An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications

Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, and Min Yang, *Fudan University;* Xiaofeng Wang, *Indiana University, Bloomington;* Long Lu, *Northeastern University;* Haixin Duan, *Tsinghua University*

https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-xiaohan

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications

Xiaohan Zhang[1,4], Yuan Zhang[1,4], Qianqian Mo[1,4], Hao Xia[1,4], Zhemin Yang[1,4], Min Yang[1,2,3,4], Xiaofeng Wang[5], Long Lu[6], and Haixin Duan[7]

[1]*School of Computer Science, Fudan University*
[2]*Shanghai Institute of Intelligent Electronics & Systems*
[3]*Shanghai Institute for Advanced Communication and Data Science*
[4]*Shanghai Key Laboratory of Data Science, Fudan University*
[5]*Indiana University Bloomington* , [6]*Northeastern University* , [7]*Tsinghua University*

Abstract

Mobile apps have become the main channel for accessing Web services. Both Android and iOS feature in-app Web browsers that support convenient Web service integration through a set of *Web resource manipulation APIs*. Previous work have revealed the attack surfaces of Web resource manipulation APIs and proposed several defense mechanisms. However, none of them provides evidence that such attacks indeed happen in the real world, measures their impacts, and evaluates the proposed defensive techniques against real attacks.

This paper seeks to bridge this gap with a large-scale empirical study on Web resource manipulation behaviors in real-world Android apps. To this end, we first define the problem as *cross-principal manipulation* (XPM) of Web resources, and then design an automated tool named XPMChecker to detect XPM behaviors in apps. Through a study on 80,694 apps from Google Play, we find that 49.2% of manipulation cases are XPM, 4.8% of the apps have XPM behaviors, and more than 70% XPM behaviors aim at top Web sites. More alarmingly, we discover 21 apps with obvious malicious intents, such as stealing and abusing cookies, collecting user credentials and impersonating legitimate parties. For the first time, we show the presence of XPM threats in real-world apps. We also confirm the existence of such threats in iOS apps. Our experiments show that popular Web service providers are largely unaware of such threats. Our measurement results contribute to better understanding of such threats and the development of more effective and usable countermeasures.

1 Introduction

Nowadays, different Web services are usually integrated together to provide users with more flexible and powerful capabilities. These integrated services are mostly delivered to the mobile platform today, with multiple services built into a single app. For the convenience of such an integration, mainstream mobile platforms (including Android and iOS) feature in-app Web browsers to run Web content. Examples of the browsers include *WebView* [9] for Android and *UIWebView/WKWebView* for iOS [8, 10]. For simplicity of presentation, we call them *WebViews* throughout the paper.

Based on WebViews, mobile systems further provide app developers with *Web resource manipulation APIs* to customize browser behaviors and enrich Web app functionalities. For example, Android and iOS both have an API named *evaluateJavascript* that allows host apps to inject JavaScript code into the Web pages and get the result. However, these Web resource manipulation APIs lack origin-based access control, which means application code can manipulate Web resources from all origins managed by the WebView through these APIs. For example, if a host app has a WebView which loads "www.facebook.com", then it can use *evaluateJavascript* API to run JavaScript in the Facebook Web pages and get user data from Facebook. As a result, this capability of cross-origin manipulation would lead to severe security and privacy threats to user data.

Some previous work have discussed this kind of threats in the context of integrating WebView to mobile apps. Luo et al. [32, 33] showed that malicious apps can attack WebView by injecting JavaScript code, sniffing and hijacking Web navigation events [32], and hijacking touch events at the Web pages [33]. Chen et al. [16] and Mohammed et al. [43] also demonstrated OAuth protocol can be attacked by a malicious app. Meanwhile, defensive mechanisms [41, 43, 20] have also been proposed to regulate the accesses from host apps to Web resources.

Despite the existing works, there lacks an empirical study to understand how severe this problem is in real-world. In fact, none of existing work provides evidences for the presence of such threats. Instead, they discuss the attacks conceptually. Furthermore, existing defensive

systems are evaluated with hand-crafted attack samples, without considering the special requirements in real-world deployment. Overall speaking, lacking such an empirical study may make us misunderstand the impact of the problem and limit the practicalness of proposed solutions.

This paper seeks to perform a large-scale empirical study on real-world apps to systematically understand the existence and impact of such threats. Since Android apps are easy to be collected in a large volume and Android platform dominates the mobile market, our empirical study is based on Android platform.

First, since not all manipulations cause security issues, we need a clear definition about the threat in Web resource manipulation. Inspired by the same-origin policy in Web platforms, we define the threats in Web resource manipulation as `cross-principal manipulation (XPM)`. In our definition, only manipulating code from a different principal to the manipulated Web resource will be flagged as suspicious.

Second, to allow measuring the Web resource manipulation problem on a large scale, we further design a tool to automatically recognize XPM behaviors in real-world apps. The key challenges are that: there are multiple principals inside an app; there is no obvious way to extract the principal of the manipulating code; it is hard to determine whether the principal of the manipulating code and that of the manipulated Web resource are the same. Our proposed tool, named XPMChecker, features several new techniques to automatically recognize XPMs in apps. Note that XPMChecker is not aimed to reliably detect all possible cross-principal manipulations. Instead, it is designed for a large-scale measurement study. Thus, we do not consider a future attacker who tries to evade XPMChecker.

Finally, we apply XPMChecker to analyze 80,694 apps from 48 categories in Google Play. Our evaluation shows that XPMChecker achieves high precision and recall in recognizing XPM behaviors. To systematically understand the threats of Web resource manipulation, we conduct several experiments and studies from these perspectives: the prevalence of the XPM behaviors, the breakdown of XPM behaviors, the awareness of such risks to service providers and the implications to current defenses. Our study leads to several insightful findings for the community to understand the impact of Web resource manipulation problem, confirms the threat of XPM behaviors with real-world samples and calls into rethinking of existing defensive mechanisms.

