Identifying Cross-origin Resource Status Using Application Cache

Sangho Lee, Hyungsub Kim, and Jong Kim
Department of Computer Science and Engineering
POSTECH, Korea
{sangho2, hyungsubkim, jkim}@postech.ac.kr

***Abstract*—HTML5 Application Cache (AppCache) allows web applications to cache their same- and cross-origin resources in the local storage of a web browser to enable offline access. However, cross-origin resource caching in AppCache has potential security and privacy problems. In this paper, we consider a novel web privacy attack that exploits cross-origin AppCache. Our attack allows a remote web attacker to exploit a victim web browser to exactly identify the status of target URLs: existence, redirection, or error. Especially, our attack can be performed without using client-side scripts, can concurrently identify the status of multiple URLs, and can exactly identify the redirections of target URLs. We further demonstrate advanced attacks that leverage the basic attack to de-anonymize and fingerprint victims. First, we determine the login status of a victim web browser by identifying URL redirections or errors due to absent or erroneous login information. Second, we probe internal web servers located in the local network of a victim web browser by identifying URL existence. We also suggest an effective countermeasure to mitigate the proposed attacks.**

I. Introduction

The Web has become the most popular distributed application platform due to its high cross-platform compatibility. Users can launch a web application on any web browser in any platform without modification or with negligible modification. Therefore, many applications, including email, calendars, word processors, and spreadsheets, are being implemented as web applications.

However, the Web's popularity has made it the most valuable attack target, so that users demand an in-depth security analysis of the Web to prevent attacks before they rapidly spread. Numerous researchers have considered various web attacks, such as clickjacking [16], cross-site scripting (XSS) [33], cross-site request forgery (CSRF) [4], and domain name system (DNS) rebinding [18], that attackers can exploit to steal sensitive information or to make profits. Despite the best efforts of researchers to reduce such security and privacy problems, unrevealed security threats probably still remain in web applications and web browsers due to undiscovered software vulnerabilities and problematic specifications. Consequently, researchers should detect and remove new vulnerabilities before attackers recognize and widely abuse them.

NDSS '15, 8-11 February 2015, San Diego, CA, USA
 ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23027

In this paper, we demonstrate a new web privacy attack that exploits security flaws of an HTML5 functionality, *Application Cache (AppCache)* [14]. AppCache allows web applications to cache resources in the local storage of a web browser to enable offline access to them. However, we discover security problems, *side channels*, of AppCache due to its *cross-origin resource caching*. By exploiting the security problems, a *web attacker* [2], who serves a malicious web application, can exploit a victim web browser to correctly identify the status of a target URL, such as whether the URL exists, whether the URL redirects the browser to another web page, or whether the URL returns an error code to the browser, without using error-prone timing information [9]. We name the attack a *URL status identification attack*.

We further describe advanced attacks that leverage the URL status identification attack. First, we can *determine the login status* of a victim web browser. Many web applications have web pages that (1) redirect a browser to a login page if the browser has no login information or (2) return an error code to a browser if the browser has erroneous login information [5], [6], [23]. By using such web pages, an attacker can identify which web sites a victim frequently visits and which web pages a victim is authorized to access. When an attacker can determine whether a victim is allowed to access web sites or web pages for specific companies, universities, regions, or groups, the attacker can *de-anonymize* the victim [34] and perform context-aware phishing [20].

Second, we can *probe internal web servers* located in the local network of a victim web browser. By using the URL status identification attack, an attacker can probe any URL including an internal URL. Probing internal URLs allows an attacker to probe networked devices (or *things*) in a victim's local network, such as routers, network printers, network-attached storage (NAS), smart TVs, and smart thermostats [10], [11], [24], [26]. Thus, the attacker can *fingerprint* the victim and can conduct succeeding attacks (e.g., DNS rebinding [18] and router reconfiguration [30]). The danger of internal web server probing will increase as the Internet of Things (IoT) becomes popular.

Our attack has three distinguishable features. First of all, our attack can obtain sensitive information *without using client-side scripts nor plug-ins*. Rather, it only uses an HTML document that declares an AppCache manifest which specifies

TABLE I. TARGET WEB BROWSERS.

Browser	Version
Chrome	34
Firefox	29
Internet Explorer	11
Opera	21
Safari	7

a target URL. Conventional security tools (e.g., NoScript [28]) usually disable or limit execution of suspicious client-side scripts and plug-ins, because most web attacks exploit client-side malicious scripts. However, such tools cannot protect user privacy from our attack because it leverages neither client-side scripts nor plug-ins. Some researchers have already considered scriptless attacks [12], [17], [27], but all of them rely on cascading style sheets (CSS) unlike our attack.

Second, our attack can *concurrently identify the status of multiple target URLs*. Attackers aim to develop a fast attack because they cannot guarantee that a victim spends a long time in their attack pages, so they have to obtain the victim's secrets as quickly as possible and as much as possible. However, conventional timing-based web privacy attacks [5], [9]–[11], [19], [21], [23], [24], [26], [27] cannot simultaneously infer the status of multiple URLs because concurrent network requests lead to timing errors. In contrast, our attack can identify the status of a target URL without timing, thereby inspecting multiple URLs concurrently (Section IV).

Third, our attack can *correctly recognize whether a URL redirection occurs when a victim web browser visits a target URL*, namely, it violates the requirement of atomic HTTP redirect handling [31]. To infer the status of a target URL, conventional attacks [5], [6], [10], [11], [24], [26] load the target URL via some tags (e.g., `img`, `script`, and `link`) and check when or whether `onload` or `onerror` events occur. Such tags transparently follow URL redirections for the atomic HTTP redirect handling, so that attackers cannot accurately recognize whether redirections occur. Therefore, identifying whether a URL redirection occurs and determining a login status according to a conditional URL redirection (Section V-A) are only exact with our attack.

We launched our attack on the recent versions of five major web browsers at the time of writing this paper, and confirmed that all web browsers which strictly followed the AppCache standard were vulnerable to our attack (Table I). One exception was Safari because it did not properly follow the up-to-date standard (Section III-D1). We reported our findings to Mozilla and Google, and they agreed that our attack could breach user privacy.

Our work makes the following contributions:

- **Novel attack.** To the best of our knowledge, this is the first in-depth study of AppCache security problems. All major web browsers that correctly implement AppCache suffer from the discovered problems. Although other researchers have considered AppCache poisoning [25] and AppCache-based DNS rebinding [22], they exploit not the security problem of AppCache but the security problem of DNS and networks. Thus, their studies differ from ours.
- **Strong attack.** Our attack can be performed without client-side scripts nor plug-ins, can simultaneously identify the status of multiple URLs, and can correctly identify the redirection of a target URL. These features make our attack difficult to defend, extend its attack coverage, and increase its performance, respectively.
- **Effective countermeasure.** We propose a countermeasure to mitigate our attack: a `Cache-Origin` request-header field. The countermeasure is essential to mitigate all of the security attacks that this work explores.

