, its important to remark that some sites could lose part of their core functionalities as a result of using Strict or Lax SameSite cookies. In fact, many types of websites, such as social networks or cashback services, rely on cookies to be added in third-party requests, and therefore the global applicability of this countermeasure could be limited in several website categories.

9 RELATED WORK

History sniffing attacks are a widely explored topic with different techniques and solutions presented over the years. Clover [15] found that it was possible to identify previously visited websites just checking the CSS `:visited` style of a specially crafted link though the `getComputedStyle` method in JavaScript. Many other similar attacks appeared using different CSS-based techniques [25, 30, 59, 72]. Kotcher et al. [32] discovered that besides from the above mentioned attacks, the usage of CSS filters allows the involuntary revelation of sensitive data, such as text tokens, exploiting time differences to render various DOM trees. Weinber et al. [71] followed another direction, using interactive techniques to get the information. While these attacks are much slower, the protection methods are in principle more difficult to implement.

With a different approach, and leaving CSS aside, Felten and Schneider [20] introduced web timing attacks as a tool to compromise users private data and, specifically, their web-browsing history. Particularly, they proposed a method based on leveraging the different forms of web browser cache to obtain user specific browsing information. By measuring the time needed to access certain data from a third-party website, the attacker could determine if that specific data was cached or not, indicating a previous access. Some years later, Jia et al. [31] analyzed the possibility of identifying the geo-location of a given visitor using to the customization of services performed by websites. As this location-sensitive content is also cached, it is possible to determine the location by checking this concrete data and without relying in any other technique.

Bortz et al. [7] organized JavaScript web timing attacks in two different types of attacks: (i) direct timing, based on measuring the difference in time of diverse HTTP requests and (ii) cross-site timing, that allows to retrieve private client-side data. The first type could expose data that may be used to prove the validity of specific user information in certain secure website, such as the username. The second attack type follows the same line of previous work by Felten and Schneider. They also performed some experiments that suggested that these timing vulnerabilities were more common than initially expected.

Two recent studies show that these attacks are far from being solved. Van Goethem et al. [69] proposed new timing techniques based on estimating the size of cross-origin resources. Since the measurement starts after the resources are downloaded, it does not suffer from unfavorable network conditions. The study also shows that these attacks could be used in various platforms, increasing the attack surface and the number of potential victims. The specific size of the resource can leak the current state of the user in the website. Lee et al. [36] demonstrated that using HTML5's AppCache functionality (to enable offline access), an attacker can correctly identify the status of a target URL. This information can later be used to check if a user is logged or not in certain website.

However, these timing techniques can generally only determine if the user is logged on a specific website or some isolated data, but not if she has just previously accessed it. Moreover, some of them use resources easily cleanable by the user, like different cache options, as they do not imply any visible consequence to the user.

10 CONCLUSIONS

Many different threats against the users security and privacy can benefit from a list of websites previously accessed by the user and a list of services where the user is logged in or ever logged in.

We showed that simply using cookies of third-party websites, is possible the detect the specific state (e.g., accessed and logged) of a user in certain website, which outperforms previous techniques that are only able to detect one single state. In particular, we present a novel timing side-channel attack against server-side request processing schema. This technique is capable of detecting execution paths with more than 5 milliseconds of difference between each other.

We also analyzed real-world servers to detect the percentage of websites vulnerable to the presented attack. All previous work analyzed less than 10 websites (manually), as they generally only detect the logged status. We performed this same analysis, and additionally, we performed an automated check of 10k websites from different categories and number of users. Results show that more than half of the websites are vulnerable to our technique. In this extended version, we also measured to which extent website developers implemented countermeasures in the Top 1M websites, and found that 65% of the websites with cookies are susceptible of being vulnerable.

ACKNOWLEDGMENTS

This work is partially supported by the Basque Government under a pre-doctoral grant given to Iskander Sanchez-Rola.

REFERENCES

[1] Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., and Preneel, B. FPDetective: dusting the web for fingerprinters. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2013).

[2] Akkus, I. E., Chen, R., Hardt, M., Francis, P., and Gehrke, J. Non-tracking web analytics. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CSS)* (2012).

[3] Amazon Web Services. Alexa top sites. https://aws.amazon.com/es/alexa-top-sites/, 2018.

[4] Backes, M., Kate, A., Maffei, M., and Pecina, K. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2012).

[5] Bilenko, M., Richardson, M., and Tsai, J. Targeted, not tracked: Client-side solutions for privacy-friendly behavioral advertising. In *Proceedings of the Privacy Enhancing Technologies (PETS)* (2011).

[6] Blocksi. Web content filtering. http://www.blocksi.net/, 2018.

[7] Bortz, A., and Boneh, D. Exposing private information by timing web applications. In *Proceedings of the International conference on World Wide Web (WWW)* (2007).

