Exploring and Mitigating Privacy Threats of HTML5 Geolocation API [*]

Hyungsub Kim
Dept. of CSE, POSTECH
Pohang, Korea
hyungsubkim@postech.ac.kr

Sangho Lee
Dept. of CSE, POSTECH
Pohang, Korea
sangho2@postech.ac.kr

Jong Kim
Dept. of CSE, POSTECH
Pohang, Korea
jkim@postech.ac.kr

ABSTRACT

The HTML5 Geolocation API realizes location-based services via the Web by granting web sites the geographical location information of user devices. However, the Geolocation API can violate a user's location privacy due to its coarse-grained permission and location models. The API provides either exact location or nothing to web sites even when they only require approximate location. In this paper, we first conduct case studies on numerous web browsers and web sites to explore how they implement and utilize the Geolocation API. We detect 14 vulnerable web browsers and 603 overprivileged web sites that can violate a user's location privacy. To mitigate the privacy threats of the Geolocation API, we propose a novel scheme that (1) supports fine-grained permission and location models, and (2) recommends appropriate privacy settings to each user by inspecting the location sensitivity of each web page. Our scheme can accurately estimate each web page's necessary geolocation degree (estimation accuracy: ∼93.5%). We further provide suggestions to improve the Geolocation API.

1. INTRODUCTION

Location-based services (LBSs) are popular personalized services that are tightly associated with user privacy. Examples of LBSs include navigation services, local search services, traffic alert services, and localized weather services, which are especially useful for mobile device users. However, without assured privacy, users may not trust LBSs. Therefore, following the seminal work of Gruteser and Grunwald [13], numerous researchers propose various schemes [5, 7–9, 20, 23, 31, 32] to ensure location privacy in LBSs.

LBSs via the Web have become necessary because the number of mobile devices that access the Web has increased. The HTML5 specification satisfies such requirements by defining the Geolocation API [29] which grants permission to access the geographical location information (*geolocation*) of devices to web sites. The API consists of two methods: `GetCurrentPosition()`, which retrieve current geolocation including latitude and longitude; and `watchPosition()`, which tracks updated position according to user movements. The accuracy of the geolocation depends on the type of user device. For example, smart mobile devices allow web browsers to precisely estimate the device's geolocation by using various sources, such as the global positioning system (GPS), cell towers, and Wi-Fi access points (APs).

However, we identify that the current specification and implementations of the Geolocation API face *four privacy threats*, which are especially harmful to mobile device users along with exact geolocation. (1) They employ *no fine-grained location model*. Even if a web site only wants to know which country a user is in, both the web site and the user cannot decrease geolocation accuracy when requesting or delivering it. Furthermore, the web site should perform *geocoding* to obtain country information from the received geolocation. (2) They employ *no per-method permission model*. A user must allow a web site to track his or her movements even when the user only wants to allow the site to retrieve the current geolocation. (3) They employ a *per-domain permission model* without a *per-page permission model*. A user cannot allow a web page `site.com/map.html` to access the geolocation while disallowing another web page under the same domain `site.com/mail.html` to access the geolocation. (4) They employ *no re-confirm process* for changed web pages. A web site preserves its permissions to access the geolocation, even though it modifies or deletes the original web page that a user has permitted. We must solve these privacy problems to ensure the location privacy of users in LBSs via the Web.

In this paper, we first explore real-world privacy problems of the Geolocation API due to *vulnerable web browsers* and *overprivileged web sites*. We analyze a number of web browsers for popular mobile platforms (Android and iOS) to inspect (1) the number of web browsers that support Geolocation API, (2) the number of vulnerable web browsers that allow any web site to access the geolocation *without user permissions*, and (3) their interfaces to grant or revoke permissions. We have detected 14 vulnerable web browsers installed on *more than 16 million Android devices* via Google Play Store, and have reported the security problem to the browser developers.

We also analyze 1,196 web sites that use the Geolocation API to identify their characteristics and the degree of geolocation that they demand. We discover that *approximately half of the web sites using the Geolocation API are overprivileged*, i.e., they are irrelevant to precise geolocation, because their content does not much change even when we provide completely different geolocation.

Next, we propose a novel scheme for mitigating the privacy problems of the Geolocation API by supporting fine-grained permission/location models and measuring each web page's location sensitivity. We modify an open-source web browser for Android to (1) grant different degrees of geolocation to different domains/web pages, (2) separate permissions for location tracking, (3) support both per-domain and per-page permission models, and (4) inspect

[*]This work was supported by ICT R&D program of MSIP/IITP. [14-824-09-013, Resilient Cyber-Physical Systems Research]

ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12$15.00
http://dx.doi.org/10.1145/2664243.2664247

web page changes and re-confirm permissions for changed web pages. Furthermore, our web browser can estimate a web page's location sensitivity and use this estimate to recommend a privacy setting to its user.

Evaluation results show that our scheme has low overhead while precisely estimating each web page's necessary geolocation degree (∼93.5% of estimation accuracy).

This paper makes the following contributions:

- **In-depth analysis.** To the best of our knowledge, this is the first study that analyzes the privacy problems of the HTML5 Geolocation API in depth.
- **New case study.** We conduct the first case studies on the Geolocation API regarding various web browsers and web sites, and discover a number of vulnerable web browsers and overprivileged web sites.
- **Effective countermeasure.** We propose effective countermeasures against the Geolocation API's privacy problems by modifying a web browser to support fine-grained permission/location models and inspect a web page's location sensitivity while considering portability and compatibility.
- **Reasonable suggestions.** We offer some suggestions to improve the Geolocation API, such as accuracy options, per-method permissions, and per-page permissions.

The remainder of this paper is organized as follows. §2 explains the HTML5 Geolocation API. §3 conducts case studies on web browsers and web sites with the HTML5 Geolocation API. §4 introduces the threat model and assumptions of this work. §5 explains our countermeasure in details. §6 discusses the limitations of this work and our suggestions to reduce the privacy threats of the Geolocation API. §7 introduces related work. Lastly, §8 concludes this work.

2. BACKGROUND

In this section, we briefly explain the current localization technologies, the Geolocation API of HTML5, and privacy concerns residing in the specification of the Geolocation API.

2.1 Localization Technologies

Smart mobile devices estimate their geographical location by using the global positioning system (GPS), cell-tower triangulation, and Wi-Fi access point (AP) triangulation [11, 14]. The GPS provides highly accurate latitude, longitude, altitude, heading, and speed information to the devices. Its accuracy can be up to several meters. The GPS, however, has two problems: (1) it takes time to initialize the communication between a GPS receiver and GPS satellites; and (2) it usually does not work indoors because GPS signals are difficult to penetrate roofs, walls, and other objects [21]. To overcome these problems, smart mobile devices also use cell-tower-based or Wi-Fi-based triangulation, or both to estimate their latitude and longitude information. Although the triangulation immediately returns location information and supports indoor positioning, its accuracy can be up to several hundreds (Wi-Fi) or thousands (cell tower) of meters. Its coverage and accuracy also depend on whether LBS providers (e.g., Google) correctly and widely collect the location information of cell towers and Wi-Fi APs.