Findings. We find that 49.2% of manipulation points are cross-principal, 4.8% of apps have XPM behaviors, 63.6% of cross-principal manipulation points originate from libraries, and more than 70% of XPM points manipulate top popular Web services. We also find that most of XPM behaviors are necessary to improve the usability for mobile users, some XPM behaviors implement OAuth implicit flow in an unsafe way, and we confirm the Web resource manipulation behaviors with obvious malicious intents for the first time in real-world Android apps and iOS apps. More specifically, we find apps can abuse Web resource manipulation APIs to steal cookies, collect user credentials and impersonate the identities of legitimate parties, and a large number of users have been affected. We also perform several experiments to test the awareness of such risks to service providers, and find that most Web service providers are unaware of these risks and can not effectively prevent users from accessing sensitive pages in WebView. Finally, our measurement results also actuate us to rethink existing defensive mechanisms and propose new suggestions for future defense design.

In summary, we make the following contributions.

- We define the threats in Web resource manipulation as cross-principal manipulation (XPM), and perform a large-scale study of such threats in real-world apps.
- We design an automatic tool which overcomes several non-trivial challenges to identify cross-principal manipulations in Android apps.
- We present new results and findings based on a study of 80,694 apps. Our results provide strong evidences for the presence of XPM behaviors with obvious malicious intents in real-world apps, and show that this problem is more severe than we think and exists in both Android and iOS. Our findings and evaluations on current defense mechanisms also bring new insights for future defense design.

2 Web Resource Manipulation

This paper seeks to understand the threats of Web resource manipulation in real-world apps. Although this kind of threats have been conceptually described in existing work [32, 33, 43, 16], none of them systematically defines this problem. To support a large-scale measurement study, we need to clearly define the threats in Web resource manipulation.

2.1 Motivating Example

We use a motivating example to ease the illustration of the security issues during Web resource manipulation. As shown in Figure 1, there are two apps, where app A is the official Facebook app and app B is a standalone chatting app called "Chatous". App B incorporates Facebook Login SDK to support user login with their

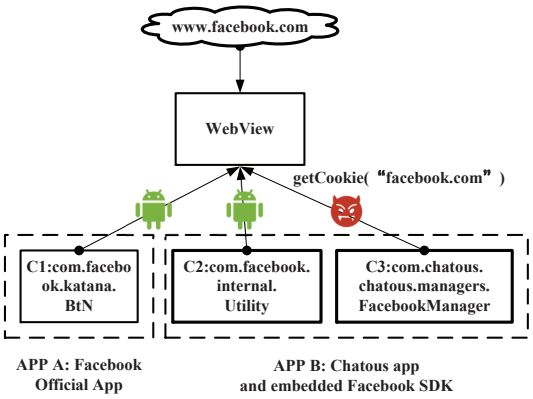


Figure 1: A motivating case where three classes in two apps use *CookieManager.getCookie* API to get cookies from *www.facebook.com*.

Facebook accounts. There are three Java classes (C1, C2 and C3) in the two apps which use WebViews to load *www.facebook.com* and use *CookieManager.getCookie* API to get cookies from *www.facebook.com*.

For C1 which belongs to the official Facebook app and C2 which belongs to the official Facebook Login SDK, it is quite normal for them to access cookies from *www.facebook.com*. However, since C3 belongs to "Chatous" which is a different party to Facebook, it is quite susipicious for C3 to get cookies from *www.facebook.com*. After a manual inspection on C3, we confirm that "Chatous" abuses Facebook cookies to collect user data in Facebook (more details are discussed in Section 4.3.3).

The insight of this example is that when Web resources are manipulated by app code, if the manipulating code and the manipulated Web resource belong to the same party, it can be regarded as quite normal. However, if they do not originate from same party, it may bring threats to the manipulated Web resources.

2.2 Problem Definition

The above example demonstrates the threats when Web resource manipulation APIs are used by a security principal to manipulate Web resources belong to another security principal. To clearly define this problem, this section introduces some new concepts.

Cross Principal Manipulation. We define where app code use Web resource manipulation APIs to manipulate Web resources as `Web Resource Manipulation Points`. At each Web resource manipulation point, there are two participated parties, i.e. the manipulating code and the manipulated Web resource. We designate the security principal of the manipulating code as `App Principal (AP)`, and the security principal of the manipulated Web resource as `Web principal (WP)`. Inspired by the same-origin policy in Web platforms, we study the threats in Web resource manipulation by considering both the app principal and the Web principal. Specifically, we define the concept of `Cross-Principal Manipulation (XPM)` of Web resources, when the app principal is not the same as the Web principal at a Web resource manipulation point. According to its definition, whether a Web resource manipulation point (named as *mp*) is XPM can be recognized with the following equation.

$$IS\_XPM(mp) := AP_{mp} \neq WP_{mp} \quad (1)$$

Threat Model. This paper studies the threats in Web resource manipulation. We consider the host app is not trusted, i.e. it may attack the Web resources by stealing sensitive data, breaking code/data integrity, etc. In our threat model, there are two kinds of attackers in the host app: the host app itself and the incorporated third-party libraries/SDKs. We assume the underlying operating system and Java runtime is trusted and not compromised. A fraudulent attacker may use low-level techniques such as directly manipulating the process memory, to evade analysis and detection. However, we do not consider such low-level attacks that may be performed by host apps, since Web resource manipulation APIs are widely supported by mainstream mobile platforms. This paper focuses on measuring the security impact of Web resource manipulation APIs in real-world applications, while does not aim to study all kinds of threats in app-web interaction, which has been well-studied by existing work [32, 33, 17, 23, 36, 48].