The remainder of this paper is organized as follows. Section II explains conventional cross-origin web privacy attacks. Section III introduces the HTML5 AppCache. Section IV describes a URL status identification attack based on AppCache. Section V demonstrates advanced attacks to determine a login status and probe internal web servers by using the URL status identification attack. Section VI discusses countermeasures against our attacks. Section VII presents related work. Lastly, Section VIII concludes this work. In addition, we describe an AppCache-based URL timing attack in Appendix.

II. CROSS-ORIGIN WEB PRIVACY ATTACKS

In this section, we briefly explain conventional cross-origin web privacy attacks. We mainly focus on attacks that rely on *timing channels that are unreliable but inevitable*. We introduce attack examples to infer browsing history, login status, and internal web servers.

A. Attack Model

The model of the cross-origin web privacy attack resembles that of CSRF attacks [4]. In the cross-origin web privacy attack, an attacker aims to obtain sensitive information of a victim web browser relevant to a target web application by convincing the victim web browser to visit an attacker's web site, which serves *slightly malicious* web pages. The malicious web pages contain no exploit codes to take control of the victim web browser or to inject malicious scripts into the target web application. Instead, the web pages contain legitimate HTML codes and scripts to *include cross-origin content while measuring fetch latency* to obtain side-channel information, such as the browsing history and login status of the victim web browser. Therefore, it is difficult to determine the maliciousness of the web site.

B. Cross-origin Content Inclusion

HTML has various methods of including cross-origin content. We briefly explain and compare them.

1) Specific content inclusion: HTML provides tags (e.g., `img`, `script`, and `link`) to embed specific types of same- or cross-origin content in a web page, such as images, scripts, and CSSs. The tags successfully include a URL that indicates a valid resource with a matched content type. But, the tags fail to include a URL when the URL is invalid (e.g., connection failure, non-existent resource, and unauthorized access) or the URL indicates a resource with an unmatched content type. Finally, web browsers call either the `onload` or `onerror` event handlers according to successful or unsuccessful content inclusion via the tags.

Although the main purpose of the explained tags is to include content with specific types, attackers can abuse the tags to obtain side-channel information by including arbitrary content and checking an error status while measuring latency. Web browsers cannot determine the content type of a URL until they receive an actual resource, so they send a normal `GET` request to a web application to fetch the resource. When the content type of the received resource differs from the tag type, the web browsers abort the content inclusion and fire an error event. However, attackers can infer the status of a URL from its fetch latency because the latency varies for various reasons, such as whether the browsers have previously visited the URL, whether the browsers are logged in, and whether the URL exists. Attackers can thereby guess sensitive information by using information implied by the varied fetch latency.

2) Arbitrary content inclusion: HTML provides tags (e.g., `frame`, `iframe`, `object`, and `embed`) to embed arbitrary content in a web page. The main purpose of the `frame` and `iframe` tags is to embed other HTML documents, and the main purpose of the `object` and `embed` tags is to embed multimedia, such as audio, video, and PDF files. The tags only support the `onload` event handler, so that attackers should guess the status of a URL by measuring how much time a web browser spends before firing an `onload` event.

However, the preceding tags are unsuitable for performing web privacy attacks due to two shortcomings. First, the fetch latency is unpredictable because the tags try to receive all resources (e.g., images, scripts, and CSSs) that compose a web page before rendering the resources. This procedure adds a high amount of noise to the time measurement [5]. Second, to avoid security problems (e.g., clickjacking [16] on login pages), many modern web applications do not allow web browsers to load their web pages in such tags. The web applications use an HTTP response-header field `X-Frame-Options` or a *frame busting* code [29] to prevent such content inclusion. Therefore, the HTML tags for arbitrary content inclusion are unsuitable for performing web privacy attacks.

C. Inferring Login Status

We explain a timing attack that uses variance in fetch latency to infer the login status of a victim web browser [5]. An attacker can reveal the real identity of a victim web browser's user according to which web sites the user is frequently logged in. When a web browser accesses the front pages of web applications, many of them provide different web pages to the browser according to the login status. They usually redirect a logged-in browser to a personalized web page, thereby introducing additional network delay. Malicious web pages leverage this delay to infer login status by manipulating a web browser to visit the front page of a target web application while measuring the latency. High latency implies that the web browser is logged in to the target web application.

A countermeasure to this attack is to make web applications spend constant time to process HTTP requests [5]. But, guaranteeing constant processing time is not only difficult but also incurs much overhead.

The CSS-filter-based attack [23] can identify login status by exploiting the difference in filtering latency between logged-in and non-logged-in web pages. However, two shortcomings make this attack less practical than others. First, it takes much time to measure the latency of CSS filtering. Second, target web applications should allow the `iframe` tag, but recent and security-aware web applications usually disallow such a tag (Section II-B2).

D. Inferring Internal Web Server

We depict a timing attack to identify internal web servers located in the local network of a victim web browser [10], [11], [24], [26]. The basic idea of this attack is using HTML tags (e.g., the `img` and `script` tags) to include arbitrary URLs of internal web servers while waiting for `onerror` events. Attackers can guess the servers' status from the elapsed time.

Knowing internal web servers is an important privacy breach because it can reveal what kinds of routers, network printers, and NAS a victim uses. An attacker can use such information to fingerprint a victim web browser. Furthermore, this knowledge becomes the basis of other security attacks, such as DNS rebinding [18] and router reconfiguration [30]. Usually, a firewall protects internal hosts from outsiders such that attackers attempt to make a victim web browser execute scripts to investigate servers in the internal network of the victim web browser.

To prevent this attack, a web browser should prevent external scripts from accessing its internal network. We also require DNS pinning and host name authorization to prevent DNS rebinding attacks [18].

E. Limitations of Conventional Attacks

Conventional cross-origin web privacy attacks have some limitations. First, their accuracy is relatively low due to unreliable page fetch latency affected by a number of error sources, such as network condition, web server loads, and client loads. Attackers can reduce the noise by averaging data from a number of timing samples, but this process requires an unreasonable amount of time to collect a sufficient number of samples. Furthermore, sampling becomes meaningless when a victim web browser visits web pages via wireless networks or Tor [7] due to their high and unstable network latency.

Second, the conventional attacks are inefficient because they cannot measure the fetch latency of multiple URLs in parallel. If attackers open more than one connection with target web applications, interference between multiple connections causes timing errors. Accordingly, attackers should probe URLs one by one.

III. HTML5 AppCache

In this section, we explain the HTML5 AppCache in detail. We especially focus on *when AppCache fails* and *how AppCache handles failures*, because they are the most important basis of our attacks presented in a later section.

A. Declaration

We depict how a web application announces that it uses AppCache, and how the web application specifies which resources web browsers should store in their local storage.

```
<!DOCTYPE HTML>
<html manifest="example.appcache">
...
</html>
```

Listing 1. HTML document that declares an AppCache manifest.

```
CACHE MANIFEST

CACHE:
/logo.png
https://example.cdn.com/external.jpg

NETWORK:
*

FALLBACK:
/ /offline.html
```

Listing 2. AppCache manifest file.

First, the web application declares the path of an AppCache manifest file (`example.appcache`) that corresponds to an HTML document in its `html` tag (Listing 1). The manifest file and the HTML document must belong to the same origin, and the content type of the manifest file should be `text/cache-manifest`.