2.2 HTML5 Geolocation API

The HTML5 specification defines the Geolocation API [29] that allows web sites to access the geolocation of user devices. This API consists of two methods. (1) `getCurrentPosition()` inspects geolocation of user devices including latitude, longitude, accuracy, altitude, heading, speed, and timestamp. (2) `watchPosition()` continually retrieves the current geolocation of user devices according to user movements. Both methods have a mandatory parameter

```
var map=document.getElementById("map");

function getLocation() {
❶  if (navigator.geolocation) {
❷    navigator.geolocation.getCurrentPosition(
       showPosition);
    }
    else {
      map.innerHTML="This browser does not support
          HTML5 Geolocation API.";
    }
}

function showPosition(position) {
❸  var latlon=position.coords.latitude+","+
      position.coords.longitude;

❹  var img_url="http://maps.googleapis.com/maps/
      api/staticmap?center="+latlon+"&zoom=13&size
      =500x400&maptype=hybrid&sensor=true";

❺  map.innerHTML="<img src='"+img_url+"'>";
}
```

Figure 1: JavaScript code to obtain a Google map image based on the current geolocation. It inspects whether a web browser supports the Geolocation API (❶), obtains the geolocation (❷), parses the geolocation (❸), receives a map-image URL from the Google map (❹), and displays the map image (❺).

`PositionCallback` that specifies a callback function to execute when they successfully obtain the geolocation.

Figure 1 shows a JavaScript code that uses the `getCurrentPosition()` method to check a user's current geolocation on the Google map (we refer [30].) The code first inspects whether a web browser supports the Geolocation API (❶). If the web browser supports the Geolocation API, the code calls the `getCurrentPosition()` method to obtain the current geolocation of the user (❷). When the method successfully retrieves the geolocation, it calls a callback function `showPosition()`. The callback function reads the current geolocation in an argument `position` (❸), receives a map-image URL from the Google map by using the geolocation (❹), and embeds the image URL in HTML content (❺). In addition, if we use the `watchPosition()` method instead of the `getCurrentPosition()` method, we can easily track the geolocation changes of the user.

The methods of the Geolocation API have two optional parameters. (1) `PositionErrorCallback` specifies a callback function to execute when they fail to obtain the geolocation. (2) `PositionOptions` represents a JavaScript object consisting of three attributes: `enableHighAccuracy` represents whether a web page prefers the best possible results (i.e., it demands GPS-based geolocation,) `timeout` represents the amount of time that the web page waits for receiving the geolocation, and `maximumAge` indicates the validity period of cached geolocation data. The default value of `enableHighAccuracy` is `false` to reduce power consumption due to GPS receivers, and the default values of `timeout` and `maximumAge` are "Infinity".

2.3 Privacy Concerns of Geolocation API

The Geolocation API specification [29] regulates some requirements of both web browsers and web sites to preserve user privacy. First, web browsers need to obtain permissions from users for each web site when it attempts to utilize the Geolocation API. Web sites can use the geolocation information only when the users grant per-

Table 1: How the 60 Android web browsers support the Geolocation API.

Geolocation permission	Number
Permanent & temporary	18
Permanent only	7
No permission check	14
Not available	21

Table 2: Vulnerable Android web browsers that do not ask for the Geolocation permissions.

Name	Version	#Downloads
Baidu Browser	4.1.0.3	10,000,000+
Maxthon Browser for Android	4.3.0.2000	5,000,000+
Angel Browser	12.30z	500,000+
Maxthon Web Browser for Tablet	4.0.4.1000	500,000+
Exsoul Web Browser	3.3.3	100,000+
Full Screen Browser	2.3	100,000+
Harley Browser	1.3.4	100,000+
Maxthon Browser for Pioneer	2.7.3.1000	100,000+
Safe Browser - The Web Filter	1.2.5	100,000+
Baidu Browser for Tablet	1.3.0.2	100,000+
Habit Browser	1.1.25	100,000+
Browser Omega	2.6.1	50,000+
Jelly Web Browser	1.1.4	10,000+
Zomi Mobile Browser	2.6.6	10,000+
Total		16,770,000+

missions. Second, the web browsers have to provide a user-friendly interface to revoke the granted permissions.

The specification also mentions other privacy problems in the non-normative section: (1) users may inadvertently grant permissions to web sites, and (2) already-permitted web sites can silently change their content regardless of a user's intention.

However, the specification mentions *no countermeasures* against these problems. It leaves to implementers the responsibility of solving the problems.

3. CASE STUDIES

In this section, we conduct case studies on the HTML5 Geolocation API to identify how web browsers and web sites implement and utilize it. The case studies are helpful to explore the real-world privacy problems of the Geolocation API.

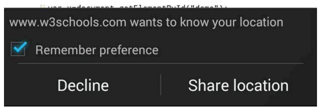
3.1 Case Study of Web Browsers

3.1.1 Android

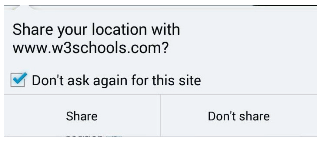
We collect and analyze 60 free web browsers for Android on Google Play Store in August 2014, which had been installed more than 10,000 times. Except some major web browsers (e.g., Chrome and Firefox), most web browsers for Android rely on WebView [2] which allows Android Apps to embed a customized web browser. We use a Galaxy S III (Android 4.3) and a Galaxy Nexus (Android 4.2) when testing the web browsers.

We first investigate how the 60 Android web browsers support the Geolocation API (Table 1). We detect 39 web browsers that support the Geolocation API. They consist of 18 web browsers that support both permanent and temporary (one time) permissions, 7 web browsers that only support permanent permissions, and 14 web browsers that *do not ask for user permissions*.

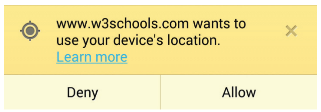
More than 16 million Android users have installed the 14 vulnerable web browsers that do not ask for permission (Table 2). Their actual number is certainly *larger* than 16 million because the number


www.w3schools.com wants to know your location

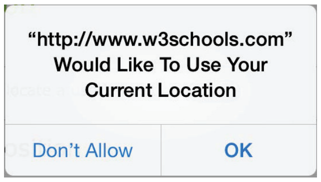
(a) Stock Android browser (4.2).


Share your location with www.w3schools.com?

(b) Android (Firefox 27).


www.w3schools.com wants to use your device's location.

(c) Android (Chrome 32).


"http://www.w3schools.com" Would Like To Use Your Current Location

(d) iOS 7.

Figure 2: Dialogs for requesting permissions to access geolocation.

of downloads shown in Google Play Store is a lower bound (Google does not reveal the exact number of downloads.) For example, "5,000,000+ downloads" means that the actual number of downloads is between five million and 10 million. Furthermore, Android users may install the vulnerable web browsers via alternative markets or by using Android application package (APK) files. If users visit web sites with the vulnerable browsers, the sites can silently collect the users' geolocation by exploiting the browsers. Therefore, we suggest that the developers of the vulnerable web browsers need to fix the problem (details: §3.2).