Besides, we only consider Web resource manipulation problem in apps using system-provided Web browsers, i.e. WebView on Android and UIWebView/WKWebView on iOS. Certainly, host apps may use hybrid frameworks such as Cordova [1] or customized browsers such as customized Chromium [7], to integrate Web services. Considering WebViews has standard interfaces, good compatibility and widely used by most apps, our study mainly focuses on WebView platform. Actually, a similar definition of cross-principal Web resource manipulation can be given for these hybrid platforms.

2.3 Web Resource Manipulation APIs

Figure 1 gives an example of Web resource manipulation using *CookieManager.getCookie* API in Android platform. However, the cross-manipulation problem is not specific to this API and not limited to Android platform. Actually, both Android and iOS provide plenty of Web resource manipulation APIs that can be used by the host apps to manipulate the integrated Web resources,

Table 1: Representative Web resource manipulation APIs on Android and iOS.

Web Resources	Android WebView	iOS UIWebView	iOS WKWebView
Local Storage	CookieManager.getCookie	NSHTTPCookieStorage	WKWebsiteDataStore
Web Content	loadUrlJs,[1] evaluateJavascript	stringByEvaluatingJavascriptFromString	evaluateJavascript
Web Address	onPageFinished, shouldOverrideUrlLoading	\	\
Network Traffic	shouldInterceptRequest	shouldStartLoadWithRequest	decidePolicyForNavigationAction, decidePolicyForNavigationResponse

[1] void loadUrl(String url) is an API that loads the given "url". However, it can also be used to load JavaScript into the Web page when the "url" is some JavaScript code. In this paper we only consider the latter usage as Web resource manipulation API, and name it "loadUrlJs" to differ from the former usage.

including quite sensitive resources, such as local storage and network traffic.

To better understand the impact of the problem of cross-principal Web resource manipulation, we perform a thorough study of the WebView APIs provided by Android and iOS platform. According to the type of the manipulated Web resources, we classify these APIs into the following four categories and select some representative APIs for both platforms in Table 1.

1. *Local Storage Manipulation* APIs. WebView may keep sensitive data on the local storage of the device, such as HTTP cookies, Web Storage[1] and Web SQL Database. For example, attackers can use *CookieManager.getCookie(String url)* to get the cookies for any domain specified by "url".

2. *Web Content Manipulation* APIs. Web content includes HTML, JavaScript and CSS of Web sites. For example, attackers can use *evaluateJavascript* API to inject JavaScript code into Web pages and get the privileges of the injected domain.

3. *Web Address Manipulation* APIs. Web address is the current URL for the WebView which contains quite sensitive information. For example, attackers can use *shouldOverrideUrlLoading(WebView view, String url)* to intercept the URL and extract the access token for OAuth implicit flow authorization.

4. *Network Traffic Manipulation* APIs. These APIs can provide attackers with the ability to monitor/-modify network traffics between the WebView and the remote server.

From Table 1, we can conclude that both Android and iOS provide powerful APIs for developers to manipulate quite sensitive Web resources. A study about how these APIs are used by developers is quite urgent to help us understand its security implications in real-world. Considering that Android is the most popular mobile platform and convenient to collect a large volume of apps, we base our empirical study on Android.

[1]Web storage includes localStorage and sessionStorage (see `https://www.w3.org/TR/webstorage/`). This paper refers any data saved on the device by a WebView as "Local Storage", not only the data saved by HTML5 localStorage API.

3 XPMChecker

To support a large-scale empirical study of Web resource manipulation behaviors in real-world apps, this paper designs an automatic tool, named XPMChecker to recognize this behavior in apps. This section first describes the challenges met in automatically checking of cross-principal manipulation behaviors and then details the design of XPMChecker.

3.1 Challenges and Ideas

According to the definition of XPM, we need to check whether app principal and Web principal are the same. However, it is non-trivial to automatically recognize cross-principal manipulation of Web resources. It at least faces the following challenges.

- *Vague App Principal*. According to same-origin policy, the security principal of a Web resource is identified by a triple (i.e. protocol, host, port). However, there lacks a way to name the security principal of app code. Meanwhile, host apps often incorporate third-party libraries and SDKs, making it quite challenging to identify the principals for different app code.

- *Naming Diversity*. Web principal and app principal are extracted from different sources and use different naming conventions for their identity, thus two kinds of naming diversity are introduced: polymorphism and abbreviation. Polymorphism is that the Web resource and app code may come from the same provider but they use different terms as their identities. Abbreviation is also very common, e.g. both "facebook" and "fb" represent the same company. Obviously, it is a huge challenge to

correctly determine whether the Web principal and app principal represent the same party.

Main Ideas. After manually analyzing several apps with Web resource manipulation behaviors, we learn some insights to design XPMChecker. Basically speaking, our solution is composed of the following two ideas.

- *Using code identity information to indicate app principal*. Although there is no existing identifiers to represent app principal, we find some indicators extracted from the code can represent app principal. For example, we can use Java package name, app name, etc. Furthermore, we could recognize third-party libraries in an app and use different app principal indicators based on their code.
- *Leveraging search engine to compare Web principal and app principal*. It is hard to automatically determine whether a Web principal and an app principal belong to the same party. Our idea is to leverage search engine knowledge. The insight is that the search results for a Web principal and an app principal should be highly related if they belong to the same party.

3.2 Design Overview

Based on the above ideas, we design and implement XPMChecker which is capable of automatically recognizing XPM behaviors in real-world Android apps. Figure 2 presents the workflow of XPMChecker. Overall speaking, XPMChecker is composed of the following three key components.

- *Static Analyzer* accepts an Android APK file as input, locates all possible Web resource manipulation points and collects manipulation information for each manipulation point. The manipulation information include the manipulated Web URL and manipulating context. *Static Analyzer* records all the information into a database for further analysis.
- *Principal Identifier* identifies `Web Principal` and `App Principal` for each manipulation point with the manipulation information in the database.
- *XPMClassifier* gives a final decision about whether a Web resource manipulation point is cross-principal or not by leveraging nature language processing techniques and search engines.

