A Large-Scale Study of Mobile Web App Security

Patrick Mutchler*, Adam Doupé†, John Mitchell*, Chris Kruegel‡ and Giovanni Vigna‡
*Stanford University
{pcm2d, mitchell}@stanford.edu
†Arizona State University
doupe@asu.edu
‡University of California, Santa Barbara
{chris, vigna}@cs.ucsb.edu

Abstract

Mobile apps that use an embedded web browser, or *mobile web apps*, make up 85% of the free apps on the Google Play store. The security concerns for developing mobile web apps go beyond just those for developing traditional web apps or mobile apps. In this paper we develop scalable analyses for finding several classes of vulnerabilities in mobile web apps and analyze a large dataset of 998,286 mobile web apps, representing a complete snapshot of all of the free mobile web apps on the Google Play store as of June 2014. We find that 28% of the studied apps have at least one vulnerability. We explore the severity of these vulnerabilities and identify trends in the vulnerable apps. We find that severe vulnerabilities are present across the entire Android app ecosystem, even in popular apps and libraries. Finally, we offer several changes to the Android APIs to mitigate these vulnerabilities.

I. Introduction

Mobile operating systems allow third-party developers to create applications ("apps") that run on a mobile device. Traditionally, apps are developed using a language and framework that targets a specific mobile operating system, making it difficult to port apps between platforms. Rather than building all of an app's functionaility using a development framework specific to a mobile operating system, developers can leverage their knowledge of web programming to create a *mobile web app*. A mobile web app is an app that uses an embedded browser to access and display web content. Often, a mobile web app will be designed to interact with web content written specifically for the app as a replacement for app-specific UI code. By building an app in this manner, developers can more easily deliver updates to users or port their app between platforms. In addition, several frameworks exist to simplify development by automatically producing the app code needed to interact with a web application [2, 8].

Unfortunately, security for mobile web apps is complex and involves a number of considerations that go beyond traditional app and web security. Developers cannot simply apply existing knowledge of web security and app security to create secure mobile web apps; vulnerabilities that cannot exist in traditional web apps can plague mobile web apps. Prior research on mobile web app vulnerabilities has either focused on small sets of apps, focused on only a subset of the kinds of mobile web apps, or made major simplifying assumptions about the behavior of mobile web apps. This has led to an understanding of the causes of vulnerabilities in mobile web apps but an inadequate understanding of their true prevalence in the wild.

In this work we study three vulnerabilities in mobile web apps (loading untrusted web content, exposing stateful web navigation to untrusted apps, and leaking URL loads to untrusted apps) and develop highly scalable analyses to identify these vulnerabilities more accurately than prior research. We analyze an extremely large dataset of 998,286 mobile web apps developed for Android, the mobile operating system with the largest market share. This dataset represents a *complete* snapshot of the free apps available on the Google Play marketplace, the largest Android app store. To the best of our knowledge, this is the most comprehensive study on mobile web app security to date. We find that 28% of the mobile web apps in our dataset contain at least one security vulnerability.

We explore the severity of these vulnerabilities and find that many real-world vulnerabilities are made much more severe by apps targeting outdated versions of the Android operating system and show that many recently published apps still exhibit this behavior. We examine trends in vulnerable apps and find that severe vulnerabilities are present in all parts of the Android ecosystem, including popular apps and libraries. Finally, we suggest changes to the Android APIs to help mitigate these vulnerabilities.

The main contributions of this paper are:

- We develop a series of highly scalable analyses that can detect several different classes of vulnerabilities in mobile web apps.
- We perform a large-scale analysis of 998,286 mobile web apps developed for Android to quantify the prevalence of security vulnerabilities in the Android ecosystem.
- We analyze trends in these vulnerabilities and find that vulnerabilities are present in all corners of the Android ecosystem.
- We suggest changes to the Android APIs to help mitigate these vulnerabilities.

We review relevant properties of Android and the WebView interface in Section II, and present an appropriate security model for mobile web apps and describe the vulnerabilities we will be studying in Section III. We explain our analysis methods in Section IV, followed by our experimental results

in Section V. Related work and conclusions are in sections VI and VII, respectively.

II. Background

Before we can understand the possible vulnerabilities in mobile web apps we must first understand their structure. Note that for the remainder of this paper we will only examine mobile web apps written for the Android platform. However, mobile web apps are in no way unique to Android. iOS and Windows Phone 8 apps have similar functionality[1].

Android allows apps to embed a custom webkit browser, called a WebView. WebViews can render web content obtained either from the Internet or loaded from files stored on the mobile device. Apps load specific content in the WebView by calling the methods `loadUrl`, `loadData`, `loadDataWithBaseUrl`, or `postUrl` and passing either strings containing HTML content or URLs as parameters. URLs can either be for Internet resources or for resources stored locally on the mobile device. We call methods used to navigate the WebView to web content *navigation methods*. Users can interact with the rendered content just like they would in a web browser. We call any app that includes embedded web content a *mobile web app*.

While developers can create a mobile web app simply by creating a WebView element and directing it to their web backend, several frameworks exist to simplify development by producing all of the code necessary to load web content packaged with an app. These apps, which we call "PhoneGap apps" based on the most popular of these frameworks, represent a subset of all mobile web apps with the unique feature that they only retrieve web content stored locally on the device rather than interacting with a remote webserver. For the purposes of this study we treat PhoneGap apps as a subcategory of mobile web apps rather than singling out their unique structure.

A. Inter-App Communication

Some vulnerabilities in mobile web apps involve inter-app communication. Apps primarily communicate with the Android operating system and other apps using an API for inter-process communication called *Intents*. When sending an Intent, apps specify either a specific app component to receive the Intent or that the Intent is a general action. Example actions include sending an email, performing a web search, or taking a picture. Intents that specify a specific app component are called *Explicit Intents* and are delivered only to the specified app component. Intents that specify an action are called *Implicit Intents* and are delivered to any app component that can handle that action. Apps declare the set of actions that each of their components can handle in their Manifest (an XML document packaged with the app) by declaring an *Intent Filter* for each component. Apps can also declare that a component can handle requests to view a particular resource by defining a custom URL pattern. A thorough examination of Intents can be found in [16].