Figures 2a, 2b, and 2c show permission dialogs of the major web browsers for Android when users attempt to access a web page that uses the Geolocation API [30]. All of the web browsers give permissions not to web pages but to domains. Furthermore, the stock Android browser (Figure 2a) and Firefox for Android (Figure 2b) support temporary permissions for accessing geolocation whereas Chrome for Android does not (Figure 2c).

Lastly, we investigate the interfaces to determine which of them allow users to revoke the granted permissions stored in the Android web browsers. Among the 25 web browsers that support the Geolocation API and ask for user permission, only six *Android Browser, Chrome Browser, Firefox, iLunascape 2 - Web Browser, Sleipnir Mobile - Web Browser, and Skyfire Web Browser 5.0* allow users to revoke the granted permissions of each domain. With the remaining 19 web browsers, users must delete all granted permissions even if they only want to revoke the permission of a single domain.

3.1.2 iOS

We collect and analyze the top 30 free web browsers for iOS on App Store (e.g., Chrome and Mercury Browser) as of September 2013, and identify that most of the web browsers for iOS use the *same* mechanism to manage the Geolocation API and permissions. The reason is that Apple forces developers to use UIWebView [3] when rendering web pages to harden security. Exceptions are cloud-based web browsers (e.g., Opera Mini and Puffin) that have no JavaScript engine, but they do not support the Geolocation API. We use iPad (third generation with iOS 7) when testing the browsers.

We confirm that the web browsers for iOS only have the permanent permission model for the Geolocation API. Figure 2d shows an example of their permission dialog which requests a permanent geolocation permission for a domain.

Lastly, we identify that users should use the unified interface of iOS to reset *all* location and privacy permission settings when they want to revoke the geolocation permissions granted to web sites. After resetting, all apps and web sites must obtain permissions again to access both geolocation and other private information (e.g., calendars, reminders, and photos). Undoubtedly, this is highly

```
.method public
  onGeolocationPermissionsShowPrompt(
    Ljava/lang/String;
    Landroid/webkit/
      GeolocationPermissions$Callback;)V

    .locals 2
    .parameter
    .parameter

    .prologue
    .line 225
    const/4 v0, 0x1
    const/4 v1, 0x0

➠    invoke-interface {p2, p1, v0, v1},
      Landroid/webkit/
        GeolocationPermissions$Callback;
        ->invoke(Ljava/lang/String;ZZ)V

    .line 227
    return-void
.end method
```

Figure 3: Decompiled `onGeolocationPermissionsShowPrompt()` of the Maxthon Browser for Android.

inconvenient for users.

3.2 Details of Vulnerable Web Browsers

We analyze the vulnerable Android web browsers to know why they do not prompt geolocation permission dialogs and finally detect that this flaw is due to mis-implemented `onGeolocationPermissionsShowPrompt()` methods. The `onGeolocationPermissionsShowPrompt()` method of the `WebChromeClient` class is essential to support the Geolocation API in WebView-based browsers because they call the method when an unseen web site attempts to use the Geolocation API [1]. This method should invoke a callback method to set permissions with three parameters: (1) a domain name, (2) whether a user allows (`true`) or blocks (`false`) the domain, and (3) whether the granted permission is permanent (`true`) or temporary (`false`). However, if the method always invokes the callback method with `true` as the second argument, web browsers always allow any web site to access the geolocation without user permissions. Accordingly, we expect that the `onGeolocationPermissionsShowPrompt()` method of the vulnerable browsers always invokes the callback method with `true`.

Figure 3 shows the decompiled `onGeolocationPermissionsShowPrompt()` method of the Maxthon Browser for Android with `apktool` (the results of other browsers are similar.) As we expect, the method has no instructions to pop up a permission dialog and invokes a callback method while statically assigning `true` (`0x1`) to the second argument `v0` (➠, `p2` is the 0th argument representing the callback method.)

Consequently, we believe that developers need to carefully implement `onGeolocationPermissionsShowPrompt()` methods and Google has to provide a built-in permission dialog to eliminate such a vulnerability.

We reported the security problem to the browser developers. Some of them replied that they would patch it in a future release.

3.3 Case Study of Web Sites

We collect 1196 web pages that use the Geolocation API; to do this we (1) inspect web sites listed on Alexa, (2) use the Google search engine with keywords, such as "near me" and "around me", and (3) use an HTML code search engine [10] with keywords, such as "getCurrentPosition" and "watchPosition", between August 2013 and September 2013, then inspect them. Each of the three sources contributes 246, 140, and 810 web pages, respectively. When we visit web sites, we use user-agent strings of Android or iPhone web browsers and recursively retrieve child web pages. We manually verify that most of the collected web pages provide "near me" services to inform point of interest (POI) locations (e.g., stores, buildings, and the sights) or local information (e.g., local radio and TV channels)[1]. Table 3 summarizes the results. We also find other web pages, including those for local weather services, geographic information, location-based online social networks, local traffic information, and local news.

Table 3: Categories of web pages using the Geolocation API.

Category	Number	%
Near me	667	55.77
Local information	288	24.08
Weather	55	4.60
Geographic information	53	4.43
Social networking	42	3.51
Traffic information	32	2.68
News	13	1.09
Others	46	3.85

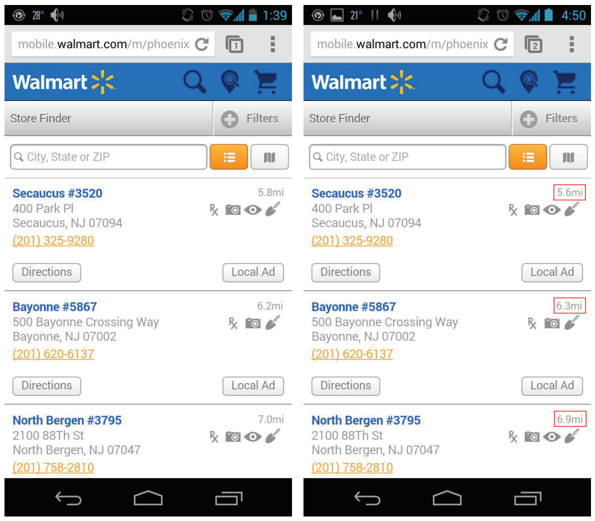
Table 4: Location sensitivity of web pages using the Geolocation API.

Sensitivity	Number	%
Pinpoint	593	49.58
City	426	35.62
State	22	1.84
Country	18	1.51
Unchanged	137	11.45

We aim to inspect the location sensitivity of the collected web pages to decide whether they are overprivileged. We manually perform the following procedure: (1) preparing a number of GPS coordinates around famous cities, (2) visiting the web pages while using various geolocations based on the GPS coordinates, and (3) verifying whether the web pages change according to the given geolocation. We treat street-level changes are equivalent to pinpoint-level changes because both levels are sufficiently fine grain. We thereby compare them for verifying pinpoint-, city-, and state-level geolocation changes, respectively. We also use locations around famous cities of other countries, such as Calgary (Canada), Paris (France), Seoul (Korea), and Sydney (Australia), to analyze whether some web pages work for a specific country. When visiting the web pages, we use Developer Tools (Chrome) or User Agent Switcher and Geolocator extensions (Firefox) to change the user agent strings and geolocation of web browsers.