Since our study mainly targets Android, XPMChecker is implemented for Android. Similarly, our methodology also works for other platforms such as iOS. We present the details of XPMChecker in the following.

Table 2: The selected 9 Web resource manipulation APIs to study.

API	Manipulated Web Resource	API Type
CookieManager.getCookie	Local Storage	I
loadUrlJs, evaluateJavascript	Web Content	II
onPageFinished, onPageStarted, onLoadResource	Web Address	II, III
shouldOverrideUrlLoading[1]	Web Address	III
shouldOverrideUrlLoading[2]	Network Traffic	III
shouldInterceptRequest	Network Traffic	II, III

[1] boolean shouldOverrideUrlLoading (WebView view, String url), before API level 24.
[2] boolean shouldOverrideUrlLoading (WebView view, WebResourceRequest request), after API level 24.

3.3 Static Analyzer

The static analyzer first finds all the manipulation points for each input APK file, and extracts the manipulated Web URL and manipulating context for each manipulation point. The static analyzer is implemented based on Soot framework [28] and Flowdroid [11].

Build ICFG. Each APK file is parsed and then an inter-procedure control flow graph (ICFG) is built. Some Web resource manipulation APIs are actually callbacks that are implicitly called by the system, thus edges representing the implicit invocations are added to the ICFG.

Locate Web Resource Manipulation Point. Web resource manipulation points are located by traversing the ICFG to look for the the signatures of Web resource manipulation APIs. We thoroughly study the official document of Android WebView APIs [9] and their usages in real-world apps. Finally, as listed in Table 2, we choose 9 APIs that manipulate sensitive Web resources to perform the study. In real-world apps, there are some API invocation sites with no manipulated Web resources actually. For example, some apps just override *shouldOverrideUrlLoading* API and call its super method using "super(this)" without any other behaviors. We use a forward data flow analysis to filter out these points.

3.3.1 Extract Manipulated Web Resource URL

It is non-trivial to extract the manipulated URL at each manipulation point, as it is highly dependent on the specific API. We study these manipulation APIs and classify them into the following three basic types.

- Type I. The URL is the parameter for such manipulation API, For example, the manipulated URL for *CookieManager.getCookie(String url)* is its first parameter, as showed in Listing 1.
- Type II. The URL should be extracted from the invoked WebView instance. For example, in Listing 2, the manipulated URL of *evaluateJavascript* is

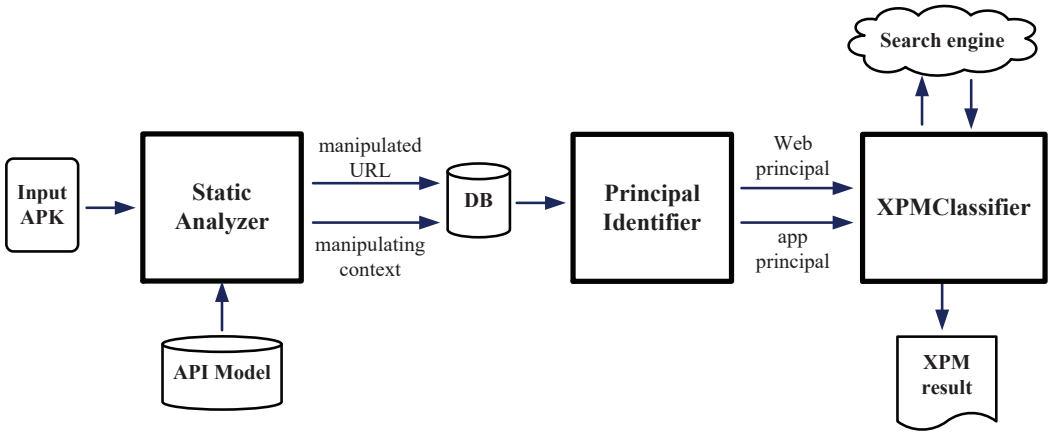


Figure 2: Basic workflow of XPMChecker. XPMChecker is composed of three components to recognize XPM behaviors in Android apps. First, *Static Analyzer* parses input APK files and collects Web resource information into a database. Second, *Principal Identifier* extracts both Web principal and app principal for each manipulation point. At last, *XPM Classifier* recognizes XPM behavior by leveraging search engine knowledge.

the string "www.google.com" loaded by its base WebView instance.

- Type III. The URL is passed as a callback parameter, and can not be statically obtained. Listing 3 shows an example of such API. For *shouldOverrideUrlLoading* API, the "url" is a callback parameter and can only be determined at runtime. However it can be inferred from the code control structure (i.e. the if conditions in line 2 and line 5).

```
1  CookieManager cm = new CookieManager();
2  cm.getCookie("www.google.com");
```

Listing 1: Type I, URL from a parameter.

```
1  WebView wv = new WebView(this);
2  // some code
3  wv.loadUrl("www.google.com");
4  // some other code
5  wv.evaluateJavascript("JS_CODE", ..);
```

Listing 2: Type II, URL from base WebView instance.

```
1  boolean shouldOverrideUrlLoading(WebView
       webview, String url){
2      if(url.startswith("www.google.com"))
           {
3          // some code
4      }
5      else if(url.equals("www.facebook.com
           ")){
6          // some other code
7      }
8      // other code
9  }
```

Listing 3: Type III, URL from a callback parameter.

URL Extraction. Table 2 presents the types for the selected 9 manipulation APIs. We use different methods to extract manipulated Web resource URL according to the API type. For Type I API, the URL is the first parameter of the API. For Type III API, the URL can be inferred from the branch statements in its code. We do a forward data flow analysis from the "url" parameter, and collect all branch statements having string operations with the "url" parameter as the inferred positions.