Listing 1 shows part of a manifest for a simple app that contains two components. The first component responds to the action `SMS_RECEIVED`, which the Android operating system sends when there is an incoming text message. The second component responds to requests to load web URLs from host `example.com`.

```
<activity android:name="SMSHandler">
  <intent-filter>
    <action android:name=
    "android.provider.Telephony.SMS_RECEIVED"/>
  </intent-filter>
</activity>

<activity android:name="WebHandler">
  <intent-filter>
    <action android:name=
    "android.intent.action.VIEW"/>
    <data android:scheme="http"/>
    <data android:host="example.com"/>
  </intent-filter>
</activity>
```

Listing 1: A partial app manifest demonstrating intent filters. The app registers two components. One will respond to incoming SMS messages and the other will respond to requests to load web pages from example.com

B. Controlling Navigation

Most mobile web apps are not general purpose browsers. They are instead designed to interact with only specific web content. Android allows developers to intercept and prevent unsupported web resources from being loaded by implementing the callback methods `shouldOverrideUrlLoading` and `shouldInterceptRequest`. A WebView calls `shouldOverrideUrlLoading` before loading a new page in a top level frame[2]. A WebView calls `shouldInterceptRequest` before making any web request, including iframe and image loads. In both cases the app has an opportunity to prevent the resource load by returning `true` in the case of `shouldOverrideUrlLoading` or `null` in the case of `shouldInterceptRequest`. The default behavior of `shouldOverrideUrlLoading` is to prevent a load and the default behavior of `shouldInterceptRequest` is to *allow* a request. For the remainder of this paper we call these methods *navigation control methods*.

Simply overriding a URL load prevents a WebView from doing anything when a user clicks a link. This is unexpected and might harm user experience, so most apps will send an Intent to have the browser app load an overridden URL in addition to overriding the URL load. This approach allows apps to correctly constrain the web content loaded in their app without breaking links on the web. Listing 2 shows a `shouldOverrideUrlLoading` implementation for a mobile web app that only supports content from the example.com domain. All other content is prevented from being loaded in the WebView and is sent to the default web browser instead.

[1]Both platforms have a class that behaves similarly to the WebView class in Android. In iOS it is called a UIWebView and in Windows Phone 8 it is called a WebView.

[2]This method is not called when loading pages by calling navigation methods (i.e., when the app explicitly tells the WebView to load web content), by making POST requests, and by following redirects on Android versions below 3.0.

```
public boolean shouldOverrideUrlLoading(
      WebView view, String url){

   String host = new URL(url).getHost();
   if(host.equals("example.com")){
      return false;
   }
   Intent i = new Intent(
      Intent.ACTION_VIEW,
      Uri.parse(url)
   );
   view.getContext().startActivity(i);
   return true;
}
```

Listing 2: A `shouldOverrideUrlLoading` implementation that constrains navigation to pages from example.com. Any page from another domain will be loaded in the default browser app.

C. JavaScript Bridge

A key difference between a mobile web app and a typical web app is the enhanced capabilities of web content loaded in a mobile web app. Web browser are beginning to expose some APIs to web applications (e.g., location services), but a mobile web app is able to combine normal web application functionality with all of the functionality available to a mobile app. This combination allows developers to create rich new types of applications that cannot exist in a typical browser.

To facilitate tight communication between app code and web content, Android includes a feature called the *JavaScript Bridge*. This feature allows an app to directly expose its Java objects to JavaScript code running within a WebView. Specifically, if an app calls `addJavascriptInterface(obj, "name")` on a WebView instance then JavaScript code in that WebView can call `name.foo()` to cause the app to execute the Java object `obj`'s method `foo`, and return its result to the JavaScript code. We call a Java object that has been added to the JavaScript Bridge a *Bridge Object* and the JavaScript object used to access the Bridge Object a *Bridge Reference*.

The relationship in Android between Bridge Objects, Bridge References, and the Same Origin Policy is unintuitive. If an app creates a Bridge Object then JavaScript code from *any* origin has access to a matching Bridge Reference, even if that content is loaded in an iframe. Bridge Objects remain available to web content loaded in a WebView even after navigating to a new page. Each Bridge Reference is protected from the others by Same Origin Policy but they can all call methods on the same Bridge Object. Therefore, Bridge Objects are tied to the WebView instance rather than isolated by Same Origin Policy.

Figure 1 shows how multiple origins can use isolated Bridge References to access the same Bridge Object. This lack of confinement can allow malicious web content to attack an app through the JavaScript Bridge. No official mechanism exists to expose Bridge References to particular origins or provide any sort of access control. Official documentation on the JavaScript Bridge feature can be found in [7].

III. Mobile Web App Security

In this section we describe the security model for mobile web apps and describe several classes of vulnerabilities in mobile web apps. We will later construct analyses to find and quantify these vulnerabilities in our dataset.

A. Adversary Model

There are three relevant adversaries to consider when discussing mobile web app security:

App Adversary. The *app adversary* captures the attack capabilities of a malicious app running alongside a trusted app. An app adversary may read from and write to the shared filesystem, may send intents to any apps installed on the device, and may register components that respond to intents.

Network Adversary. The *network adversary* may receive, send, and block messages on the network. However, the network adversary does not have access to cryptographic keys of any other party. This is the standard network adversary used in the design and analysis of network security protocols.

Navigation-Restricted Web Adversary. The *navigation-restricted web adversary* is a variant of the typical web adversary. Specifically, the navigation-restricted web adversary may set up any number of malicious web sites and place any content on them. However, because mobile device users can only navigate mobile web apps through the interface of the app, a mobile web app may only navigate to a restricted set of sites that is limited by the internal checks and behavior of the app.

For comparison, the standard web adversary model assumes a user will visit any malicious content (in a separate tab or window from other content) [13]. This is a reasonable assumption in the design and analysis of web security mechanisms because browsers provide a URL bar for the user to visit any web content and there are ample mechanisms for tricking an honest user into visiting malicious content. In contrast, a user navigates a mobile web app only by interacting with the app itself or by following links in embedded web content that is reached in this way.