Our inspection of the location sensitivity of the collected web pages reveals that *half of them do not need to use exact geolocation* (Table 4). Except for 49.6% of the web pages that demand pinpoint geolocation, other web pages provide city-, state-, or country-level information (35.6%, 1.8%, and 1.5%, respectively). For example, Groupon's web page for nearby deals provides city-level information; it surely does not need to obtain exact geolocation. Interestingly, the content of 137 web pages (11.5%) does not change even when we alter the geolocation drastically. We confirm that they unnecessarily demand geolocation by manually investigating them.

[1]Five graduate students participated in manual inspection. Decisions were made by majority voting.



mobile.walmart.com/m/phoenix

Walmart

Store Finder — Filters

City, State or ZIP

Secaucus #3520 400 Park Pl Secaucus, NJ 07094 (201) 325-9280	5.8mi
Directions	Local Ad
Bayonne #5867 500 Bayonne Crossing Way Bayonne, NJ 07002 (201) 620-6137	6.2mi
Directions	Local Ad
North Bergen #3795 2100 88Th St North Bergen, NJ 07047 (201) 758-2810	7.0mi

mobile.walmart.com/m/phoenix

Walmart

Store Finder — Filters

City, State or ZIP

Secaucus #3520 400 Park Pl Secaucus, NJ 07094 (201) 325-9280	5.6mi
Directions	Local Ad
Bayonne #5867 500 Bayonne Crossing Way Bayonne, NJ 07002 (201) 620-6137	6.3mi
Directions	Local Ad
North Bergen #3795 2100 88Th St North Bergen, NJ 07047 (201) 758-2810	6.9mi

(a) At a latitude of 40.71365 and a longitude of -74.00971. (b) At a latitude of 40.71500 and a longitude of -74.01000.

Figure 4: Mobile web pages of Walmart to find the nearest stores from two places in New York.

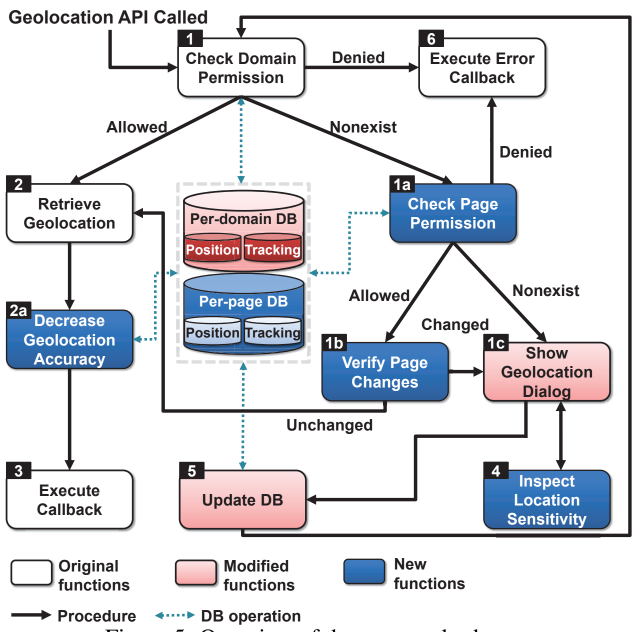


Figure 5: Overview of the proposed scheme.

3.4 Considerations on "Near Me" Services

Even though the "near me" services include notable LBSs that use exact geolocation, we expect that many of these services can provide the same functionalities without relying on such exact geolocation. As an example, we consider a "near me" service web page of Walmart to find the nearest stores as an example (`http://mobile.walmart.com/m/phoenix#location/locate`). When we visit the web page with two slightly different GPS coordinates, latitudes and longitudes are (40.71365, -74.00971) and (40.71500, -74.01000). Walmart redirects us to two slightly different web pages (Figures 4a and 4b). Therefore, the "near me" service web page demands exact geolocation. The URLs of the final web pages are `http://mobile.walmart.com/m/phoenix#location/list/40.713/-74.009` and `http://mobile.walmart.com/m/phoenix#location/list/40.715/-74.010`, respectively, so Walmart can know the GPS coordinates of visitors from GET request parameters.

However, Walmart can use the city-level geolocation for providing the same service to mitigate privacy threats while reducing computational overhead. They can provide this service by (1) delivering a list of GPS-coordinates of its stores in New York to web browsers and (2) allowing the web browsers to execute a JavaScript code to calculate the distances between the stores and the user's current geolocation. Walmart does not need to compute and sort the distances, so it can reduce computational overhead especially when the number of concurrent users is large. Although each browser needs to perform some computations instead of Walmart, it can protect exact geolocation from Walmart. Consequently, we believe that this approach is good for both web sites and their users.

4. THREAT MODEL AND ASSUMPTIONS

In this section, we explain the threat model and assumptions of this work before introducing our scheme for mitigating the privacy threats of the HTML5 Geolocation API.

First, we assume that our attackers are *honest-but-curious* LBS providers, who use the Geolocation API in a *legitimate but overprivileged* way. They attempt to obtain precise geolocation of users even when their LBSs demand neither precise nor timely geolocation. Therefore, we focus on how to restrict the privilege of web sites in a fine-grained way.

Second, we assume that the attackers try to *implicitly* access the geolocation *without user interaction*. In the case studies, we discover some web sites that explicitly demand user interaction whenever the sites access the geolocation (e.g., users should click a button or type some search keywords to use LBSs.) Such user interaction reminds users that they are granting their geolocation to the web sites; they may deny it at that time to ensure privacy. Consequently, the attackers would not demand such explicit interaction when requesting geolocation, so we focus on geolocation requests without user interaction.

Lastly, we assume that both the web sites providing LBSs and the web browsers visiting the web sites have *no vulnerability* that attackers can exploit. External attackers cannot inject a malicious script into vulnerable web pages or exploit vulnerable web browsers to access geolocation. For this reason, we do not consider solutions to detect script injections and to mitigate vulnerabilities because they are out of the scope of this paper.

5. PROPOSED SCHEME

In this section, we explain the proposed scheme for mitigating the Geolocation API's privacy threats. We implement the proposed scheme on Android because it is popular and allows custom web browsers unlike iOS.

5.1 Overall Design

We explain the overall design of the proposed scheme. When a user visits a web page that contains a JavaScript code to execute the Geolocation API, the proposed scheme handles the execution as follows (Figure 5).

1. The proposed scheme checks whether the *domain* serving the code exists in a *per-domain* permission database (DB). If the domain exists in the DB and the user allows the domain, the scheme goes to Step 2. If the domain does not exist in the DB, it goes to Step 1a. If the user denies the domain, it goes to Step 6.

   1a. The proposed scheme checks whether the *web page* serving the code exists in a *per-page* permission DB. If the web page exists in the DB and the user allows the web page, the scheme goes to Step 1b. If the web page does not exist in the DB, it goes to Step 1c. If the user denies the web page, it goes to Step 6.