Next, through the manifest file, the web application specifies URLs that web browsers should cache (Listing 2). A manifest file starts with `CACHE MANIFEST` and has three sections: `CACHE`, `NETWORK`, and `FALLBACK`. (1) The `CACHE` section declares URLs that need to be stored in local storage. Each scheme of the declared URLs should be the same as the main HTML document's scheme. For example, when the main HTML document's scheme is HTTP, AppCache ignores HTTPS URLs listed in the `CACHE` section. When the scheme is HTTPS, AppCache ignores HTTP URLs listed in the `CACHE` section. (2) The `NETWORK` section declares whitelisted URLs that web browsers can download from outside. Web browsers treat URLs listed in neither `CACHE` nor `NETWORK` sections as *unreachable*. We can use an asterisk to allow arbitrary URLs. (3) The `FALLBACK` section declares alternative URLs to use when original URLs are inaccessible. The first URL is the original resource, and the second URL is the fallback to substitute for the first one. The `FALLBACK` section only allows relative URLs because replacing a URL with another URL that belongs to a different origin can violate SOP.

B. Download and Update Procedures

We illustrate the two procedures of AppCache: download and update procedures. The first time a web browser visits a web page that declares an AppCache manifest, the browser performs the download procedure. Otherwise, it performs the update procedure.

1) Downloading non-cached web page: We first describe the AppCache download procedure for a newly-visited web page and the corresponding events that are fired during the procedure. A web browser initiates the following download procedure when it visits a web page that declares an AppCache manifest for caching specific resources.

1) The browser attempts to fetch and parse the manifest while firing a `checking` event to an AppCache object. If the manifest either has errors or is non-existent, the browser terminates the download procedure and fires an `error` event.
2) The browser starts to download resources listed in the manifest while firing a `downloading` event.
3) The browser downloads each of the resources while firing a `progress` event for each resource. If the browser *cannot cache at least one of the resources* (Section III-C) or *recognizes the changes in the manifest* while downloading the resources, the browser terminates the download procedure and fires an `error` event.
4) The browser stores the downloaded resources in its local storage and fires a `cached` event.

2) Updating cached web page: Next, we describe the AppCache update procedure for a cached web page and corresponding events fired during the procedure. A web browser initiates the following procedure to update corresponding resources when it visits a web page that has already been cached in its local storage.

1) The browser attempts to fetch and interpret the manifest originating from the remote server while firing a `checking` event. First, if the content of the manifest does not change, the browser terminates the update procedure and fires a `noupdate` event. Next, if the manifest either has errors or is unreachable due to network failures, the browser terminates the update procedure and fires an `error` event. Lastly, if the manifest no longer exists in the remote server, the browser terminates the update procedure, deletes the cached resources, and fires an `obsolete` event.
2) The browser starts to download resources listed in the manifest while firing a `downloading` event.
3) The browser re-downloads each of the resources while firing a `progress` event for each resource. If the browser cannot cache at least one of the resources or if the manifest changes during re-downloading, it terminates the update procedure and fires an `error` event.
4) The browser stores the re-downloaded resources in its local storage and fires an `updateready` event.

3) Error handling: To avoid partial resource replacement to preserve content consistency, AppCache *reverts* completely to its previous status when it encounters errors during the download or update procedures. AppCache discards all new resources that were successfully downloaded during the failed download or update procedures.

4) Web page refreshing: Occasionally, an AppCache procedure finishes after a web page has been loaded because a web browser performs the procedure in the background. Therefore, the web browser needs to refresh the web page to reflect the most recent version.

C. Non-cacheable URLs

We state the types of URLs that AppCache does not cache and returns errors. Using such information allows us

to identify the status of a target URL, which will be explained in Section IV. AppCache does not cache URLs that satisfy any one of the following three conditions.

- **Invalid URL.** AppCache does not cache this kind of URL because the URL returns no content for caching. If a web application returns client or server error codes or does not respond when AppCache accesses a URL of the web application, AppCache treats the URL as invalid.
- **Dynamic URL.** AppCache does not cache this kind of URL because offline access to dynamic content is almost meaningless. Web applications use HTTP response-header fields (`Cache-Control` or `Content-Length`) to specify their dynamic content. AppCache does not cache content when the response header contains a `no-store` directive in a `Cache-Control` field [14] or has no `Content-Length` field (i.e., *chunked encoding*).
- **URL with redirections.** AppCache does not cache this kind of URL to avoid a security problem. Since web browsers refer to the cached content with a URL that is specified in a manifest file, allowing redirections can violate SOP. For example, some wireless access points (APs) use a captive portal technique that redirects web browsers to a special web page for authentication or payment. If AppCache allows this redirection, the stored content differs from the content that a web application intends to cache, but has the same URL. When the stored content embeds malicious scripts, this problem becomes serious because SOP is no longer guaranteed. Furthermore, malicious web applications can abuse redirections to cache the content of target web pages under their origin to execute their malicious scripts on the target web pages. Thus, to enforce SOP, AppCache does not resolve URL redirections.

Although AppCache restricts standard URL redirections that use 3xx status codes, it ignores non-standard redirection methods (e.g., the `meta refresh` tag and the JavaScript object `window.location`). When AppCache encounters a web page that uses such a non-standard redirection method, AppCache does not follow a redirection, but caches the web page "as is".

D. Browser Differences

We analyze differences in AppCache implementations of different web browsers. Due to the differences, some web browsers are more vulnerable to our attack than others, and some other web browsers are robust against our attack explained in Section IV.

1) Secured resources: Safari does not cache cross-origin HTTPS URLs in a manifest file, so that we cannot attack cross-origin HTTPS URLs when a victim uses Safari. The previous version of the AppCache standard [13] specified that a web browser should only cache URLs from the same origin as a manifest when the manifest's scheme is HTTPS. Therefore, in the past, web application developers were not able to use AppCache to cache cross-origin HTTPS URLs. This is bad for secured web applications that want to cache resources provided by secured content delivery networks (CDNs). The recent standard [14] relaxes this restriction: when a manifest's scheme is HTTPS, a web browser can cache any HTTPS URLs but no HTTP URLs. An exception is Safari because it does not use the recent standard changes at the time of writing this paper.

2) `no-store` directive: Chrome, Opera, and Safari ignore the `no-store` directive of a HTTP resource, so that we can attack `no-store` HTTP resources when a victim uses one of the web browsers. The AppCache standard [14] specifies that a web browser should not cache any resources with a `no-store` directive. But, we observe that Chrome, Opera, and Safari ignore a `no-store` directive when they cache HTTP resources via AppCache.