It is more complicated to handle Type II APIs, where the manipulated URLs are actually loaded by the base WebView instances. There are two cases to determine the URL of the WebView instance: statically loaded URLs and dynamically loaded URLs. Statically loaded URLs are loaded with *LOAD_URL* APIs, including *loadUrl*, *loadDataWithBaseURL*, *postUrl*, etc. In this case, we use the ICFG to find invocations of *LOAD_URL* APIs, and the manipulated URL can be extracted from their parameters. Dynamically loaded URLs are loaded when the users navigates from one page to another. Similar to Type III APIs, the dynamic URLs are inferred from the control flow structure of the code.

String Analysis. After we know the position of the manipulated URL, we then use string analysis to reveal the string value. Specifically, we first do backward slicing along the ICFG to collect all instructions used to construct the URL. Then, we forward traverse the program slice to reconstruct the string-related operations. We try to calculate the string value by modeling common string operations such as initialization and concatenation of *StringBuilder* and *StringBuffer*. Besides, Android-specific APIs such as reading strings from asset files and SharedPreferences are also modeled.

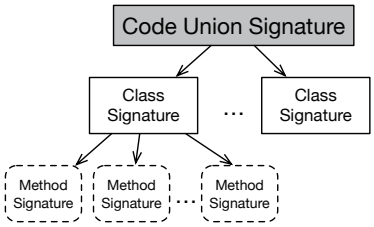


Figure 3: Use Merkle tree to represent manipulating code signature.

Since we focus on integrated Web services, URLs with protocols other than HTTP/HTTPS are not considered and filtered out. Furthermore, there may be more than one manipulated Web URL at one manipulation point, such as the example in Listing 3. These URLs are all extracted and saved into the database for further analysis.

3.3.2 Extract Manipulating Context

To identify the app principal, we need to collect some context information at each manipulation point. Specifically, the following information is collected.

- `META`, the meta-information of the app, including application package name and developer information;
- `DP`, the declaring package name of the manipulating code;
- `SIG`, the signature for the manipulating code;

The `META` and `DP` information can be directly extracted from the APK file and app market. The `SIG` is a signature used to identify the provenance of the manipulating code, i.e. the host app or a third-party library. To calculate the code signature, we first need to determine the boundary of the manipulating code and then extract its signature based on code feature inside the code boundary.

Manipulating Code Union. We introduce the code union concept to represent the code originates from the same principal. Considering the problem context of our paper, we define the code union by grouping code that manipulates the same WebView instance. Specifically, it contains the class of the manipulation point, classes that are connected with the same WebView instance, and classes of the Java objects that have been injected into WebView through `addJavaScriptInterface` API.

Manipulating Code Signature. We use a variant of Merkle trees [35] with depth of 2 to represent the manipulating code signature (as shown in Figure 3). In these hash trees, every non-leaf node is labeled with the hash of its child nodes. The first layer of the tree is the signatures for the classes in the same code union. The second layer of the tree is the signatures for the methods in the parent class. The method signature is calculated by hashing all the Android APIs it invoked. We only consider the Android APIs listed by PScout [12].

When comparing two manipulating code signatures, we first need to judge whether they use the same manipulation API. If they invoke different manipulated APIs, the manipulating code signatures are thought to be different. Otherwise, we compare the Merkle trees for the two manipulating code signatures from top to bottom.

In summary, the static analyzer module locates all manipulation points in each APK, extracts the manipulated URL and manipulating context for each point, and saves this information into a database.

3.4 Principal Identifier

Based the extracted manipulation information at each manipulation point, we further need to identify the Web principal and app principal.

Identify Web Principal. A naive idea is to use the Web origin (a triple of protocol, host and port) as the Web principal. Since the protocol and port element defined in the Web origin are hard to compare with app principal, our solution uses the domain name at each manipulation point as the Web principal.

Before extracting domains from Web resource URLs, we need to normalize the extracted URLs as there may be some abnormal URLs, such as short URL, IP address. The domain names of short URLs and IP addresses can be retrieved by dynamically loading them or resolved with reverse DNS lookup. For domains which are common cloud sub-domains, we extract their domain names as the sub-domains or paths to the host server. For example, for the URL "s3.amazonaws.com/X" or "Y.s3.amazonaws.com", we extract "X" and "Y" as their domains (Web principals).

Identify App Principal. Unlike Web principals, there is no existing way to construct app principal. Our solution is to leverage code features to indicate the security principal of the manipulating code. Generally, manipulating code may originate from two sources: the host app or a third-party library. If the code is from the host app, we use `META` of the app as the app principal indicator. Otherwise we use the declaring package name `DP` instead. Our insight is that Android developers usually include *reverse domain name* in the package name of their code.

To distinguish library code and host app code, we use the signature for the code union (`SIG`). Our observation is that library code tends to appear in many apps. If the `SIG` appears in only one app, or apps from the same developer, the code union belongs to the host app. Otherwise, if it appears in more than one app from

different developers, it originates from a library.

Obfuscated Package Name Recovery. The package name of the library may be obfuscated in an app, thus directly using the package name is not accurate. Considering the fact that not all apps obfuscate their code, we can use non-obfuscated package name of the same library (which has similar `SIG`). In this way, most of the obfuscated package names are recovered for libraries.

Currently, for each manipulation point, we can extract its Web principal and app principal. The next step is to determine whether AP_{mp} and WP_{mp} represent the same security principal.

3.5 XPMClassfier

According to our definition in Equation (1), cross-principal manipulation of Web resources is recognized by judging whether a Web principal and an app principal are the same. However, it is hard to automatically make such decisions. For example, if the app principal is "fb" and the Web principal is "facebook", it is obvious to recognize them as same principal by manual inspection while there is no straightforward way to automatically give the same result.