1b. The proposed scheme verifies whether the web page *changes* after obtaining permissions. If the degree of changes exceeds a threshold value, it goes to Step 1c, otherwise, it goes to Step 2.

1c. The proposed scheme composes an enhanced *Geolocation permission dialog* which asks the user to (1) either allow or deny the domain/web page, (2) inspect the location sensitivity of web page, (3) choose the geolocation accuracy, and (4) grant either temporary or permanent permissions. It then goes to either Step 4 or Step 5 according to the user's choices.

2. The proposed scheme retrieves precise geolocation from the OS (Android) and goes to Step 2a.

   2a. The proposed scheme *decreases* geolocation accuracy according to the *allowed geolocation accuracy* of the domain/web page stored in per-domain/per-page DBs. It then goes to Step 3.

3. The proposed scheme terminates while executing a *success* callback function with (less accurate) geolocation. The web page's JavaScript code handles the remaining procedures such as composing an HTML document and rendering.
4. The proposed scheme inspects the *location sensitivity* of the web page to estimate the necessary geolocation degree. When composing a Geolocation permission dialog, it uses the sensitivity information. It then returns to Step 1c.
5. The proposed scheme *updates* DBs according to a user's choices and goes to Step 1 to *re-initiate* the procedures.
6. The proposed scheme terminates while executing an *error* callback function when the user denies the domain/web page demanding geolocation.

We respectively apply the explained procedures to `GetCurrentPosition()` and `watchPosition()` as we separate the permissions for position and tracking.

5.2 Inspecting Geolocation Web Pages

We explain how our web browser verifies the changes in a web page that uses the Geolocation API and measures the location sensitivity of the web page.

5.2.1 Verifying web page changes

We verify the changes of web pages that have geolocation permissions because users may want to revoke the granted permissions if they do not prefer the changed web pages or the web pages no longer contain location-based content. For this goal, we need a method to effectively identify the differences between the old and current versions of web pages.

We use a *context triggered piecewise hash algorithm* [19], also known as a *fuzzy hash algorithm*, to verify changes. This algorithm divides a document into blocks based on triggers, computes hash values of each block, and uses the list of hash values to compare different documents. We use this algorithm to confirm changes by comparing the old and current versions of a rendered web page. For image files, we use their histogram values and file sizes for composing hash values.

We explain the procedure to verify web page changes. When our web browser visits a web page for the first time and a user allows the web page to retrieve geolocation, the browser computes a hash value of the web page and stores the hash value along with the current geolocation in its DB. On subsequent visits, the browser additionally opens the web page with the stored geolocation to compute a new hash value of the web page. If the difference between the stored and new hash values exceeds a threshold value, the browser may ask for user permissions again.

When computing the hash value of a web page, our web browser ignores *dynamic content* embedded in the page (e.g., a web banner) because it frequently changes regardless of whether the main text of the web page changes. The browser identifies such dynamic content by (1) visiting a web page several times in a short time period and (2) inspecting changed content over the visits. Finally, the browser computes hash values while excluding the dynamic content.

To lighten its burden, the web browser keeps a time-stamp of the last visit to the web page and verifies changes only when the elapsed time from the last visit is above a certain time.

5.2.2 Estimating location sensitivity

Our web browser estimates the location sensitivity of web pages by varying geolocation within a predefined set of GPS coordinates (§3.3) and verifying web page changes (§5.2.1). First, the web browser extracts *static content* from a web page (i.e., excluding banners). Second, the browser visits the web page several times while using five different addresses that differ in street, city, state, and country, respectively. Third, the browser verifies the changes of the rendered web pages to estimate location sensitivity. Lastly, the browser announces the estimated location sensitivity to a user for recommending privacy configurations. If the browser fails to estimate the location sensitivity because web pages have location-independent content, the browser notifies *random* to a user. Visiting a web page multiple times to enhance privacy is not a new idea. [18,27] have already considered similar techniques.

5.2.3 Concurrent inspection

Our web browser should visit a web page several times for inspection, so a user may need to wait a long time to check the inspection result. We reduce the waiting time by *concurrently inspecting* a web page. First, the browser simultaneously creates five WebView activities for visiting a single web page with five different addresses that differ in street, city, state, and country, respectively. Next, the browser creates another WebView activity for visiting the web page with one of the five addresses and extracts static content from the two rendered web pages of the same address. Lastly, the browser compares the static content with the five rendered web pages of the different addresses, respectively, to estimate location sensitivity. The overall inspection time is 1.8 times longer than page loading time (details: §5.4.2).

5.3 Managing and Serving Geolocation

We explain how our web browser manages geolocation and serves it to web pages. Our goal is to develop a *portable* and *compatible* solution to the privacy problem of the Geolocation API. Our method, *overriding the Geolocation API*, meets the goal because it modifies neither Android platforms nor JavaScript engines.

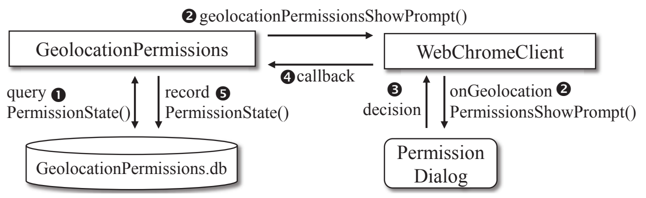
5.3.1 Modifying the Geolocation API

To implement our scheme, we modify the behavior of the Geolocation API by overriding its JavaScript methods. We choose an open-source web browser for Android, Lighting Browser [25], relying on WebView [2]. WebView supports JavaScript and allows developers to inject arbitrary JavaScript codes into a loaded web page; we use these features to override the Geolocation API (we discuss possible limitations of this approach in §6.1.)

Figure 6 shows a code snippet to change the behavior of the

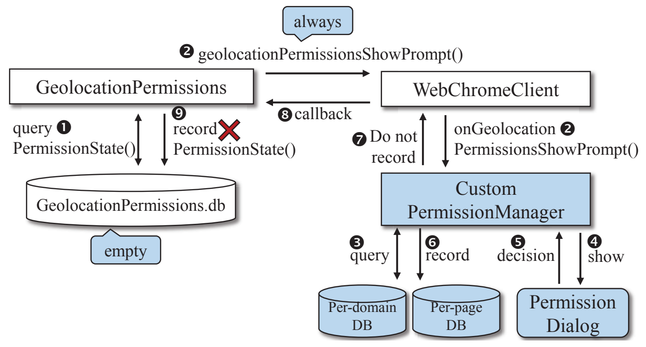
```
WebView webview = (WebView) findViewById(R.id.
    webview);

❶webview.getSettings().setJavaScriptEnabled(true);
webview.getSettings().setGeolocationEnabled(true)
    ;

❷webview.loadUrl("javascript:
  navigator.geolocation.getCurrentPosition
    = function(success) {
        success( {
❸         coords:{
            latitude:"+dLatitude+",
            longitude:"+dLongitude+",
          },
          timestamp:Date.now()
        });
      }"
);

❹webview.loadUrl(URL);
```

Figure 6: Java code to override the `getCurrentPosition()` method using WebView. It enables JavaScript and the Geolocation API (❶), overrides `getCurrentPosition()` (❷), changes location information (❸), and reloads the current web page (❹).