3) Referrer information: Chrome, Opera, and Safari send no referrer information during an AppCache process, so that a stealthy attack is possible. The AppCache standard [14] does not specify whether a web browser should send referrer information during an AppCache process. Accordingly, browser vendors choose different policies: Firefox and Internet Explorer record the URL of an HTML document that declares an AppCache manifest in a `Referrer` request-header field whereas Chrome, Opera, and Safari specify no referrer information in an HTTP request. The lack of referrer information implies that target web applications cannot recognize who forces a victim web browser to investigate themselves.

```
1  <?php
2  header("Content-Type: text/cache-manifest");
3
4  $target = "https://target.net"; //dynamically
       assigned
5  echo "CACHE MANIFEST\n";
6  echo "CACHE:\n";
7  echo "$target\n\n";
8  echo "NETWORK:\n";
9  echo "*\n";
10 ?>
```

Listing 3. PHP-based AppCache manifest to perform a URL status identification attack.

IV. URL Status Identification Attack

In this section, we illustrate an AppCache-based URL status identification attack that *does not rely on timing*. This attack is possible due to a standard behavior of AppCache: to avoid content inconsistency and security problems, AppCache should fail when any URL listed in a manifest is non-cacheable. By using this attack, an attacker can correctly determine the status of target URLs because this attack does not rely on unreliable timing information. We demonstrate both script-based and scriptless attacks.

A. Attack Manifest

An AppCache manifest written in PHP (example in Listing 3) can be used to perform a URL status identification attack. The example only specifies a single target URL (`https://target.net`) that attackers want to identify.

```
var appCache = window.applicationCache;

function handleError(e) {
    // fail to download a given URL
    var img = new Image();
    img.src = "/results.png?failure";
}

function handleCached(e) {
    // succeed to download a given URL
    var img = new Image();
    img.src = "/results.png?success";
}

appCache.addEventListener('error', handleError
    , false);
appCache.addEventListener('cached',
    handleCached, false);
appCache.addEventListener('updateready',
    handleCached, false);
```

Listing 4. Script-based URL status identification attack using AppCache.

B. Script-based Attack

Listing 4 shows a JavaScript code that uses AppCache to perform a URL status identification attack. A web browser fires an `error` event when at least one URL in a manifest is non-cacheable (Section III-B and Section III-C). Thus, attackers can identify whether a target URL in the manifest is cacheable according to whether an `error` event occurs. We added event handlers to an AppCache object to capture the events and to notify an attack web application that they have occurred.

C. Scriptless Attack

The overall procedure of our scriptless URL status identification attack (Fig. 1) consists of four steps. First, when a victim web browser visits an attack web application that specifies an AppCache manifest, the attack web application records the information of the victim web browser. Second, the victim web browser downloads an attack manifest and extracts a target URL from it. Third, the victim web browser attempts to download the URL. Lastly, if the victim web browser successfully downloads the target URL (Fig. 1a), the browser re-downloads the manifest to check its changes. If the victim web browser fails to cache the target URL (Fig. 1b), the browser terminates the AppCache procedure and does not re-download the manifest to check it. Consequently, attackers can identify whether the victim web browser caches the target URL according to whether the browser re-downloads the manifest.

Additionally, to re-confirm whether the AppCache procedure succeeds, attackers can refresh the attack web page (e.g., using the `meta refresh` tag). If the AppCache procedure has succeeded, the victim web browser only checks the manifest for each page refresh (Fig. 1a). Otherwise, the victim web browser starts the AppCache procedure from the beginning (Fig. 1b).

D. Concurrent Attack

The URL status identification attack does not rely on timing such that we can conduct the attack in parallel. By

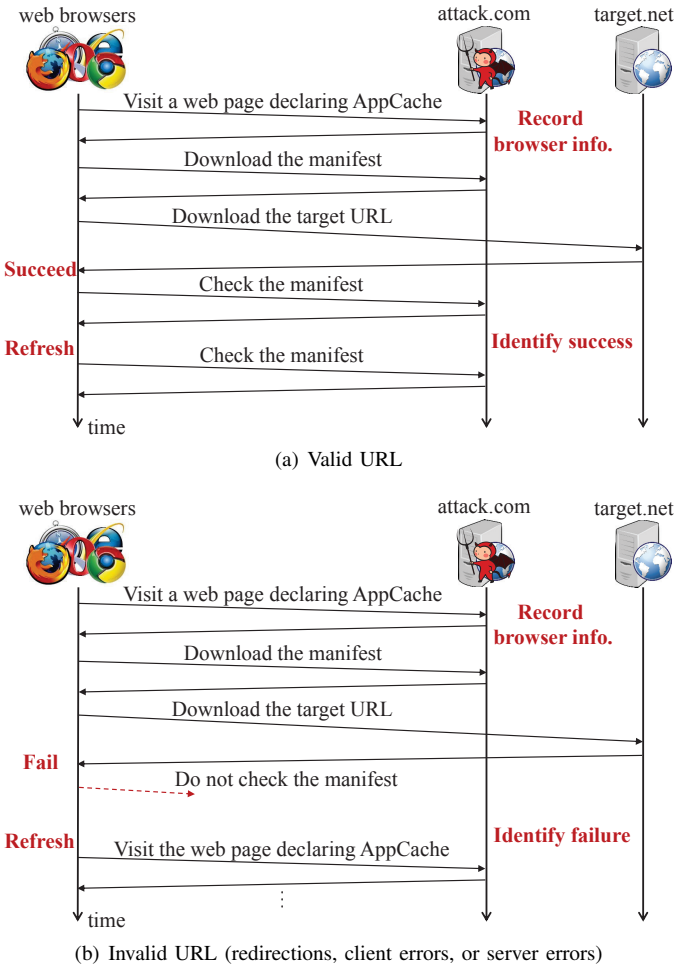

Fig. 1. Scriptless URL status identification attack using AppCache.

using multiple web pages and AppCache manifests located in multiple `iframe` tags, attackers simultaneously identify a number of URLs. This differs from conventional attacks that rely on timing [5], [9], [19], [21], [27].

Fig. 2 shows dynamic web pages and AppCache manifests to concurrently perform scriptless URL status identification attacks by using multiple hidden `iframe` tags. First, the main attack page (`attack_all.php`) contains a number of hidden `iframe` tags that each point to a sub-attack page (`attack_each.php`) while using a `GET` parameter to deliver target URLs. Next, each sub-attack page declares an attack manifest (`manifest.php`) while forwarding the received target URL to the manifest. Lastly, each attack manifest specifies the received target URL in the `CACHE` section for identification. We developed some PHP scripts (Listing 5) to dynamically generate the explained web pages and AppCache manifests.