As it is difficult to strictly tell whether two principals are the same, we perform some relaxation on this problem. Specifically, we transform the strict definition of cross-principal manipulation in Equation (1) into the following definition where *Sim* is the similarity of the two principals. If the similarity proceeds a predefined threshold θ, we think the two principals are the same. Otherwise, the two principals are thought to be different.

$$IS\_XPM(mp) \coloneqq Sim(AP_{mp}, WP_{mp}) \geq \theta \qquad (2)$$

The key to recognize cross-principal manipulation turns to calculate the similarity of two principals. Our idea is to take advantage of search engine knowledge. The insight is that more similar are the two principals, more similar results should be searched for them. Thus, we search the two principals in the search engine, and calculate the similarity between the search results. Specifically, the classification of XPM is performed in the following steps. Note that in rare cases where search engine returns no results, we use literal edit distance between Web principal and app principal to calculate the similarity.

1. Firstly, we remove noise words in $< AP_{mp}, WP_{mp} >$ such as suffixes [5] and stop words [6] (e.g. remove "com" and "get" from "get.appdog.com"), since they make little contribution to XPM classification. After that, we get AP'_{mp} and WP'_{mp}.

2. Secondly, we use AP'_{mp} and WP'_{mp} as search keywords to query Google search engine and get search results as R_{ap} and R_{wp} respectively. All the results are translated into English using Google Translate.

3. Thirdly, we segment the words in the R_{ap} and R_{wp} using the bag-of-words model. Specifically, we only keep the multiplicity and ignore grammar and word order. We normalize each word (term) and transform their term frequencies into two vectors: A and W.

4. Fourthly, we calculate the similarity of the two principals as cosine similarity between the two vectors using the following equation.

$$Sim(AP_{mp}, WP_{mp}) = \frac{\sum_{i=1}^{n} A_i W_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} W_i^2}} \qquad (3)$$

5. Finally, we compare the calculated similarity with a threshold θ. If the similarity does not exceed the threshold, we regard the Web principal and app principal are from different parties and classify the manipulation point (mp) as XPM.

4 Empirical Study

Our empirical study is performed on a large dataset of apps collected from Google Play during July 2017. These apps were selected with at least 5,000 installations across 48 categories, and 84,712 (out of 108,477) apps were successfully downloaded. To the best of our knowledge, this study is the first to understand the Web resource manipulation behaviors with large-scale real-world apps.

Analysis Statistics. We use XPMChecker to analyze these apps on a CentOS 7.4 64-bit server with 64 CPU cores (2GHz) and 188 GB memory. We start 9 processes to parallel the analysis and set timeout of 20 minutes for each app. In all, the analysis takes 233 hours to process the whole dataset, that is about 10 seconds per app. The static analyzer module of XPMChecker successfully processes 80,694 (95.3%) apps, and the rest apps either run out of time or fail to be analyzed by Soot or FlowDroid. For the successfully analyzed apps, XPMChecker finds 13,599 apps with 29,448 manipulation points, and 3,858 of the apps contain 14,476 XPM points. The detailed data is showed in Table 3.

4.1 Evaluation of XPMChecker

Evaluation of Static Analyzer. The static analyzer module is used to find all manipulation points and extract manipulation information (i.e. manipulated Web URL and manipulating context) for further principal

Table 3: Overall result of our study.

Category	#
All Apps	84,712
Finished Apps	80,694
Apps with Manipulation Points	13,599 (29,448)[1]
Apps with XPM Behaviors	3,858 (14,476)

[1] The number in the bracket represents the number of manipulation points.

identification. To evaluate the effectiveness of static analyzer, we randomly select 50 successfully analyzed apps and manually label all the manipulation points for these apps including manipulation information. In total, we manually find 36 manipulated points, while XPMChecker correctly labels 33 of them. The left 3 cases are failed to extract the manipulating Web URLs due to complex string encoding and deep inter-procedure call. As a result, the static analyzer module successfully recall 91.7% of all manipulation points with correctly labeled manipulating information. Further improvement can be achieved by enhancing the string analysis which is a orthogonal research direction [18, 29].

Evaluation of Principal Identifier and XPMClassifier. For each Web resource manipulation point, Principal Identifier extracts the Web principal and app principal, then XPMClassifier judges whether this is XPM by leveraging search engine knowledge. To evaluate the performance of the two modules, we randomly select 1,200 manipulation points identified by the static analyzer, and manually label them as XPM or not. The performance of XPMClassfier depends on the threshold θ. To set θ, we select 1,000 labeled manipulation points from our ground truth and draw the receiver operating characteristic (ROC) curve by trying different thresholds (as shown in Figure 4). Our aim is to gain the balance between false positive rate (FPR) and false negative rate (FNR), so we choose the threshold at the equal error rate (EER) point, that is 0.3134.

We use the left 200 manipulation points to test the performance of Principal Identifier and XPMClassifier. As showed in Table 4, our tool finds 94 XPM points, while 93 of them are true positive. Therefore, the precision and recall of Principal Identifier and XPMClassifier are 98.9% and 97.9% respectively.

We further manually inspect the false positives and false negatives. The cause for the false positives is the lack of search result for some Web principals from small websites. Since these Web sites are not popular, these false positives do not affect the overall result and finding. The false negatives are caused by unofficial apps whose app principals are highly related to those of the official ones. For these cases we need to use more complex techniques to extract app principal. Considering the recall rate is relatively high, we argue current design is quite acceptable to perform a large-scale study.

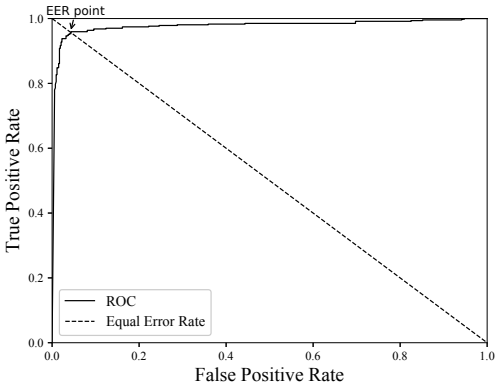


Figure 4: ROC curve for varied θ in XPMClassfier with 1000 manipulation points.

Table 4: Precision and recall of Principal Identifier and XPMClassifier.