B. Studied Vulnerabilities

1) Loading Untrusted Content: It is very difficult for a mobile web app to ensure that untrusted web content loaded in a WebView is safely confined to the WebView. Apps cannot easily control which domains have access to Bridge Objects, allowing untrusted web content to execute app code through the JavaScript Bridge. In addition, WebView contains an unpatched Universal Cross-Site Scripting vulnerability in versions below Android 4.4 [4, 1]. This vulnerability affects almost 60% of in use Android devices [5]. Finally, because mobile web apps do not include a URL bar, users have no indication about what site they are visiting and whether their connection is secure. This means that users cannot make an informed decision about whether to input sensitive information like credentials.

For these reasons, security best practices for mobile web apps state that it is not safe to load *any* untrusted web content in a WebView. This is true even if the untrusted content is

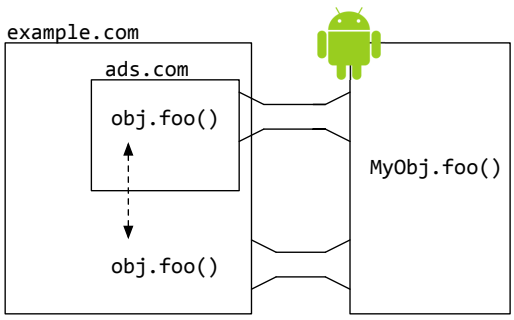

Fig. 1: An example of Same Origin Policy limitations for the JavaScript Bridge. An app exposes an instance of MyObj to the JavaScript Bridge and loads a HTML page from example.com with an iframe containing content from ads.com. Both example.com and ads.com have access to a separate Bridge Reference "obj". These Bridge References are separated from each other by Same Origin Policy (dashed line) but both Bridge References can call methods on the same Bridge Object through the JavaScript Bridge.

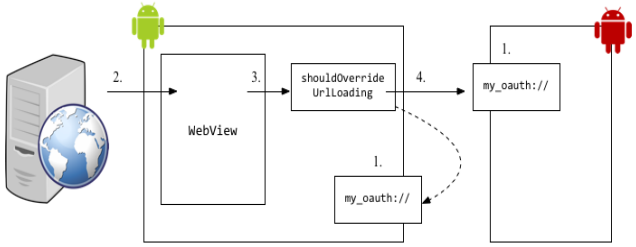

Fig. 2: An app leaking an OAuth callback URL. In Step 1 both the vulnerable app and the malicious app register an app component to handle URLs matching the protocol scheme `my_oauth`. In Step 2 the OAuth provider completes the protocol and responds with an HTTP 302 response to redirect the WebView. In Step 3 the WebView passes this URL to `shouldOverrideUrlLoading`, which uses an Implicit Intent to deliver the URL and leaks the URL to the malicious app in Step 4.

loaded in an iframe. In general, there are four ways that a mobile web app can load untrusted content. An app can allow navigation to untrusted content through normal user interaction, it can load trusted content over HTTP, it can load trusted content that is stored insecurely on the device, or it can load trusted content over HTTPS but use HTTPS incorrectly.

The first three methods of loading untrusted web content are straightforward but the fourth method demands more explanation. In a traditional browser environment an app has no control over the browser's SSL implementation. If the browser finds a problem with its SSL connection then it displays a warning to the user. WebView, on the other hand, allows developers to control an app's behavior in the presence of SSL certificate errors by overriding the callback `onReceivedSslError`. This even includes proceeding with a resource load without informing the user. Apps that load resources over SSL despite invalid certificates lose all of the protection HTTPS gives them against active network adversaries.

2) Leaky URLs: Apps can leak information through URL loads that are overridden by navigation control methods. When an app overrides a URL load and uses an Implicit Intent to load that resource, any app can handle that URL load. If a leaked URL contains private information then that information is leaked along with the URL. A developer might think that it is safe to use an Implicit Intent to deliver a URL to an app component because the URL matches a custom URL scheme but Android does not provide any protections on custom URL schemes. An example of this vulnerability was discussed by Chen et al. [15] in relation to mobile OAuth implementations. If an app registers a custom URL pattern to receive the final callback URL in an OAuth transaction and uses an Implicit Intent to deliver the URL then a malicious app can register the same URL pattern and steal the OAuth credentials. See Figure 2 for a visual representation of this vulnerability.

3) Exposed Stateful Navigation: Developers must be careful about what app components they expose to Intents from foreign apps. Existing research has explored apps that leak privileged device operations (e.g, access to the filesystem or GPS) to foreign apps through Intents. Similarly, a mobile web app can leak privileged *web* operations to foreign apps by blindly responding to Intents. More specifically, if an app performs a call to `postUrl` in response to a foreign Intent then a malicious app can perform an attack similar to Cross-Site Request Forgery. For example, if a mobile web app uses a POST request to charge the user's credit card, and this request is exposed to foreign Intents then a malicious app could send an Intent to place a fraudulent charge without the user's knowledge or consent. In order to prevent this vulnerability, developers must ensure that any calls to `postUrl` that can be triggered by an Intent from a foreign app are confirmed by the user through some UI action.

IV. Analyses

In this section we describe the methods used to identify vulnerabilities in mobile web apps and determine the severity of these vulnerabilities. In order to scale this experiment to a dataset of nearly one million apps we designed these techniques with efficiency as a priority. This can lead to some imprecision in our results, however, we note that several properties of mobile web apps make these analyses more precise then one might expect. We also note that our methods were designed to be conservative whenever possible so that we do not incorrectly flag secure apps as vulnerable. We discuss the limitations of our analyses and their effects on our results in more detail in Section V.

A. Reachable Web Content

Mobile web apps that load unsafe web content expose themselves to attack. We identify apps that load unsafe web content (e.g., content from untrusted domains or content loaded over HTTP) in three steps. (1) extract the set of initially reachable URLs from the app code, (2) extract the navigation control implementations from the app code, and (3) perform a web crawl from the initial URLs while respecting the navigation control behavior and report any unsafe web content. This method mirrors the true navigation behavior of an app.