E. Performance of Concurrent Attack

We evaluated the performance of concurrent, scriptless URL identification attack. Fig. 3 shows the execution time of concurrent attacks according to the number of target web pages (10 to 200 front web pages chosen from Alexa Top domains) on Windows 8.1. The AppCache implementations of Chrome,

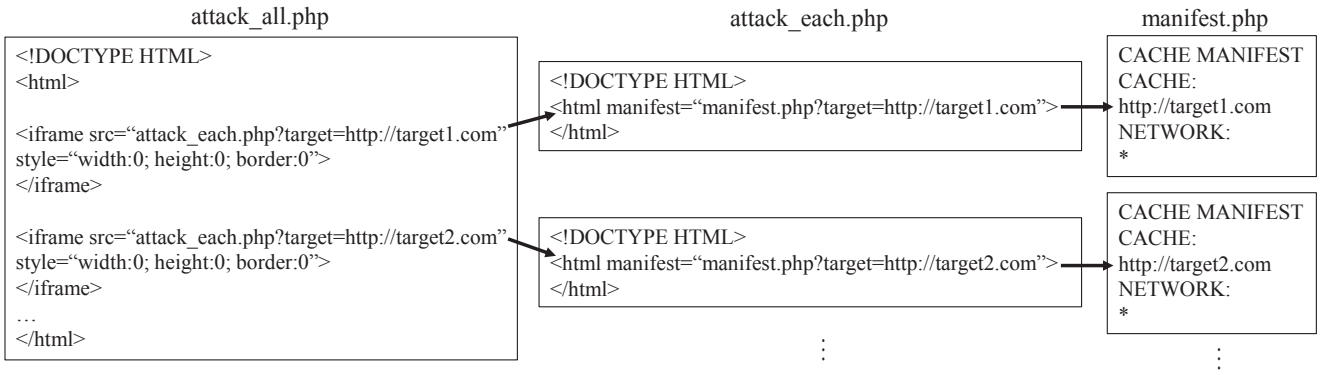

Fig. 2. PHP-based web pages and AppCache manifests to concurrently perform scriptless URL status identification attacks.

```php
<?php // attack_all.php
header("Content-Type: text/html");

echo "<!DOCTYPE HTML>\n";
echo "<html>\n";
$fp = fopen("target_urls.txt", "r");
while (!feof($fp)) {
    $target = substr(fgets($fp),0,-1);
    echo "<iframe src=\"attach_each.php?target
        =$target\" style=\"width:0; height:0;
        border:0\"></iframe>\n";
}
echo "</html>\n";
?>
...
<?php // attack_each.php
header("Content-Type: text/html");

$target = $_REQUEST['target'];
echo "<!DOCTYPE HTML>\n";
echo "<html manifest=\"manifest.php?target=
    $target\">\n";
echo "</html>\n";
?>
...
<?php // manifest.php
header("Content-Type: text/cache-manifest");

$target = $_REQUEST['target']; // dynamically
    changed according to the parameter passed
    through Lines 9 and 19
echo "CACHE MANIFEST\n";
echo "CACHE:\n";
echo "$target\n\n";
echo "NETWORK:\n";
echo "*\n";
?>
```

Listing 5. PHP scripts to perform concurrent attacks.

Internet Explorer, and Opera concurrently processed AppCache manifests in multiple `iframe` tags. Therefore, the overall execution time of the attack was short on these browsers. For example, when we attacked 200 web pages in parallel, the average times taken to process each web page were 0.11 s when using Chrome, 0.11 s when using Opera, and 0.27 s when using Internet Explorer. In contrast, Firefox processed AppCache manifests sequentially. When we attacked 200 web

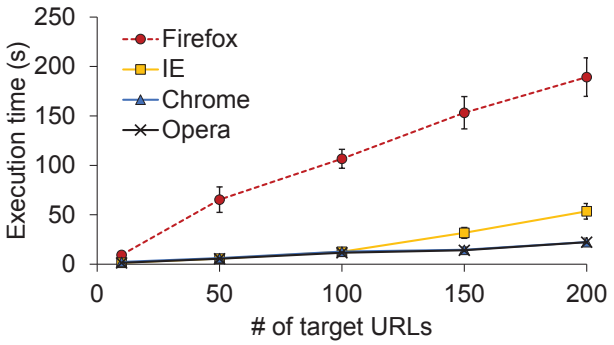

Fig. 3. Execution time of concurrent attacks on a number of front web pages selected from Alexa Top domains. The target platform was Windows 8.1. We repeated each experiment 10 times and report the average results where the error bars represent standard deviation.

pages by using Firefox, the average time to process each web page was to 0.95 s.

We conducted the same experiment by using Chrome on Ubuntu 12.04.2 LTS and OS X 10.9.1, and obtained almost the same results. Thus, we concluded that a victim web browser's platform is irrelevant to the speed of identification attack against reachable URLs. Consideration on unreachable URLs is in Section V-B.

V. Advanced Attacks

In this section, we describe how we use the URL status identification attack explained in Section IV to perform advanced web privacy attacks: login status determination and internal web server probing.

A. Login Status Determination

We can use the URL status identification attack to determine the login status of a victim web browser, since many web applications conditionally return a redirection code, a client error code, or a server error code according to whether the browser has proper login information. We confirmed that the five web browsers attached (authentication) cookies to their HTTP requests whenever the browsers downloaded resources listed in an AppCache manifest. Therefore, by inspecting whether AppCache procedures succeed or fail, we can identify whether a victim web browser has proper login information.

https://www.amazon.com/gp/wallet?ie=UTF8&ref_=ya_wallet
➥ https://www.amazon.com/ap/signin?...
http://my.ebay.com/ws/eBayISAPI.dll?MyeBay
➥ https://signin.ebay.com/ws/eBayISAPI.dll?SignIn...
https://instagram.com/accounts/edit/
➥ https://instagram.com/accounts/login/?next=/accounts/edit/
https://sourceforge.net/projects/filezilla/reviews/new
➥ https://sourceforge.net/account/login.php?return_to=...
http://www.tumblr.com/dashboard
➥ https://www.tumblr.com/login?redirect_to=/dashboard
http://wordpress.com/post/
➥ http://wordpress.com/wp-login.php?redirect_to=...
https://www.yelp.com/profile_bio
➥ https://www.yelp.com/login?return_url=/profile_bio
http://www.youtube.com/feed/subscriptions
➥ https://accounts.google.com/ServiceLogin?...

Fig. 4. Examples of URLs that redirect non-logged-in web browsers to login pages.

https://bitbucket.org/account/user/⟨user-id⟩
https://github.com/⟨user-id⟩/⟨repository-name⟩/settings
http://stackoverflow.com/users/edit/⟨user-id⟩
http://twitpic.com/media/editLocation/⟨photo-id⟩
http://⟨blog-id⟩.wordpress.com/wp-admin
https://ndss2015.ccs.neu.edu/paper/⟨paper-no⟩

Fig. 5. Examples of private URLs that return errors to web browsers without proper login information.

We evaluated our attack by showing some real URLs of popular web applications that were vulnerable to the attack. First, by using URLs that conditionally redirected a web browser to login pages, we were able to identify whether a victim web browser was *logged in* to some web applications. Fig. 4 shows some URLs that redirected a web browser to login pages when the browser had no login information. If a web browser with login information uses AppCache to cache such URLs, the browser succeeds because no redirection occurred. In contrast, if a web browser without login information uses AppCache to cache such URLs, the browser fails due to redirections. Consequently, we can use the URL status identification attack to confirm whether a victim web browser is currently logged in to some web applications when they have such conditional URLs. In addition, some web applications (e.g., SourceForge) redirected a logged-in web browser to HTTPS URLs when the browser attempted to access any HTTP URL of them. In such a case, attackers can exploit conditional redirections of the HTTP URLs to distinguish whether a victim web browser has logged in to the web applications.