❷ geolocationPermissionsShowPrompt()
GeolocationPermissions
WebChromeClient
❹callback
query ❶ PermissionState()
record ❺ PermissionState()
❸ decision
onGeolocation ❷ PermissionsShowPrompt()
GeolocationPermissions.db
Permission Dialog

(a) Default manager.



(b) Custom manager.

Figure 7: Procedures of default and custom geolocation permission managers.

`getCurrentPosition()` method. First, the code enables both JavaScript and the Geolocation API in WebView (❶). Also, an application must acquire Android location permissions (ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION) or both, and implement an `onGeolocationPermissionsShowPrompt()` callback method to use the Geolocation API [1]. The callback method pops up a permission dialog (§5.3.3) when a browser visits web pages that use the Geolocation API, but are not in the web browser's per-domain nor per-page permission DBs. Second, the code overrides the `getCurrentPostion()` method with a custom function (❷). We use the `loadUrl()` method of WebView and the `javascript` protocol for overriding, which enable arbitrary JavaScript code execution within the scope of the current web page [22, 24]. Third, the code makes the custom function execute a success callback function with degraded geolocation (`dLatitude` and `dLongitude`) in accordance with a user's choices (❸). Lastly, the code reloads the current web page to activate the injected JavaScript code (❹). The revised code for the `watchPosition()` method is similar.

5.3.2 *Enhancing permission manager*

We implement a custom geolocation permission manager to enhance Android's default geolocation permission manager (`GeolocationPermissions` class). Figure 7a shows a simplified diagram representing how the default geolocation permission manager works. When a web site attempts to use the Geolocation API, the manager first checks whether the permission state of the web site resides in a DB file `GeolocationPermissions.db` (❶). If a corresponding record exists, the manager returns geolocation according to the record. Otherwise, the manager calls `geolocationPermissionsShowPrompt()` (❷) which eventually calls `onGeolocationPermissionsShowPrompt()` to show a permission dialog to obtain a user's decision (❸). The manager receives the decision (❹) and finally records it in the DB (❺).

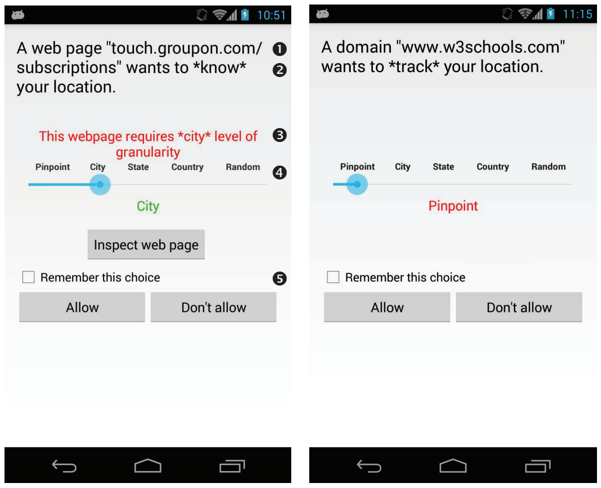
We extend the preceding procedure by adding a custom geolocation permission manager as shown in Figure 7b. Unlike the default manager, the custom manager distinguishes per-page and per-domain permissions, and executes a code to change geolocation (§5.3.1; §5.3.5). The default manager always calls the custom manager when a web site tries to use the Geolocation API because we record no data in the `GeolocationPermissions.db`.

5.3.3 *Designing permission dialog*

We use `onGeolocationPermissionsShowPrompt()` to design a new permission dialog (Figure 8), allowing us to intercept and customize the dialog for requesting geolocation permissions [1]. First, the dialogs show whether a user currently uses per-page (Figure 8a) or per-domain (Figure 8b) permission models (❶). The user can choose one of the permission models in browser settings. Second, they show whether a current web page attempts to use `getCurrentPosition()` or `watchPosition()` (❷). This information is helpful for the user because conventional geolocation dialogs do not distinguish the two different methods. Third, the dialog of the per-page permission model (Figure 8a) displays the estimated location sensitivity of the web page (❸) when the user touches an "Inspect web page" button. The user may refer the result when choosing privacy settings. In contrast, the dialog of the per-domain permission model (Figure 8b) does not have such a button because our scheme does not verify the changes of a domain's content (§5.3.4) Fourth, the dialogs allow the user to choose accuracy options: *pinpoint*, *city*, *state*, *country*, and *random* (❹). If the user selects the random, the geolocation becomes one of well-known cities (e.g., New York, London, or Paris). Lastly, the dialogs allow the user to either temporarily or permanently grant permissions to the web page, or deny providing any geolocation (❺).

5.3.4 *Managing permission DBs*

Our web browser has two permission DBs: a per-page permission DB and a per-domain permission DB. The per-page DB consists of (1) the URL of a web page, (2) whether a user allows or denies the web page to access geolocation, (3) degree of location accuracy the user grants to the web page, (4) the fuzzy hash of the web page for change verification, (5) the geolocation when computing the fuzzy hash, and (6) the (Unix) time when the web browser has visited the


A web page "touch.groupon.com/subscriptions" wants to *know* your location.

This webpage requires *city* level of granularity

A domain "www.w3schools.com" wants to *track* your location.

(a) Per-page, position, and inspected. (b) Per-domain, tracking, and not yet inspected.

Figure 8: Examples of the proposed permission dialogs.

web page lastly to reduce the number of repeated verifications of web page changes.

The per-domain DB is similar to the per-page DB except that it maintains neither fuzzy hash values nor the timestamp of the last visit because we cannot verify changes of all web pages under a domain. To verify whether a domain's content changes, we must compute the fuzzy hash values of all web pages in it. However, this computation has two problems: (1) we cannot enumerate all the web pages under the domain and (2) some of the web pages may change frequently. Moreover, granting permissions to a domain implies that a user trusts all web pages of the domain. Consequently, verifying all web pages under a domain is less meaningful.

Lastly, we must solve a *synchronization problem* between the DBs that occurs when users grant a geolocation permission to a domain whose web pages have already obtained geolocation permissions. To solve this problem, we remove permissions granted to web pages from the per-page DB when their domain obtains a permission.

5.3.5 *Decreasing geolocation accuracy*

According to a user's choices, our web browser decreases the accuracy of geolocation retrieved from an Android platform. A naïve method to decrease the geolocation accuracy is to adjust GPS coordinates [15], but this approach may lead to address information being wrong. For example, the adjustment can virtually locate a user in a different city, so if the web page that the user is on relies on city-level geolocation, the user receives meaningless information.

Address-aware geolocation manipulation. We propose an address-aware method that cleverly decreases geolocation accuracy, by altering postal address information instead of GPS coordinates. We use the Google Geocoding API [12] that translates GPS coordinates into the corresponding address information and vice versa.