1) Initial Resources: To find the set of web resources that a mobile web app loads directly we perform a string analysis that

reports the possible concrete values of parameters to navigation methods like `loadUrl`. In order to run quickly, our analysis is mostly intraprocedural and supports simple string operations (e.g., concatenation) but does not support more complex string operations (e.g., regular expression matching or substring replacement). This limits the precision of our analysis but, based on our analysis, we conclude that mobile web apps will often access hard-coded web addresses or build URLs very simply so this simple approach is often very effective. Strings that cannot be computed by our analysis usually originate either far away from a call to a navigation method or even in an entirely separate app component. Extracting these strings would require not only a precise points-to analysis but also an accurate understanding of the inter-component structure of an app, taking us beyond the state of the art in scalable program analyses. We built our string analysis using Soot, a Java instrumentation and analysis framework that has support for Dalvik bytecode [29, 12].

When possible, our string analysis also reports known prefixes to unknown values. The prefix can give us information about the loaded content even if we cannot compute the URL. For example, a URL with the concrete prefix `http://` tells us that an app is loading content over an insecure connection even if we do not know what content the app is loading.

In Android apps, many string constants are not defined in app code. Instead they are defined in a XML documents packaged with the app and then referenced by calling `Resources.getString` or similar methods. A naive string analysis will fail to find these constants. We parse these XML documents and replace calls to `Resources.getString` and similar methods with their equivalent constant string values before running our analysis. We use apktool [6] to unpackage app contents and access these XML documents.

2) Handling Navigation Control: In order to understand how an app can navigate the web and expose itself to unsafe web content we must understand the behavior of any implementations of `shouldOverrideUrlLoading` and `shouldInterceptRequest`, the details of which are described in Section II-B. Previous work by Chin et al. [17] categorized implementations as allowing *all* navigation or *no* navigation based on a heuristic and performed a web crawl if an implementation was categorized as allowing navigation. This does not capture the full behavior of navigation control in Android because many apps will allow navigation to some URLs but not others (as our example in Listing 2 demonstrates). In addition, the authors do not analyze implementations of `shouldInterceptRequest`. Below, we describe our approach that more precisely handles navigation control methods by computing the results of these methods for concrete URLs.

We extract an app's implementations of `shouldOverrideUrlLoading` and `shouldInterceptRequest` and create a runnable Java program that, when given a URL to test, reports the behavior of these methods as if they had been called on that URL during normal app execution. Specifically, we compute and extract a *backwards slice* of the method with respect to any return statements, calls to navigation methods, and calls to send Intents. A backwards slice is the set of all program statements that can affect the execution of a set of interesting program statements [31]. Algorithms to compute backwards slices for sequential programs are well understood, and we use a known algorithm to compute our slices [11]. This approach is much more efficient than running an app in an emulator to determine if a WebView is allowed to load a page.

A backwards slice might contain program statements that cannot be executed outside of the Android emulator. For example, an implementation of `shouldOverrideUrlLoading` might access a hardware sensor or the filesystem. In order to keep our approach sound but still execute extracted slices, we remove any statements that we cannot execute in a stand-alone Java executable and insert instrumentation to mark data that these statements can edit as unknown. If during execution a statement uses unknown data we halt execution and report that the result could not be determined.

A true backwards slice is sometimes unecessary to correctly capture the behavior of a navigation control method. We do not care about the behavior of particular statements in a navigation control method. We only care about the overall behavior of the method. Two different program statements that return the same value are identical for our purposes. Therefore, we can further simplify our slice by combining "identical" basic blocks. Specifically, if all paths from a particular branch statement exhibit the same behavior with respect to return statements, calls to navigation methods, and calls to send Intents then we can replace the branch and all dominated statements with their shared behavior. This pruning step makes it possible to execute slices that branch based on app state for reasons other than controlling navigation. Like our string analysis, our slicer was built using Soot.

3) Crawling: Once we have the set of initial URLs and the extracted navigation control implementations we can identify the set of resources that an app can reach by performing a web crawl starting from each initial URL and only loading a resource if the extracted navigation control implementations allow it. We are careful to spoof the appropriate headers[3] of our requests to ensure that the web server responds with the same content that the app would have retrieved.

Finally, we must decide if web content reachable through user interaction is trusted or not. Apps do not explicitly list the set of web domains that they trust so this must be inferred from the behavior of the app. We assume that an app trusts the local private filesystem (file://), all domains of initially loaded web content, and all subdomains of trusted domains. If a web crawl reaches content from any untrusted domain then we report a security violation.

B. Exposed Stateful Navigation

Apps that expose `postUrl` calls to foreign Intents are vulnerable to a CSRF-like attack where foreign apps can force state changes in the backing web application. A call to `postUrl` is exposed if it can be reached from an exposed app component without any user interaction. We find the set of exposed app components by examining the app's Manifest (see Section II-A). The challenge is how to determine if there

[3] Specifically, we specify the `X-Requested-With` header, which Android sets to the unique app id of the app that made the request, and the `User-Agent` header.

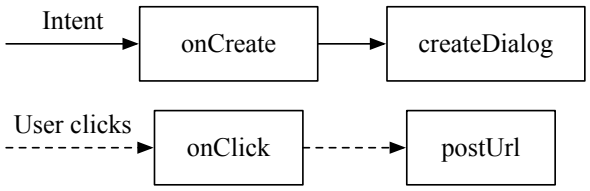

Fig. 3: A path from a foreign Intent to `postUrl` that is broken by a UI callback. Clicking the button calls `onClick` but the call edge comes from the OS and is not present in app code. Solid edges are found during reachability analysis and dashed edges are not.

Mobile Web App Feature	% Apps
JavaScript Enabled	97
JavaScript Bridge	36
shouldOverrideUrlLoading	94
shouldInterceptRequest	47
onReceivedSslError	27
postUrl	2
Custom URL Patterns	10

TABLE I: The percentage of mobile web apps that contain functionality we study in this experiment.

has been any user interaction during an execution path from an app component's initialization methods to a call to `postUrl`.

We observe that user interaction in Android is generally handled through callback methods. For example, to create a conformation dialog box an app might create a UI element and hook a callback method to the *confirm* button. If the user confirms the action, the operating system will then call the registered method. This design pattern means that control flow involving user interaction will break at the callback methods in a normal reachability analysis so paths that involve user input will not be reported. We can therefore perform a traditional reachability analysis starting with each exposed app component's initialization methods (`onCreate`, `onStart`, and `onResume`) to find exposed POST requests. Figure 3 shows a path to a navigation method that is broken at a callback method where user interaction occurs.