# of Manually Labeled XPM	95
# of Detected XPM	94
# of True Positive	93
Precision	98.9%
Recall	97.9%

4.2 Prevalence of XPM Behaviors

This section measures the prevalence of XPM behavior in real-world apps. Our results consist of the following findings.

Finding 1: 49.2% of manipulation points are cross-principal. As shown in Table 3, XPMChecker finds 29,448 manipulation points, while 14,476 of them is crossing principal, which means 49.2% of manipulation points are cross-principal.

Finding 2: 16.9% of apps manipulate Web resources, and 4.8% of apps have XPM behaviors. As shown in Table 3, in all the successfully analyzed 80,964 apps, XPMChecker finds 13,599 apps that contain at least one manipulation points, that is 16.9% of all apps. Further more, XPMChecker finds 3,858 apps have XPM behaviors, which is 4.8% of all apps.

Finding 3: 63.6% of cross-principal manipulation points originate from libraries. As shown in Table 5, our results show that 63.6% of cross-principal manipulation points are from 88 libraries, covering 2,545 apps. Meanwhile, 36.4% of the cross-principal manipulation points belong to 1,414 apps. Note some apps may have

Table 5: XPM point distribution according to its location.

XPM Location	# of XPM Points (%)	# of Apps
Library	9,201 (63.6%)	2,545
App	5,275 (36.4%)	1,414
All	14,476	3,858

Table 6: Top 10 Web hosts that are cross-principal manipulated.

rank	manipulated host	rank	manipulated host
1	play.google.com	6	player.vimeo.com
2	market.android.com	7	maps.google.com
3	facebook.com	8	google.com
4	youtube.com	9	drive.google.com
5	docs.google.com	10	twitter.com

XPM behaviors in both its app code and library code.

Finding 4: More than 70% of XPM points manipulate top popular Web services. We collect the manipulated Web host for all the XPM points and find that more than 70% of them belong to top Web services, such as Google, Facebook and Twitter. We list the top 10 manipulated Web hosts in Table 6.

Finding 5: Web contents and Web addresses are the most commonly manipulated and cross-principal manipulated Web resources. We count the manipulation APIs used for all the discovered manipulation points and present the result in Figure 5. We can see that *loadUrlJs* and *evaluateJavascript* are the most frequently used, which support JavaScript injection into Web pages. Besides, APIs that can manipulate Web addresses, such as *shouldOverrideUrlLoading*, *onPageStarted* are also widely used, rendering that Web addresses are of high interest for manipulating. We find *getCookie* API is quite exceptional because it is widely used in manipulation points but few are cross-principal.

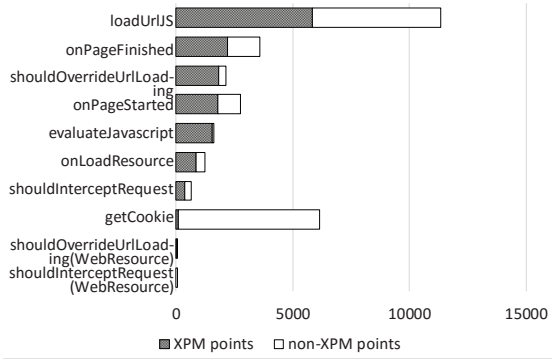


Figure 5: Manipulation API Usage.

4.3 Breakdown of XPM Behaviors

To further understand what XPM behaviors do in real-world apps, we select some apps to study. In all, we manually study all the 88 libraries in Table 5 which cover 63.6% of all XPM behaviors, and randomly select 100 apps from the 1,414 apps. We classify these XPM behaviors and present the results in Table 7.

Table 7: XPM behaviors in 88 libraries and 100 randomly selected apps.

Behavior	% in libraries	% in apps
Customizing Web services	56.8%	67.0%
Invoking local apps	30.7%	16.1%
Obtaining OAuth tokens	2.3%	4.6%
Malicious behaviors	0	0.9%
Other behaviors	5.7 %	8.2%
False positive	4.5%	3.2%

[1] Note that one app may have several XPM behaviors.

We find that the most popular XPM behaviors we found are customizing Web services and invoking local apps. Furthermore, we find several apps exhibiting obvious malicious behaviors, and it is the first time that we can confirm the threat of Web resource manipulation in real-world apps. In the following, we further present our findings in dissecting these XPM behaviors.

4.3.1 Necessary XPM Behaviors

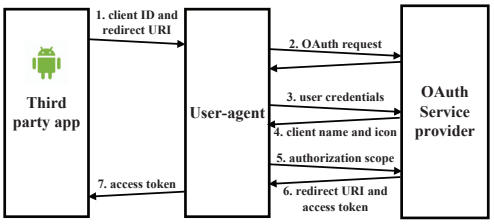
Finding 6: Most of XPM behaviors are necessary to improve the usability for mobile users. Our manual analysis finds that about 90% of the XPM behaviors provide new functionalities. Here we give some examples. Since Android WebView does not support navigation control [2], we find many XPM behaviors inject JavaScript code to add this feature. We also find a library called "Android-MuPDF" which injects JavaScript code into the Google cloud print page to help users reduce the steps in using cloud print. Another common use case of XPM behavior is to invoke local apps. For example, the "org.nexage.sourcekit.mraid" library uses `shouldOverrideUrlLoading` API to monitor the loaded URLs. If the URLs are ads about apps, it will invoke the local "Google Play" app to display the advertised apps.