[8] Cao, Y., Li, S., and Wijmans, E. (Cross-)browser fingerprinting via os and hardware level features. In *Proceedings of the Network and Distributed System Symposium (NDSS)* (2017).

[9] ChromeDevTools. DevTools Protocol API. https://github.com/ChromeDevTools/debugger-protocol-viewer, 2019.

[10] Chromium Blog. Chrome 51 Beta: Credential Management API and reducing the overhead of offscreen rendering. https://blog.chromium.org/2016/04/chrome-51-beta-credential-management.html, 2016.

[11] Chromium Blog. Temporarily rolling back SameSite Cookie Changes. https://blog.chromium.org/2020/04/temporarily-rolling-back-samesite.html, 2020.

[12] Cisco Adaptive Security Appliance. CVE-2019-1713. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1713, 2019.

[13] Cisco Umbrella. Umbrella Popularity List. https://umbrella-static.s3-us-west-1.amazonaws.com/index.html, 2019.

[14] Cloudacl. Web security service. http://www.cloudacl.com/, 2018.

[15] Clover, A. Css visited pages disclosure. *BUGTRAQ mailing list posting* (2002).

[16] Dymo. Missing Accept_languages in Request for Headless Mode. https://bugs.chromium.org/p/chromium/issues/detail?id=775911, 2017.

[17] Englehardt, S., and Narayanan, A. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016).

[18] Directive 2009/136/EC of the European Parliament and of the Council of 25 November 2009. *Official Journal of the European Union* (2009).

[19] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* (2016).
[20] Felten, E. W., and Schneider, M. A. Timing attacks on web privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2000).
[21] Fortinet. Fortiguard web filtering. http://www.fortiguard.com/, 2018.
[22] Fredrikson, M., and Livshits, B. Repriv: Re-imagining content personalization and in-browser privacy. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2011).
[23] Google. Leak of visited status of page in blink. https://chromereleases.googleblog.com/2018/05/stable-channel-update-for-desktop_58.html, 2018.
[24] Guha, S., Cheng, B., and Francis, P. Privad: practical privacy in online advertising. In *Proceedings of the USENIX conference on Networked Systems Design and Implementation (NDSI)* (2011).
[25] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CSS)* (2012).
[26] HTTPbis Working Group. Same-site Cookies (draft-west-first-party-cookies-05). https://tools.ietf.org/html/draft-west-first-party-cookies-05#section-4.1, 2016.
[27] HTTPbis Working Group. Same-site Cookies (draft-west-first-party-cookies-06). https://tools.ietf.org/html/draft-west-first-party-cookies-06#section-4.1, 2016.
[28] HTTPbis Working Group. Same-site Cookies (draft-west-first-party-cookies-07). https://tools.ietf.org/html/draft-west-first-party-cookies-07#section-4.1, 2016.
[29] Internet Engineering Task Force. Http state management mechanism. https://tools.ietf.org/html/rfc6265, 2016.
[30] Janc, A., and Olejnik, L. Web browser history detection as a real-world privacy threat. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2010).
[31] Jia, Y., Dong, X., Liang, Z., and Saxena, P. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing 19* (2015).
[32] Kotcher, R., Pei, Y., Jumde, P., and Jackson, C. Cross-origin pixel stealing: timing attacks using css filters. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2013).
[33] Laperdrix, P., Rudametkin, W., and Baudry, B. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2016).
[34] Lapowsky, I. California unanimously passes historic privacy bill. Wired, 06 2018.
[35] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Annual Network and Distributed System Security Symposium (NDSS)* (2019).
[36] Lee, S., Kim, H., and Kim, J. Identifying cross-origin resource status using application cache. In *Proceedings of the Network and Distributed System Symposium (NDSS)* (2015).
[37] Majestic. The Majestic Million. https://majestic.com/reports/majestic-million, 2019.
[38] Mowery, K., and Shacham, H. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of the Web 2.0 Workshop on Security and Privacy (W2SP)* (2012).
[39] Mozilla. Public Suffix List. https://publicsuffix.org/, 2007.
[40] Mozilla. Privacy and the :visited selector. https://developer.mozilla.org/en-US/docs/Web/CSS/Privacy_and_the_:visited_selector, 2018.
[41] Mozilla Bugzilla. Enable sameSite=lax by default on Nightly. https://bugzilla.mozilla.org/show_bug.cgi?id=1604212, 2019.
[42] Mozilla Bugzilla. Implement sameSite lax-by-default 2 minutes tolerance for unsafe methods. https://bugzilla.mozilla.org/show_bug.cgi?id=1608384, 2019.
[43] Mozilla Security Blog. Supporting Same-Site Cookies in Firefox 60. https://blog.mozilla.org/security/2018/04/24/same-site-cookies-in-firefox-60/, 2018.
[44] Network Working Group. HTTP State Tokens (draft-west-http-state-tokens-00). https://tools.ietf.org/html/draft-west-http-state-tokens-00, 2019.
[45] Network Working Group. Incrementally Better Cookies (draft-west-cookie-incrementalism-00). https://tools.ietf.org/html/draft-west-cookie-incrementalism-00#section-3.1, 2019.
[46] Network Working Group. Incrementally Better Cookies (draft-west-cookie-incrementalism-01). https://tools.ietf.org/html/draft-west-cookie-incrementalism-01#section-3.1, 2020.
[47] Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)* (2013).
[48] phpMyAdmin. CVE-2019-12616. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12616, 2019.
[49] Quantcast. Audience Insights That Help You Tell Better Stories. https://www.quantcast.com/top-sites/, 2019.