Performing a precise points-to analysis necessary to compute a precise call graph is too inefficient to scale to our dataset. Instead, we compute the possible receiver methods of a call site by considering only the syntactic type of the receiver object. This approach can generate spurious call edges and lead to false positives but allows us to fully analyze our dataset more efficiently.

C. Mishandled Certificate Errors

By default, an embedded WebView will cancel loading a resource over HTTPS when there is a certificate error, however developers are able to change this behavior by implementing the method `onReceivedSslError`. Developers can do one of three things in this method. They can let the request proceed as normal by calling `SslErrorHandler.proceed`, cancel the request by calling `SslErrorHandler.cancel`, or load a different resource by calling a navigation method. It is difficult to statically determine if an implementation is correctly validating a certificate. Therefore, we only analyze whether an implementation *must* ignore certificate errors on all paths and conservatively report all other apps as secure. We can do this completely intraprocedurally by also performing an escape analysis to ensure that a reference to the `SslErrorHandler` instance does not leave the method body.

D. Leaky URLs

An app "leaks" a URL load if it prevents that URL from being loaded with `shouldOverrideUrlLoading` and instead has the operating system process the request. Foreign apps can register an app component to handle the URL load. However, it is often correct behavior for an app to send a URL load to be handled by the operating system. A leaky URL is only a security violation if the app *intends* to load that URL itself.

We say that an app intends to load a URL if the URL matches a custom URL pattern registered by the app in its Manifest. We identify vulnerable apps by computing a regular expression that describes the URL patterns that an app registers in its Manifest and then generating sample URLs to test against an app's implementation of `shouldOverrideUrlLoading`. We report a vulnerability if any URLs matching a custom URL pattern are allowed to escape the app by the appropriate implementation of `shouldOverrideUrlLoading`.

V. Results

Our snapshot of the Google Play store contains 1,172,610 apps, 998,286 (85%) of which use a WebView in some fashion. These apps represent a complete snapshot of the free mobile web apps on the store as of June, 2014. Along with the apps, we collected data about each app including the number of times it has been downloaded and the most recent date that it was updated. We performed our analyses on 100 Amazon EC2 c3.4xlarge virtual machines for approximately 700 compute hours. Table 1 shows the percentage of mobile web apps that use features relevant to this study. In total, 28% of mobile web apps in our dataset contained at least one vulnerability. We are currently reporting these vulnerabilities to the developers of the most popular apps and libraries. A summary of our results can be found in Table 2.

A. Unsafe Navigation

15% of the apps in our dataset contained at least one fully computed URL for an Internet resource (as opposed to a local file). Of these apps, 34% (5% of the total population) were able to reach untrusted web content by navigating from an initial URL while still obeying the behavior of any navigation control methods. Notably, almost all of these apps could reach untrusted content either in a single link or in an iframe on an initially loaded page.

B. Unsafe Content Retrieval

40% of the mobile web apps in our dataset had a computable scheme for at least one URL. 56% of these apps (22% of the total population) contained a URL with an HTTP

scheme. Only 63% of the apps contained a scheme for *any* Internet resource (HTTP or HTTPS) so the large majority of mobile web apps that retrieve content from the Internet (as opposed to a local file) retrieve some content unsafely. We found that almost no apps (less than 0.1%) load web content from the SD card.

C. Unsafe Certificate Validation

27% of mobile web apps in our dataset contain at least one implementation of `onReceivedSslError`. 29% of these apps (8% of the total population) contained at least one implementation that ignores certificate errors on all code paths. If we include unsound results where the `SslErrorHandler` instance can escape the method body then the vulnerability rate jumps to 32%. These numbers together provide a sound under and over approximation of the percentage of apps that always ignore certificate errors.

We note that there appears to be widespread confusion about the intended use and consequences of `onReceivedSslError`. We analyzed the corpus of posts on StackOverflow [9] and found 128 posts that include implementations of `onReceivedSslError`. 117 of these posts ignore the certificate error on all code paths. These posts are most commonly helping developers get content loaded over HTTPS to display properly when using a self-signed certificate. While apps that perform web requests manually can pin a trusted certificate using a `TrustManager` instance, WebView does not have an official mechanism for trusting self-signed certificates. We suspect that developers believe that implementing `onReceivedSslError` is an appropriate alternative and do not understand the consequences of ignoring all errors.

D. Exposed POST Requests

1.9% of mobile web apps in our dataset contain at least one call to `postUrl`. Of these apps, our analysis reports that 6.6% expose a call to `postUrl` to foreign intents. This result is interesting in that the percentage of apps that fail to use `postUrl` safely is quite high but the vulnerability is rare in the overall app ecosystem.

E. Leaky URLs

10% of mobile web apps in our dataset register at least one custom URL scheme. Of these apps, 16% of these apps (1.6% of the total population) can leak a matching URL to another app by overriding it with `shouldOverrideUrlLoading` and sending it to the operating system as an implicit intent. Among the vulnerable apps are 1,135 apps that leak URLs with OAuth schemes in the manner described in Section III-B2.

F. Expired Domains

While crawling the apps in our dataset we found that 193 mobile web apps loaded an expired domain directly—that is, the expired domain was one of the initial URLs loaded by the app. Because apps remain installed on user's devices even after a company goes out of business, this represents an interesting new way that an attacker can deliver malicious content to a mobile web app even if an app uses HTTPS properly and only loads content from trusted domains.

Vuln	% Relevant	% Vulnerable
Unsafe Navigation	15	34
Unsafe Retrieval	40	56
Unsafe SSL	27	29
Exposed POST	2	7
Leaky URL	10	16

TABLE II: A summary of vulnerability rates in mobile web apps. The "% Relevant" column reports the percentage of apps in our dataset that could exhibit a vulnerability because they have all of necessary functionality and we could compute sufficient information about the app. The "% Vulnerable" column reports the percentage of these apps that contain vulnerabilities.