We explain the procedures of the address-aware method with the following conditions: (1) the current GPS coordinates of a device are latitude 40.71751 and longitude -74.00348; and (2) our web browser wants to provide *city-level* geolocation to a web page. First, the web browser attempts to translate the GPS coordinates into the corresponding address information by requesting `http://maps.googleapis.com/maps/api/geocode/json?latlng=40.71751,-74.00348&sensor=true`. When the Google Geocoding API succeeds to process the request, the browser receives a JSON file with estimated geolocation, and extracts detailed address information from the JSON file, e.g., "359 Broadway, New York, NY 10007, USA" stored in the `formatted_address` field.

Next, the web browser truncates the street information and the zip code of the detailed address information, and obtains the corresponding GPS coordinates by requesting `http://maps.googleapis.com/maps/api/geocode/json?address=New+York,+NY,+USA&sensor=true`. When the Google Geocoding API successfully processes the request, the web browser receives a JSON file that includes GPS coordinates, such as latitude 40.71435 and longitude -74.00597, which point to the center of New York City. Lastly, the web browser provides the degraded GPS coordinates to the web page instead of the precise GPS coordinates. As a result, by using the proposed method, users can hide their detailed geolocation while assuring city-level accuracy.

Address caching. Using the Google Geocoding API for manipulating address information has unavoidable network overhead: our web browser should interact with the Google server whenever it visits web pages that demand geolocation. To solve this problem, we use an address caching method. Our web browser caches pinpoint-, city-, state-, and country-level GPS coordinates obtained by the address-aware geolocation manipulation in a DB. When a web page granted either city-, state-, or country-level permissions attempts to access geolocation while our browser's current GPS coordinates are close to some of the cached pinpoint GPS coordinates, the browser uses the corresponding cached information of allowed accuracy. If the browser's current location is not close to any of the cached GPS coordinates, it performs the address-aware geolocation manipulation and updates the DB. We thereby minimize the network overhead of using the Google Geocoding API. A number of studies identify that a user's location history does not suddenly change [26, 28], so address caching is effective.

5.3.6 *Suppressing the overflow of permission dialogs*

A per-page permission model has a problem: this model can pop up too many permission dialogs when our web browser visits a web site that has many web pages using the Geolocation API. To suppress the overflow of permission dialogs, our web browser automatically applies a per-domain permission model to a web site that pops up per-page permission dialogs too frequently.

5.4 Evaluation

We evaluate our scheme in terms of (1) accuracy, (2) time overhead of the location sensitivity estimation, and (3) overall storage overhead. For the evaluation, we use Galaxy S III and choose 200 web pages among the collected 1196 web pages that are in English and access the geolocation without user interaction.

5.4.1 *Accuracy*

We verify how accurately our scheme estimates location sensitivity of the selected web pages by comparing the results with manual inspection. The proposed scheme correctly estimates the location sensitivity of 93.5% of the web pages. We further analyze the failed estimations and confirm that the failures are due to web pages that (1) store the obtained geolocation in cookies and retrieve them in future (4.5%), (2) change their content according to both time and geolocation (1.5%), and (3) use IP addresses to inspect locations while ignoring the geolocation that our web browser provides (0.5%). To access the geolocation information stored on such web pages, the proposed scheme may need to manipulate cookies, analyze web page semantics, or cloak IP addresses, respectively.

5.4.2 *Time overhead*

We measure how much time our scheme spends to inspect location

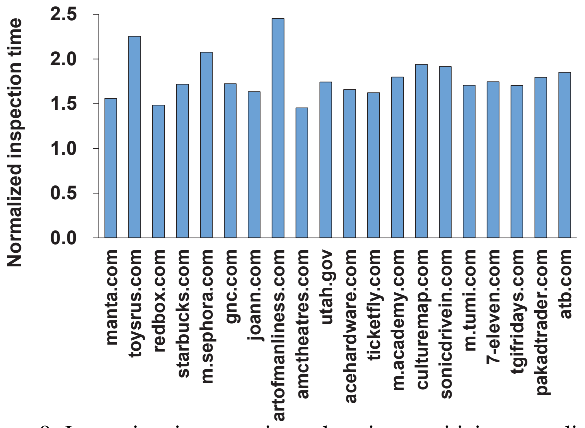


Figure 9: Inspection time to estimate location sensitivity normalized to page loading time.

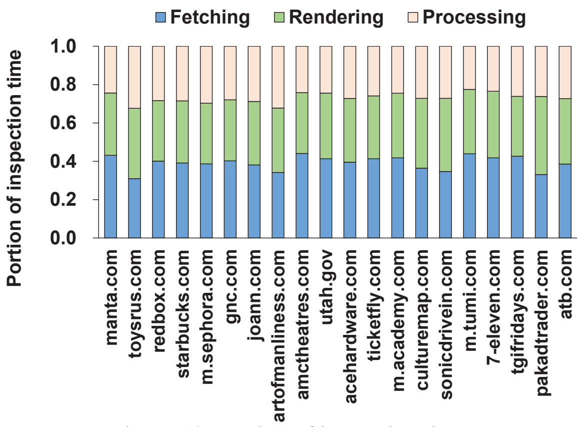


Figure 10: Portion of inspection time.

sensitivity of the selected web pages, which is performed when a user touches an "Inspect web page" button (§5.3.3). Measuring the time overhead, however, demands much time so that we choose 20 web pages out of the selected 200 web pages highly ranked in Alexa Top Sites. Figure 9 shows inspection time to estimate location sensitivity of the 20 web pages normalized to page loading time that consists of fetching and rendering time. On average, the inspection time is 1.8 times longer than the page loading time. Although the time overhead is high, it is tolerable because location-sensitivity inspection would not be frequently performed.

We also measure the processing time that our algorithm spends to estimate location sensitivity (§5.2.3). On average, the processing time only takes ∼27% of the overall inspection time (Figure 10), so the computational overhead of our scheme is acceptable.

5.4.3 Storage overhead

Lastly, we estimate the storage overhead of our scheme. We first check how much storage is necessary to store the per-page information of the 200 web pages in the per-page permission DB: the average size of each record is ∼126 B. Next, we check how much storage is necessary to store address caching information consisting of pinpoint-, city-, state-, and country-level GPS coordinates in a DB. We define an eight-byte field for storing a pair of latitude and longitude values, so the size of each record is 32 B. For example, if our web browser manages 10,000 per-page records and caches up to 10,000 address caching records, storage overhead is only ∼1.5 MB. Therefore, we conclude that our scheme's storage overhead is low.

6. DISCUSSION

6.1 Limitations

We discuss some limitations of the proposed scheme. First, our web browser cannot estimate the location sensitivity of web pages that demand high user interaction (e.g., button clicks or search keywords typings). However, this limitation is less important because such explicit user interaction reminds users that the web sites try to access their geolocation. Users who value their privacy may re-determine whether to grant their geolocation to the web sites.

Second, we cannot apply the proposed scheme to web browsers that disallow JavaScript overriding for security reasons because the proposed scheme should replace the `getCurrentPosition()` and `watchPosition()` methods. If web browser developers accept the proposed scheme and directly modify their web browsers, we can eliminate this limitation.