Second, we were able to use conditional errors to identify whether a victim web browser had proper login information to access *private web pages* that were allowed to authorized users. Fig. 5 shows examples of private URLs that returned either client or server error codes when a web browser had erroneous login information. By using the URL status identification attack to identify whether errors occur, we were able to infer the victim web browser's user, blog, or photo IDs for the corresponding web applications. Definitely, URLs with randomly-selected IDs make an attacker's search space tremendously large. Therefore, to enhance practicability of our attacks, we should reduce search space by using techniques such as exploiting browsing history and group information [34].

Third, we were able to identify the *preference* of a victim

TABLE II. APPCACHE TIMEOUT OF AN UNREACHABLE URL ACCORDING TO PLATFORMS. CHROME, FIREFOX, AND OPERA HAD ALMOST THE SAME TIMEOUT VALUE IN THE SAME PLATFORMS. WE REPEATED EACH EXPERIMENT 10 TIMES.

Platform or Browser	Timeout (s)	σ
OS X 10.9	76.48	0.162
Windows 8.1	21.00	0.114
Internet Explorer	7.01	0.006
Ubuntu 12.04	3.00	0.002

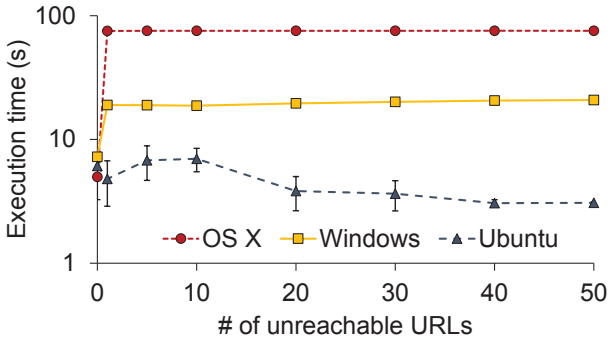

Fig. 6. Execution time of concurrent URL identification attacks on 50 URLs where some of them were unreachable (Chrome). We repeated each experiment 10 times and report the average results.

user according to his or her following and group information. For example, we discovered that the URL to follow a Tumblr blog redirected a web browser to the URL to unfollow the blog when a user had already followed the blog. When a web browser visited http://www.tumblr.com/follow/⟨blog-id⟩, Tumblr redirected the browser to http://www.tumblr.com/unfollow/⟨blog-id⟩ if the user had already followed the blog. We also identified that the URL to leave a Flickr group redirected a web browser to the main page of the group when a user was not a member of the group. When a web browser visited http://www.flickr.com/groups_leave.gne?id=⟨group-id⟩, Flickr redirected the browser to http://www.flickr.com/groups/⟨group-id⟩ if the user was not a member of the group. Thus, attackers were able to use the URL status identification attack to infer which Tumblr blogs a victim user followed and which Flickr groups a victim user joined. Furthermore, attackers were able to use co-memberships [34] to infer a victim user's Flickr user ID because public Flickr groups opened their member lists.

B. Internal Web Server Probing

We can use the URL status identification attack to manipulate a victim web browser to probe internal web servers in its local network. Although numerous researchers have considered approaches to probe internal web servers [10], [11], [24], [26], they rely on client-side scripts or plug-ins that security tools may block. In contrast, our URL status identification attack demands no client-side script, so that we can probe web servers even when a victim web browser disables or limits script execution. Appendix A introduces URLs of some internal web servers.

We evaluated the execution time of concurrent internal web-server probing attacks. In many cases, most internal IP addresses were unreachable because they were not assigned to hosts. Also, a TCP connection timeout value due to unreachable URLs was usually greater than the value of page fetch latency, so that the execution time of the internal web

server probing mainly depended on whether target URLs were unreachable. Table II shows measured AppCache timeout values of a single unreachable URL that consisted of a literal IP address, instead of a domain name, belonging to our campus, with various platforms. OS X had the greatest timeout value and Ubuntu had the smallest timeout value. Chrome, Firefox, and Opera had almost the same timeout values in the same platforms, but Internet Explorer had a different timeout value.

Fig. 6 shows the execution time of concurrent internal web server probing using Chrome. The number of targets URLs was 50, consisting of 0 to 50 unreachable URLs and 50 to 0 reachable URLs. All URLs belonged to our campus. The timeout value of OS X was greater than those of Ubuntu and Windows, so that the execution time of internal web server probing was longest when a victim web browser's platform was OS X. We also identified that the number of unreachable URLs did not affect the overall execution time because Chrome concurrently opened multiple sockets for AppCache.

Unlike other web browsers, Firefox was secure against the internal web server probing due to its sequential AppCache handling. For example, it took 7648 s and 2100 s when we performed URL identification attacks on 100 unreachable URLs by using Firefox in OS X and Windows, respectively. Since most victim users will not spend such a long time in an attack web page, we conclude that Firefox is secure against our attack when its platform is OS X or Windows.

VI. Countermeasures

In this section, we present our countermeasures to mitigate the proposed attacks. We first depict some naïve countermeasures with shortcomings and suggest our solution.

A. Problematic Countermeasures

We present some countermeasures that partially prevent our attacks or that prevent our attacks but lead to other problems. First, we can revise AppCache to ask user permissions to allow web applications to cache resources as Firefox does. This countermeasure prevents our attacks only if a user correctly judges whether a web application is malicious.

Second, we can revise AppCache to not check the changes in a manifest during download or update procedures as Safari does. This countermeasure, however, results in an AppCache inconsistency problem. Further, it cannot prevent a URL status identification attack if an attacker refreshes an attack page to re-confirm an AppCache procedure.

Third, we can revise AppCache to check the manifest even when some resources are non-cacheable. This countermeasure prevents a scriptless URL status identification attack only when an attacker does not refresh an attack page.

Fourth, we can attach a `no-store` directive to HTTP responses from web applications. This countermeasure prevents all our attacks, but makes AppCache meaningless because web browsers no longer cache resources.

Lastly, we can modify vulnerable web pages that conditionally redirect web browsers to login pages or that return error codes according to a login status. For example, we can use a login pop-up window instead of redirections and a custom error page with 200 OK instead of an error code. This countermeasure prevents a URL status identification attack, but finding and modifying all vulnerable web pages are sophisticated tasks.

B. Restricting Cross-origin AppCache

We aim to restrict arbitrary cross-origin AppCache to protect browser and URL status from the URL status identification attack. One possible solution is to apply the `Origin` request-header field of cross-origin resource sharing (CORS) [32] to AppCache procedures, although this approach can violate the principle of least privilege. The `Origin` header field allows a web application to identify which web applications initiate cross-origin requests so that the web application can deny requests from unknown or blacklisted web applications. However, the `Origin` header field further asks a permission to allow client-side scripts to access the requested resource, which is unnecessary for AppCache. Therefore, we require another method that only asks a web application whether it allows resource caching.

We suggest a new HTTP request-header field that contains the origin of an AppCache manifest; this field, `Cache-Origin`, resembles the `Origin` header field of CORS. The `Cache-Origin` header field only asks web applications whether they permit *caching* of their resources, unlike the `Origin` header field which requests *access permissions* to their resources. A web browser must attach the `Cache-Origin` header field to its HTTP requests during AppCache procedures.