We took over seven expired domains, chosen based on the popularity of the app that loaded that domain. We followed the methodology for domain takeovers established by Nikiforakis et al. [26]. In total, we received 833 hits to these domains that came from a mobile web app (determined by looking at the `X-requested-with` header). This result demonstrates that this is a feasible new method of attacking mobile web apps.

G. Library Vulnerabilities

Libraries that contain security vulnerabilities are very worrisome as a single vulnerability can be present in a large number of apps and developers generally do not audit libraries they use for security vulnerabilities. Table 3 shows the percentage of vulnerabilities that can be attributed to library code. It is clear from these results that vulnerabilities in libraries are as serious of a concern as vulnerabilities in main app code. More than half of the observed vulnerabilities related to unsafe retrieval of web content or ignoring SSL errors come from library code. A few library classes in particular account for an enormous number of vulnerabilities in our dataset. A single faulty implementation of `onReceivedSslError` is present in 10,175 apps and is the sixth most frequently occurring implementation of `onReceivedSslError` in the entire app ecosystem.

The overwhelming majority libraries that contain vulnerabilities are either ad libraries or mobile web app framework libraries. Of the 50 library classes that contribute the most total vulnerabilities across all apps, 28 (56%) are from framework libraries and 22 (44%) are from ad libraries.

H. Vulnerability Trends

The huge majority of apps on the Google Play store are only downloaded by a few users and many apps are published but not maintained over time. An overall vulnerability rate

Vuln	% Library Vulnerabilities
Unsafe Navigation	29
Unsafe Retrieval	51
Unsafe SSL	53
Exposed POST	41
Leaky URL	45

TABLE III: The percentage of vulnerabilities in each vulnerability class that can be attributed to library code.

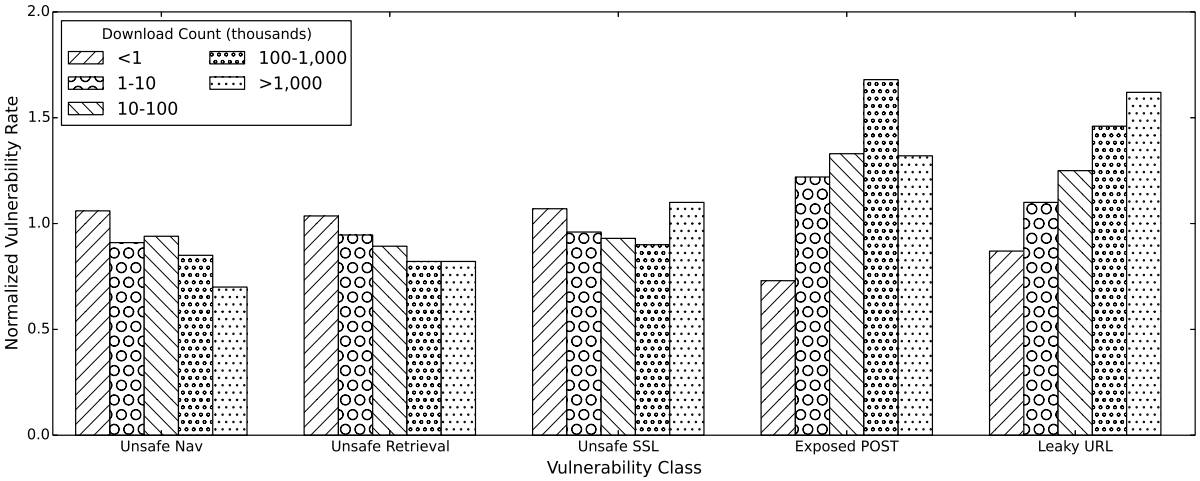

Fig. 4: A comparison of normalized vulnerability rates by app download count.

can therefore be misleading if most of the vulnerable apps are either unpopular or long out of date. However, we find that this is not the case for the vulnerabilities we studied. Figure 4 shows the normalized vulnerability rate for each vulnerability class broken down by app popularity. There is no clear trend across all vulnerability classes, but there are strong trends within vulnerability classes. The percentage of apps that allow navigation to untrusted content, load content over HTTP, and ignore certificate errors decreases with app popularity. The percentage of apps that expose POST requests to foreign intents and leak URLs increases dramatically with app popularity. We suspect that vulnerabilities related to unsafe web content are more well known in the community and the major developers are addressing these vulnerabilities while vulnerabilities related to exposed POST requests and leaky URLs might be less well understood.

Figure 5 shows the normalized vulnerability rate for each vulnerability class broken down by whether an app received an update within one year of our collection date (June, 2014). We might expect apps that receive regular updates would be less likely to contain vulnerabilites but the data is more mixed, showing different trends for different vulnerability classes. Recently updated apps are less likely to ignore certificate errors, expose POST request, and leak URLs but are more likely to allow unsafe navigation to untrusted content. Among all vulnerability classes the vulnerability rate is still high in recently updated apps, indicating that users cannot stay safe by only using up-to-date apps.

We also want to highlight a specific vulnerability related to the JavaScript Bridge. Apps that load untrusted web content in some manner, use the JavaScript Bridge, and target Android API version 4.1 or below are vulnerable to a remote code execution attack. Malicious web content uses the Java Reflection interface on an exposed Bridge Object to execute arbitrary Java code with the permissions of the app [10]. This dramatically increases the capabilities of malicious web content and exposes the user to all kinds of havoc. 56% of apps that can load untrusted web content use the JavaScript Bridge

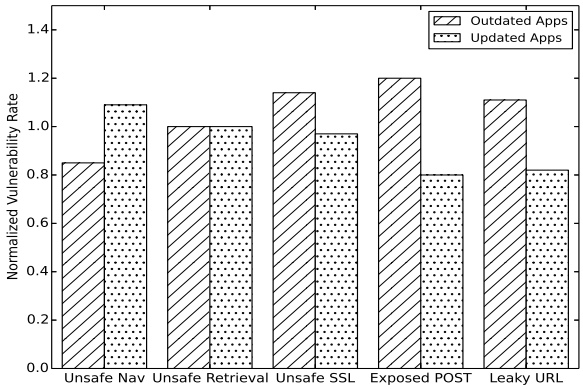

Fig. 5: A comparison of normalized vulnerability rates between apps that have been updated or first published within one year of data collection (June, 2014) and apps that have not been updated recently. Note that the rates do not average to 1.0 because of different population sizes.