Third, web sites that recognize geolocation inspection may provide fake web pages to deceive our web browser. Our browser visits a web page several times with various geolocations, so web sites can identify whether the browser inspects their web pages when they compare consecutive HTTP requests. However, this identification can be a huge burden on the web sites, especially when they have a large number of concurrent users. Therefore, we presume that web sites are unwilling to identify our web browser.

Lastly, web sites that only use IP-based geolocation can ignore our scheme. However, a recent study [4] has identified that IP-based geolocation fails when an IP address belongs to cellular networks. For example, the study found that mobile devices hundreds of miles apart can share the same IP address space. Therefore, we anticipate that web sites would use the Geolocation API to obtain correct geolocation instead of using IP-based geolocation.

6.2 Suggestions

We summarize our suggestions to mitigate the privacy problems of the Geolocation API. First, we suggest that the Geolocation API has to allow web sites to control the accuracy of the geolocation that they demand. If the Geolocation API provides accuracy options (e.g., pinpoint, city, state, and country) and returns postal address instead of GPS coordinates, web sites can choose one of the options according to their requirements while eliminating geocoding costs. This improvement mitigates privacy threats while reducing computational and communication overhead of the web sites (§3.4).

Next, we recommend that the Geolocation API needs to employ per-method permission models for separating positioning and tracking because tracking may cause more serious privacy problems than positioning.

Lastly, we propose that the Geolocation API needs to allow web sites to request permissions for either domains or pages. Some web sites serving maps (e.g., Google Maps or Bing Maps) surely demand per-domain permissions because most of their web pages use the Geolocation API. However, other web sites such as shopping mall sites may demand per-page permissions because most of their web pages do not use the Geolocation API.

7. RELATED WORK

Location privacy. Several researchers have proposed various methods to preserve location privacy, which can be used to enhance the location privacy of our scheme. Gruteser and Grunwald [13] establish the concepts of k-anonymity and a trusted-anonymization server in LBSs. Instead of each user's exact GPS coordinates, the server computes a cloaking area including k different users and demands LBSs for the area. Other researchers [7, 23, 31] also consider personalized cloaking areas while assuring k-anonymity.

The k-anonymity-based methods have an important limitation: their cloaking area may contain meaningful locations (e.g., a company, a hospital, and a school), which implies that users are at the locations with a high probability. To solve this problem, Bamba *et al.* [5] propose an l-diversity-based method which enlarges a cloaking area until the area includes l different locations, and Lee *et al.* [20] propose a location-semantics-based method.

Lastly, researchers propose a client-based method to preserve location privacy without a trusted server. Yiu *et al.* [32] allow users to utilize fake locations instead of their precise locations when computing cloaking areas. Ghinita *et al.* [8] use a private information retrieval (PIR) protocol that allows users to retrieve records from a DB without revealing what records they request. Peer-to-peer-based methods [9, 16] also exist.

Geolocation inference attack. Jia *et al.* [17] propose an attack revealing the geolocation of a web browser without a user permission. They exploit a cache-timing attack [6] to recognize what geolocation-dependent resources are stored in the browser cache.

8. CONCLUSION

In this paper, we considered the privacy problems of the HTML5 Geolocation API due to its lack of fine-grained permission and location models. We detected vulnerable web browsers and over-privileged web sites that violate the location privacy of users by conducing case studies. We also proposed a novel scheme to enhance the privacy of using the Geolocation API by supporting fine-grained permission and location models, and by inspecting the location sensitivity of each web page.

In future, we will develop a cloud service to effectively solve the explored problems. We anticipate that the cloud service can minimize inspection overhead by reducing the number of redundant page inspections and can maximize the quality of location sensitivity estimations by using crowdsourcing.

9. REFERENCES

[1] Android Developers. WebChromeClient. `http://developer.android.com/reference/android/webkit/WebChromeClient.html`.

[2] Android Developers. WebView. `http://developer.android.com/reference/android/webkit/WebView.html`.

[3] Apple. UIWebView class reference. `https://developer.apple.com/library/ios/documentation/uikit/reference/UIWebView_Class/Reference/Reference.html`.

[4] M. Balakrishnan, I. Mohomed, and V. Ramasubramanian. Where's that phone?: geolocating IP addresses on 3G networks. In *IMC*, 2009.

[5] B. Bamba, L. Liu, P. Pesti, and T. Wang. Supporting anonymous location queries in mobile environments with PrivacyGrid. In *WWW*, 2008.

[6] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *CCS*, 2000.

[7] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *ICDCS*, 2005.

[8] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K.-L. Tan. Private queries in location based services: Anonymizers are not necessary. In *SIGMOD*, 2008.

[9] G. Ghinita, P. Kalnis, and S. Skiadopoulos. PRIVE: Anonymous location-based queries in distributed mobile systems. In *WWW*, 2007.

[10] globalogiq.com. HTML code search engine - search within HTML source and HTTP headers. `http://globalogiq.com/htmlcodesearch.htm`.

[11] Google. Improve your location's accuracy - maps for mobile help. `https://support.google.com/gmm/answer/3144282?hl=en&ref_topic=3137371`.

[12] Google Developers. Google Geocoding API - Google Maps API web services. `https://developers.google.com/maps/documentation/geocoding`.

[13] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys*, 2003.

[14] A. T. Holdener. *HTML5 Geolocation*. O'Reilly Media, Inc., 2011.

[15] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "these aren't the droids you're looking for": Retrofitting Android to protect data from imperious applications. In *CCS*, 2011.

[16] H. Hu and J. Xu. Non-exposure location anonymity. In *ICDE*, 2009.

[17] Y. Jia, X. Dong, Z. Liang, and P. Saxena. I know where you've been: Geo-inference attacks via the browser cache. In *W2SP*, 2014.

[18] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *CCS*, 2008.

[19] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

[20] B. Lee, J. Oh, H. Yu, and J. Kim. Protecting location privacy using location semantics. In *KDD*, 2011.

[21] H. Liu, H. Darabi, P. Banerjee, and J. Liu. Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(6):1067–1080, 2007.

[22] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In *ACSAC*, 2011.

[23] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new Casper: Query processing for location services without compromising privacy. In *VLDB*, 2006.

[24] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: WebView exploitation. In *LEET*, 2013.

[25] A. Restaino. Lightning browser. `https://github.com/anthonycr/Lightning-Browser`.

[26] O. Riva, C. Qin, K. Strauss, and D. Lymberopoulos. Progressive authentication: deciding when to authenticate on mobile phones. In *USENIX Security*, 2012.

[27] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *CCS*, 2006.

[28] E. Shi, Y. Niu, M. Jakobsson, and R. Chow. Implicit authentication through learning user behavior. In *ISC*, 2010.

[29] W3C. Geolocation API specification. `http://www.w3.org/TR/geolocation-API`.

[30] W3Schools. HTML5 geolocation. `http://www.w3schools.com/html/html5_geolocation.asp`.

[31] T. Xu and Y. Cai. Feeling-based location privacy protection for location-based services. In *CCS*, 2009.

[32] M. L. Yiu, C. S. Jensen, X. Huang, and H. Lu. SpaceTwist: Managing the trade-offs among location privacy, query performance, and query accuracy in mobile services. In *ICDE*, 2008.