By using the `Cache-Origin` header field, a web application can identify other web applications that request to cache its resources. When the web application doubts the requesters or caching the requested resources can reveal sensitive information (e.g., access-controlled resources), the web application either assigns a `no-store` directive to its response header or returns an error code to abort an AppCache procedure. Attackers can no longer identify browser and URL status because their AppCache procedures always fail. Even if some attackers bypass the `Cache-Origin` check, they cannot identify a browser status when the target web application disallows web browsers to cache sensitive resources.

We modified a build of Chromium (35.0.1856.0) to introduce a `Cache-Origin` request-header field during AppCache procedures (Listings 6). Adding three lines of code was enough to enable this countermeasure with negligible performance overhead.

The `Cache-Origin` request-header field is a minor revision of the `Origin` request-header field, so we believe that adopting `Cache-Origin` is not a big deal of the web standard. Otherwise, using `Origin` during AppCache procedures is at least desired to prevent our attack.

VII. Related Work

In this section, we introduce two AppCache attacks that manipulate DNS information: AppCache poisoning [25] and AppCache-based DNS rebinding [22]. AppCache poisoning attempts to store fake login pages in AppCache to steal login credentials. When a victim web browser visits some web pages via an attacker's network (e.g., a rogue AP), the attacker

```
/* src/webkit/browser/appcache/
    appcache_update_job.cc */
void AppCacheUpdateJob::URLFetcher::Start() {
  request_->set_first_party_for_cookies(job_
      ->manifest_url_);
  request_->SetLoadFlags(request_->load_flags
      () | net::LOAD_DISABLE_INTERCEPT);
  if (existing_response_headers_.get())
    AddConditionalHeaders(
        existing_response_headers_.get());

  /* Set a Cache-Origin header field */
  net::HttpRequestHeaders headers;
  headers.SetHeader("Cache-Origin", job_->
      manifest_url_.GetOrigin().spec());
  request_->SetExtraRequestHeaders(headers);

  request_->Start();
}
```

Listing 6. Modified Chromium code to attach a Cache-Origin request-header field during AppCache procedures.

injects hidden `iframe` tags that point to target login pages in responses. The victim web browser then sends requests to the target login pages. The attacker intercepts the requests and responds with fake login pages that look the same as the original login pages while declaring an AppCache manifest and including backdoors. Later, even when the victim web browser visits the target login pages via a secured network, it will load the fake login pages from AppCache. To mitigate this attack, we need to use private browsing modes [1] in an insecure network, and use HTTP strict transport security (HSTS) [15] or HTTPS Everywhere [8] to secure login pages.

AppCache-based DNS rebinding is a modification of the original DNS rebinding attack [18], which attempts to violate SOP by changing domain-to-IP mapping with a short-lived DNS entry. In the original form, when a victim web browser visits an attacker's web site, the attacker delivers some malicious scripts to the victim web browser while associating the domain name of the web site with a target IP address. Subsequently, the malicious scripts can send arbitrary same-origin requests to the target IP address because they have the same domain name. To mitigate this attack, modern web browsers maintain domain-to-IP mapping for a while (*DNS pinning*). However, the two characteristics of AppCache allow attackers to write a malicious script executed after domain-to-IP mapping changes [22]: (1) allowing web sites to persistently cache arbitrary resources in web browsers and (2) supporting a JavaScript API to recognize whether a script comes from a local cache or a server. To eradicate the attack, Johns et al. [22] suggest an `X-Server-Origin` response-header field that lists server-provided origin information.

VIII. Conclusion

This paper introduced a new web privacy attack that indirectly identified the status of cross-origin URLs by using HTML5 AppCache without client-side scripts nor plug-ins. We confirmed that all major web browsers which supported AppCache were vulnerable to our attacks. We also suggested an effective countermeasure: a `Cache-Origin` request-header field. The countermeasure successfully mitigated our attacks.

Acknowledgment

We would like to thank the anonymous reviewers for their invaluable comments and suggestions. This work was supported by ICT R&D program of MSIP/IITP. [14-824-09-013, Resilient Cyber-Physical Systems Research]

References

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh, "An analysis of private browsing modes in modern browsers," in *USENIX Security Symposium*, 2010.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *Computer Security Foundations Symposium (CSF)*, 2010.

[3] AnswersThatWork, "List of default router passwords and default router IP addresses," http://www.answersthatwork.com/Download_Area/ATW_Library/Networking/Network__4-Admin_List_of_default_Router_Passwords_and_IP_addresses_Netgear_D-Link_Belkin_Linksys_Others.pdf, 2013.

[4] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[5] A. Bortz, D. Boneh, and P. Nandy, "Exposing private information by timing web applications," in *International World Wide Web Conference (WWW)*, 2007.

[6] K. Brewster, "Patching privacy leaks," http://kentbrewster.com/patching-privacy-leaks/, 2008.

[7] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *USENIX Security Symposium*, 2004.

[8] Eletronic Frontier Foundation, "HTTPS Everywhere," https://www.eff.org/https-everywhere.

[9] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *ACM Conference on Computer and Communications Security (CCS)*, 2000.

[10] N. Garcia, "Javascript port scanner," http://jsscan.sourceforge.net/.

[11] J. Grossman and T. Niedzialkowski, "Hacking intranet websites from the outside: JavaScript malware just got a lot more dangerous," in *Blackhat USA*, 2006.

[12] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks – stealing the pie without touching the sill," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[13] I. Hickson, "5.6 offline web applications – HTML5," http://www.w3.org/TR/2011/WD-html5-20110525/offline.html, 2011.

[14] ——, "6.7 offline web applications – HTML standard," http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html, 2013.

[15] J. Hodges, C. Jackson, and A. Barth, "HTTP strict transport security (HSTS)," Internet Requests for Comments, RFC 6797, 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6797.txt

[16] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, "Clickjacking: Attacks and defenses," in *USENIX Security Symposium*, 2012.

[17] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson, "Protecting browsers from cross-origin CSS attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[18] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from DNS rebinding attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[19] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting browser state from web privacy attacks," in *International World Wide Web Conference (WWW)*, 2006.

[20] M. Jakobsson and S. Stamm, "Invasive browser sniffing and countermeasures," in *International World Wide Web Conference (WWW)*, 2006.

[21] Y. Jia, X. Dong, Z. Liang, and P. Saxena, "I know where you've been: Geo-inference attacks via the browser cache," in *Web 2.0 Security & Privacy (W2SP)*, 2014.

[22] M. Johns, S. Lekies, and B. Stock, "Eradicating DNS rebinding with the extended same-origin policy," in *USENIX Security Symposium*, 2013.

[23] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attacks using CSS filters," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[24] L. Kuppan, "JS-Recon - HTML5 based JavaScript network reconnaissance tool," http://www.andlabs.org/tools/jsrecon.html.

[25] ——, "Chrome and Safari users open to stealth HTML5 AppCache attack," http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html, 2010.