(a much higher percentage than the general app population) and 37% target Android API version 4.1 or below.

Because Android 4.2 was released more than two years ago (November, 2012), we might expect that recently published or updated apps would be safe against this exploit. Unfortunately this is not the case. Figure 6 shows the percentage of vulnerable apps using the JavaScript Bridge that target unsafe Android API versions. We can see a clear trend downwards between old and new apps but the percentage of apps that target unsafe Android API versions, and thereby increase the severity of attacks from untrusted web content, remains high.

I. *Threats to Validity*

The scale of this experiment demands that our analyses are less computationally intensive, which can lead to both

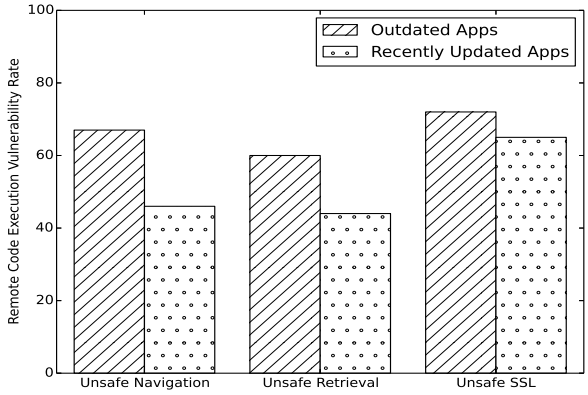

Fig. 6: A comparison of Remote Excecution Exploit rates between apps that have been updated or first published within one year of data collection (June, 2014) and apps that have not been updated recently.

false positives (reporting a vulnerability when none exists) and false negatives (failing to report a true vulnerability). Knowing this, we designed our analyses to be conservative and avoid false positives. This ensures that our measurements represent a lower bound on the number of WebView vulnerabilities in the Android ecosystem. However, there are still some opportunities for false positives, which we address here.

The string analysis used to identify initial URLs is very conservative. The analysis is intraprocedural to prevent call graph imprecision from creating incorrect results and we treat joins as assignments of unknown values so every reported URL will be a true concrete parameter to a navigation method. The only risk is if a call to `loadUrl` is dead code. We manually analyzed 50 apps that contained reported initial URLs and found no examples of dead `loadUrl` calls that had reported URL values.

We only report that an ignores certificate errors if an implementation of `onReceivedSslError` calls `proceed` on all code paths. While this approach will fail to report apps that incorrectly validate certificates, it ensures that every reported app contains an insecure implementation of `onReceivedSslError`.

Our analysis of `shouldOverrideUrlLoading` implementations is similarly conservative. Because we only report the result of this method if the slice was able to execute completely and without using unknown data, we can be sure that any behavior we observe is identical to the behavior in the running app.

The reachability analysis used to identify apps that expose POST requests to foreign apps is the most likely to report false positives. Rather than generating a call graph using points-to sets, we only use the static type of receiver objects and the class hierarchy to determine virtual method targets. This can lead to spurious call edges and invalid paths from an exposed app component to a POST request. Apps can also include logic to validate incoming Intents, which will not be captured by our analysis. We manually analyzed 50 apps that our reachability analysis reported and found a false positive rate of 16%. If this rate holds true across the entire dataset then the true vulnerability rate among apps that use `postUrl` is 5.5%. We note that the false positives usually involved much longer call chains than the true positives, which implies that setting a maximum call chain length could reduce the false positive rate.

J. Mitigation

Addressing these varied vulnerabilities is a serious challenge, but here we offer suggestions to Google Android and mobile OS developers to help reduce the frequency or severity of vulnerabilities unique to mobile web apps.

- Allow developers to specify a whitelist of trusted domains in a declarative fashion in the app's Manifest. Today, developers must correctly implement both `shouldOverrideUrlLoading` *and* `shouldInterceptRequest` to safely constrain navigation. This change would ensure that safe navigation control is foolproof.
- Allow developers to expose Bridge Objects to a whitelist of domains. Presently, apps depend on correct navigation control to ensure that untrusted content does not have access to the JavaScript Bridge. This would reduce the severity of attacks available to untrusted web content loaded in a WebView.
- Display information about the security of the user's connection. Right now there is no way for a concerned user to know the status of his or her connection and cannot make an informed decision to input sensitive information.
- Display a warning in AndroidStudio on calls to `SslErrorHandler.proceed`. A warning is already displayed when apps enable JavaScript and such a warning would hopefully reduce the number of apps that ignore any and all certificate errors.
- Allow developers to declare unique custom URL schemes and ensure that no other installed app registers the same schemes. In Android 5.0 custom permissions were changed to have a uniqueness requirement [3]. A similar feature for URL schemes would ensure that apps cannot leak URLs with custom schemes to foreign apps.

If followed, these recommendations will reduce both the frequency and severity of mobile web app vulnerabilities.

VI. Related Work

A number of studies have examined individual vulnerabilities related to the vulnerability classes we study. None look at datasets of the same scale or are able to discuss trends in vulnerable apps. In each case, as explained below, our work expands beyond the prior understanding of these vulnerabilities.

Luo et al. [25], Chin et al. [17], and Georgiev et al. [21] studied attacks on mobile web apps from untrusted content. Luo et al. [25] manually analyzed a small set of apps for vulnerabilities but incorrectly assumed that mobile web apps

allow navigation to arbitrary web content and therefore failed to explore the effect of navigation control on mobile web app security. Chin et al. [17] built a static analyzer to find apps that allow navigation to untrusted content. The authors understand the importance of the `shouldOverrideUrlLoading` method but handle it using a simple heuristic that fails to capture its behavior accurately. The authors also do not consider implementations of `shouldInterceptRequest` in their analysis. Georgiev et al. [21] studied the JavaScript Bridge in mobile web apps developed using PhoneGap and analyzed several thousand mobile web apps for vulnerabilities but their experiment is limited to apps built using PhoneGap, and, like Luo et al. [25], they do not explore the effect of navigation control. Within the scope of attacks from untrusted web content, our work extends beyond these results by accurately capturing the effect of navigation control methods when identifying apps that navigate to untrusted web content.