[50] Sanchez-Rola, I., Balzarotti, D., Kruegel, C., Vigna, G., and Santos, I. Dirty Clicks: a Study of the Usability and Security Implications of Click-related Behaviors on the Web. In *World Wide Web Conference (WWW)* (2020).
[51] Sanchez-Rola, I., Balzarotti, D., and Santos, I. BakingTimer: Privacy Analysis of Server-Side Request Processing Time. In *Annual Computer Security Applications Conference (ACSAC)* (2019).
[52] Sanchez-Rola, I., Dell'Amico, M., Kotzias, P., Balzarotti, D., Bilge, L., Vervier, P.-A., and Santos, I. Can I Opt Out Yet? GDPR and the Global Illusion of Cookie Control. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)* (2019).
[53] Sanchez-Rola, I., and Santos, I. Knockin' on Trackers' Door: Large-Scale Automatic Analysis of Web Tracking. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2018).
[54] Sanchez-Rola, I., Santos, I., and Balzarotti, D. Clock Around the Clock: Time-Based Device Fingerprinting. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018).
[55] Sangaline, E. Making Chrome Headless Undetectable. https://intoli.com/blog/making-chrome-headless-undetectable/, 2017.
[56] Sangaline, E. It is Not Possible to Detect and Block Chrome Headless. https://intoli.com/blog/not-possible-to-block-chrome-headless/, 2018.
[57] Schwartz, J. Giving the web a memory cost its users privacy. http://www.nytimes.com/2001/09/04/technology/04COOK.html, 2001.
[58] Singer, N. Do not track? advertisers say "don't tread on us". http://www.nytimes.com/2012/10/14/technology/do-not-track-movement-is-drawing-advertisers-fire.html, 2012.
[59] Smith, M., Disselkoen, C., Narayan, S., Brown, F., and Stefan, D. Browser history re: visited. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)* (2018).
[60] Stetic. Browser Statistics March 2020. https://www.stetic.com/market-share/browser/, 2020.
[61] Sutton, M. A wolf in sheep's clothing, the dangers of persistent web browser storage. *Black Hat DC Briefings (BHDC)* (2009).
[62] Swartz, A. Web.py web framework. http://webpy.org/, 2018.
[63] Symantec. The Need for Threat Risk Levels in Secure Web Gateways. https://www.symantec.com/content/dam/symantec/docs/white-papers/need-for-threat-tisk-Levels-in-secure-web-gateways-en.pdf, 2017.
[64] Symantec. WebPulse. https://www.symantec.com/content/dam/symantec/docs/white-papers/webpulse-en.pdf, 2017.
[65] The Chromium Projects. SameSite Updates. https://www.chromium.org/updates/same-site, 2016.
[66] The Chromium Projects. Cookie Legacy SameSite Policies. https://www.chromium.org/administrators/policy-list-3/cookie-legacy-samesite-policies, 2020.
[67] The WebKit Open Source Project. Implement Same-Site cookies. https://github.com/WebKit/webkit/commit/91ac5b831f84731aad164b48d53007f6e82d60d2, 2018.
[68] Toubiana, V., Narayanan, A., Boneh, D., Nissenbaum, H., and Barocas, S. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Symposium (NDSS)* (2010).
[69] Van Goethem, T., Joosen, W., and Nikiforakis, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2015).
[70] WebKit Bugzilla. Bug 198181: Cookies with SameSite=None or SameSite=invalid treated as Strict. https://bugs.webkit.org/show_bug.cgi?id=198181, 2019.
[71] Weinber, Z., Chen, E., Jayaraman, P., and Jackson, C. I still know what you visited last summer. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2011).
[72] Wondracek, G., Holz, T., Kirda, E., and Kruegel, C. A practical attack to de-anonymize social network users. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2010).
[73] World Wide Web Consortium. User timing. https://www.w3.org/TR/user-timing/, 2018.