[26] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis, "Puppetnets: Misusing web browsers as a distributed attack infrastructure," in *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[27] B. Liang, W. You, L. Liu, W. Shi, and M. Heiderich, "Scriptless timing attacks on web browser privacy," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[28] G. Maone, "NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!" http://noscript.net.

[29] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: A study of clickjacking vulnerabilities on popular sites," in *Web 2.0 Security & Privacy (W2SP)*, 2010.

[30] S. Stamm, Z. Ramzan, and M. Jakobsson, "Drive-by pharming," in *International Conference on Information and Communications Security (ICICS)*, 2007.

[31] A. van Kesteren, "Fetch standard," http://fetch.spec.whatwg.org, 2014.

[32] ——, "Cross-origin resource sharing," http://www.w3.org/TR/cors/, 2013.

[33] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Network and Distributed System Security Symposium (NDSS)*, 2007.

[34] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

APPENDIX A
URLS OF INTERNAL WEB SERVERS

We introduce some URLs related to web-based administration pages of networked devices that can reveal the kinds of devices located in the local network of a victim web browser. First, many wireless routers have default IP addresses for serving web-based administration pages. For example, default IP addresses for wireless routers are 192.168.0.1 for Netgear, 192.168.1.1 for ASUS, and 192.168.2.1 for Belkin [3]. We can infer the router that a victim web browser uses by probing such IP addresses. Second, many networked devices with web-based administration pages have some unique URLs that can be used to recognize them. For example, an HP network printer has a jQuery library under `/hp/device/Scripts/jquery/jquery.js` and a Buffalo NAS has a CSS file under `/static/ext/resources/css/ext-all.css`. Using such information, attackers can know the networked devices that a victim user currently uses.

APPENDIX B
APPCACHE-BASED URL TIMING ATTACK

We can exploit AppCache not only to conduct a URL status identification attack, but also to conduct a URL timing attack. A standard behavior of AppCache makes a URL timing attack possible: to avoid content inconsistency problems, AppCache should check whether a manifest changes while downloading the corresponding URLs (Section III-B).

```
var appCache = window.applicationCache;
var downloadStart;

function handleDownloading(e) {
    // start to download a given URL
    downloadStart = new Date();
}

function handleCached(e) {
    // succeed to download a given URL
    var downloadEnd = new Date();
    var img = new Image();
    img.src = "/results.png?time=" + (
        downloadEnd - downloadStart);
}

appCache.addEventListener('downloading',
    handleDownloading, false);
appCache.addEventListener('cached',
    handleCached, false);
appCache.addEventListener('updateready',
    handleCached, false);
```

Listing 7. Script-based URL timing attack using AppCache.

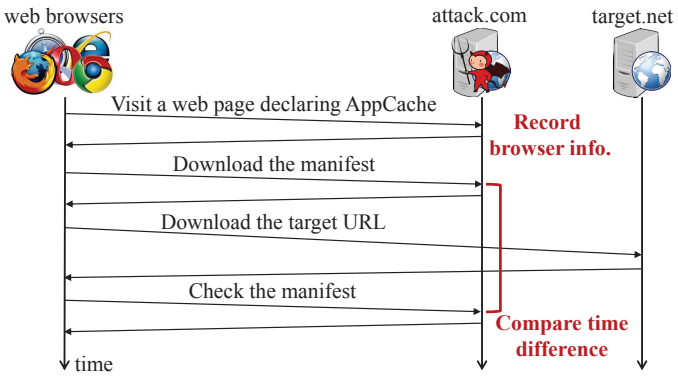

Fig. 7. Scriptless URL timing attack using AppCache.

A. Attack Manifest

An AppCache manifest to perform a URL timing attack is the same as in Section IV-A.

B. Script-based Attack

Listing 7 shows a JavaScript code that uses AppCache to perform the URL timing attack. A web browser fires (1) a `downloading` event when it starts to download resources in a manifest and (2) either `cached` or `updateready` events when it successfully downloads all the resources (Section III-B). Our attack manifest specifies a single target URL so that we indirectly time the URL by comparing when the two events occur. We add event handlers to an AppCache object to notify an attack web application that the events have occurred.

C. Scriptless Attack

The overall procedure of our scriptless URL timing attack (Fig. 7) involves five steps. First, a victim web browser visits a web page of an attack web application and recognizes that the web page declares an AppCache manifest. The attack web application records the information of the victim web browser

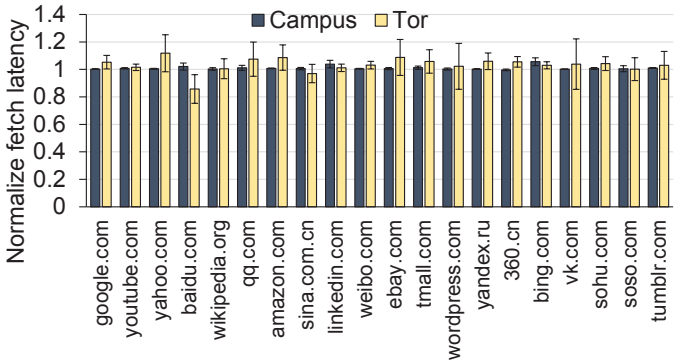

Fig. 8. Page fetch latency of scriptless URL timing attacks normalized to that of script-based attacks. A victim web browser (Chrome) accessed an attack web server via either a campus network or Tor. We repeated each experiment 10 times and report the average results.

(e.g., IP address and user-agent string) in its database. Second, the victim web browser downloads an attack manifest file and extracts a target URL from the manifest. The attack web application also records the current time in its database. Third, the victim web browser downloads the target URL specified in the manifest. Fourth, the victim web browser re-downloads the manifest file to check whether the manifest changes. Lastly, the attack web application compares the current time with the recorded time to infer how much time the victim web browser spends to fetch the target URL.

D. Effectiveness of Scriptless Attack

Unlike the script-based attack, which is performed within a victim web browser, network latency between a victim web browser and an attack web server affects the scriptless attack. To identify the effects of the network latency on the scriptless attack, we compared both attacks' page fetch latency. We chose URLs included in 20 front pages of Alexa Top domains (e.g., CSSs, images, and JavaScript codes) and measured how much time a victim web browser (Chrome on Windows 8.1) spends to fetch the URLs by using the script-based and scriptless attacks. When choosing the 20 domains, we ignored some regional domains (e.g., `google.ca` and `yahoo.co.jp`) and domains whose URLs used a `no-store` directive (e.g., Facebook and Twitter). We also used Tor [7] to increase the network latency between the victim web browser and the attack web server because they were on the same campus network. When evaluating each URL via Tor, we encountered at least three different exit nodes located in different countries. We performed the Tor-based experiments immediately after the campus-network-based experiments to avoid changes in the front pages.

Fig. 8 shows the normalized page fetch latency of scriptless attacks to that of script-based attacks. We conclude that the scriptless attack is as effective as the script-based attack due to the acceptable latency differences between the attacks even when the victim web browser used Tor. Average differences were 0.7% and 3.5% when the victim web browser accesses the attack web server via our campus network and Tor, respectively. Therefore, we are convinced that we can use the scriptless attack instead of the script-based attack with tolerable errors.