Fahl et al. [19] and Tenduklar et al. [28] analyzed Android apps for errors in their SSL implementations and found that many apps validated certificates incorrectly. Sounthiraraj et al. [27] built a dynamic analysis tool to detect apps that incorrectly validate SSL certificates. However, these studies overlook `onReceivedSslError` as a method of incorrectly validating SSL certificates and do not measure how freqently developers implement it unsafely. Our results are therefore orthogonal to these results and combine to demostrate that incorrect use of SSL is a widespread problem in Android apps.

Chen et al. [15] manually studied several popular OAuth providers for mobile apps and identified how the differences between the mobile and browser environments can lead to OAuth vulnerabilities. One vulnerability they describe is how a leaky URL vulnerability can expose OAuth credentials to malicious apps. Their work focuses specifically on OAuth and does not consider the more general vulnerability class of leaky URLs or the broader vulnerability classes we consider.

Several studies have explored exposed app components as a mechanism to perform privilege escalation attacks in Android. Davi et al. [18] first demonstrated that inter-app communication can leak privileged operations to other apps. Grace et al. [22] and Lu et al. [24] built systems for identifying apps that leak privileges. Felt et al. [20] proposed a mechanism for preventing privilege leaks using IPC inspection, and Bugiel et al. [14] provided another solution to prevent privilege leaks at the operating system level. The mechanism for these vulnerabilities is similar to the mechanisms behind the exposed POST request vulnerability but represent attacks on the mobile device and its APIs instead of an attack on a mobile web application.

Other studies have explored different vulnerabilities in mobile web apps than we study. Jin et al. [23] built a tool to analyze mobile web apps developed using PhoneGap for an XSS-like vulnerability where untrusted data retrieved from channels like SMS and barcodes is used to construct scripts that are rendered in a WebView. Wang et al. [30] examined how malicious web content can force inter-app communication and exploit apps (whether they are mobile web apps or traditional apps) that register custom URL patterns. This vulnerability can be seen as the inverse of the leaky URL vulnerability, where a untrusted URL load is forced upon an app rather than a trusted URL load leaking to an untrusted app. The authors manually analyzed a small set of popular apps and found several vulnerabilities. They proposed a defense involving attaching labels to inter-app messages with an origin and enforcing same-origin policies on the communication channel.

VII. Conclusion

While allowing rich interaction between embedded web content and app code, mobile web apps also present security problems beyond those otherwise associated with the mobile platform or the web. We selected several vulnerabilities in mobile web apps and developed scalable analyses to identify these vulnerabilities. We analyzed a large dataset of 998,286 mobile web apps and found that 28% of the apps contained at least one security vulnerability. Our analyses provide a conservative underestimate of the true vulnerability rate for the most common vulnerabilities in our dataset so this result represents a lower bound on the number of vulnerable apps in the wild. We explored trends in vulnerable apps and found that vulnerabilities are present across the entire app ecosystem, including in libraries and the most popular apps. Finally, we listed mitigations that involve changes to the Android APIs that will reduce the frequency and severity of these vulnerabilities.

References

[1] plus.google.com/+AdrianLudwig/posts/1md7ruEwBLF. Acessed: 2015-1-28.

[2] About Apache Cordova. cordova.apache.org/#about. Accessed: 2015-1-28.

[3] Android 5.0 Behavior Changes. developer.android.com/about/versions/android-5.0-changes.html. Accessed: 2015-1-28.

[4] Android Browser Same Origin Policy Bypass. rafayhackingarticles.net/2014/08/android-browser-same-origin-policy.html. Acessed: 2015-1-28.

[5] Android Platform Versions. developer.android.com/about/dashboards/index.html. Accessed: 2015-1-28.

[6] Apktool. code.google.com/p/android-apktool. Accessed: 2014-2-14.

[7] Building Web Apps in WebView. developer.android.com/guide/webapps/webview.html. Accessed: 2015-1-28.

[8] PhoneGap. phonegap.com/about. Accessed: 2015-1-28.

[9] Stack Exchange Data Dump. archive.org/details/stackexchange. Accessed: 2015-1-28.

[10] WebView addJavascriptInterface Remote Code Execution. labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution. Accessed: 2015-1-28.

[11] Ball, T., and Horwitz, S. Slicing Programs with Arbitrary Control-Flow. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging* (1993).

[12] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (2012).

[13] Barth, A., Jackson, C., and Mitchell, J. C. Securing Frame Communication in Browsers.

[14] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Symposium on Network and Distributed System Security* (2012).

[15] Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P. Oauth demystified for mobile application developers. In *Proceedings of the 21st ACM Conference on Computer and Communications Security* (2014).

[16] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services* (2011).

[17] CHIN, E., AND WAGNER, D. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Proceedings of the International Workshop on Information Security Applications* (2013).

[18] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference*. 2010.

[19] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (2012).

[20] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[21] GEORGIEV, M., JANA, S., AND SHMATIKOV, V. Breaking and Fixing Origin-Based Acces Control in Hybrid Web/Mobile Application Frameworks. In *Proceedings of the 21st Symposium on Network and Distributed System Security* (2014).

[22] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Symposium on Network and Distributed System Security* (2012).

[23] JIN, X., HU, X., YING, K., DU, W., YIN, H., AND PERI, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[24] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications security* (2012).

[25] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Atacks on WebView in the Android System. In *Proceedings of the 27th Computer Security Applications Conference* (2011).

[26] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., ACKER, S. V., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (2012).

[27] SOUNTHIRARAJ, D., SAHS, J., GREENWOOD, G., LIN, Z., AND KHAN, L. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 19th Symposium on Network and Distributed System Security* (2014).

[28] TENDULKAR, V., AND ENCK, W. An application package configuration approach to mitigating android ssl vulnerabilities. In *Mobile Security Technologies* (2014).

[29] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot: A Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research* (1999).

[30] WANG, R., XING, L., WANG, X., AND CHEN, S. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *Proceedings of the 20th ACM Conference on Computer and Communications Security* (2013).

[31] WEISER, M. Program Slicing. In *Proceedings of the 5th International Conference on Software engineering* (1981).