4.3.2 Unsafe XPM Behaviors

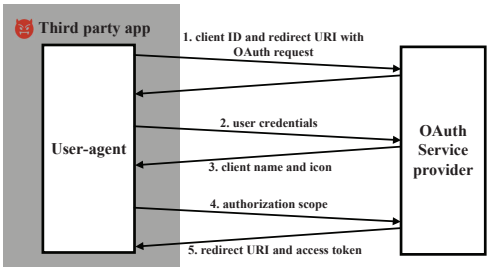
Finding 7: Some XPM behaviors implement OAuth implicit grant flow in an unsafe way. We find some XPM behaviors in 2 libraries and 10 apps implement

OAuth implicit grant flow, but in an unsafe way. Figure 6(a) shows the standard and secure OAuth 2.0 implicit grant flow, where an external user-agent is used and third-party app can only access data in step 1 and step 7. However, we find XPM behaviors are used to implement OAuth implicit flow as depicted in Figure 6(b). Instead of using an external user-agent, the third-party app uses an internal user-agent, i.e. a WebView to do the OAuth implicit grant. Then the third-party app uses Web resource manipulation APIs to intercept the access token from the WebView in step 5 in Figure 6(b). For example, we find a library called "com.magzter" that uses `onPageFinished` API to intercept access token when doing OAuth on Twitter.

According to previous research on OAuth security [41, 16, 43] and RFC OAuth 2.0 specification [4], it is unsafe to use internal user-agent. Specifically, the OAuth 2.0 specification [4] says "native apps MUST NOT use embedded user-agents". The security concern is that using internal user-agent means that the whole user-agent can be controlled by the host app, thus all data in OAuth steps can be manipulated by the host app. As shown in Figure 6(b), data in step 1 to step 5 can all be manipulated by the host app, including client ID and redirect URI, user credentials, client name and icon, authorization scope and access token. All these data are highly sensitive and the leakage or modification on these data can cause severe security problems. Unfortunately, although well-studied and documented, our findings show that insecure OAuth implementations with WebViews are still very common.



(a) standard and secure implicit grant



(b) insecure implicit grant implementation

Figure 6: OAuth 2.0 implicit grant. (a) is the standard and secure implicit grant flow using external user-agents (such as external browsers), where the third-party app can only control data in step 1 and step 7. (b) shows common insecure implementation using internal user-agents such as WebViews, where the third-party app is able to manipulate all data from step 1 to step 5.

4.3.3 Malicious XPM Behaviors

Finding 8: We confirm the Web resource manipulation behaviors with clearly malicious intents for the first time. As shown in Table 7, our study leads to the discovery of some apps with malicious XPM behaviors. To find more malicious XPM behaviors, we analyze more apps in the 1,414 apps that have XPM behaviors. We write scripts to prioritize XPM behaviors that manipulates either top Web services such as Facebook, Google, or URLs contain very sensitive words, such as "oauth", "token", "password". Then we select 200 apps for manual study, and finally we confirm 22 malicious XPM behaviors in 21 distinct apps (listed in Appendix A). Based on their malicious aims, we classify these apps into three categories: impersonating relying party in OAuth (A1, 2 apps), stealing user credentials (A2, 6 apps) and stealing cookies (A3, 14 apps). Note that one app named InstaView exhibits both A1 and A2 behaviors. We have reported these apps to Google Play, and most of these apps have already been removed.

A1: Impersonating Relying Party in OAuth. We find apps impersonate another relying party in OAuth by providing the client ID of the victim in step 1 (see Figure 6(b)) and intercepting access token of the victim in step 5. For example, *Instaview* is a visitor tracking app that tells users who has viewed their *Instagram* account. It has 1,000,000-5,000,000 installations in Google Play. To provide users with the visiting information, it asks users to grant several permissions by OAuth in a WebView. However, it uses the client ID of another app named *Tinder*. After user authorization, it intercepts the access token for *Tinder* using *shouldOverrideUrlLoading* API. After that it continues to impersonate *Tinder* to access user data from the authorization server *Instagram*.

By using the client ID and access token of another app, *Instaview* bypasses registration auditing and resource usage monitoring from *Instagram*. One may think that users would refuse to authorize *Instaview* when they see the permissions are granted to *Tinder*. Actually, we find this app receives more than 27,000 five stars in Google Play. Furthermore, since *Instaview* controls the WebView, it can modify the name and icon in step 3 in Figure 6(b) to cheat users.

A2: Stealing User Credentials. Apps in this category inject JavaScript code to sensitive Web pages, such as login page and OAuth authorization page to steal user credentials. For example, *Adkingkong* is an app for

users to buy advertisements. This app has 500,000 to 1,000,000 installations in Google Play. This app asks users to login with their Google accounts in a WebView. However, when users input their emails and passwords, it uses *loadUrlJs* API to inject JavaScript code into the login page and steals user credentials. The *Instaview* app described above also steals user credentials in step 2 of Figure 6(b) using similar methods.

A3: Stealing and Abusing Cookies. We find several apps using XPM to steal cookies and abuse these cookies. For example, *Chatous* is an app for users to randomly chat with real people. Its installation count is about 10,000,000 to 50,000,000. It incorporates Facebook OAuth SDK for users to sign in with their Facebook accounts. When Facebook official app is not installed on user devices, Facebook SDK uses a WebView to do the OAuth. After user login, Facebook cookies will be saved into the local storage of WebView. We find that *Chatous* gets Facebook cookies using *CookieManager.getCookie* API and directly invokes Facebook APIs using these cookies to get the user friend list and send invitation messages to all the friends of the user. Actually, without Facebook cookies these APIs are invisible to third-party apps such as *Chatous*. We also find other apps from the same developer of *Chatous* exhibit similar behaviors, including *Melon*, *Kiwi*, and *Plaza*. Both *Melon* and *Kiwi* have 10,000,000 to 50,000,000 installations, and *Plaza* has 1,000,000 to 5,000,000 installations.

Finding 9: Malicious XPM behaviors exist on both Android and iOS. For the 21 apps with malicious XPM behaviors, we try to look for their counterparts on iOS platform and successfully find 8 apps have iOS versions. Then we use network traffic analysis to check if they have the same XPM behaviors as their Android counterparts. Finally we confirm the *Chatous* iOS app and other 3 apps from the same developers still have the same malicious XPM behaviors (i.e. stealing and abusing cookies).

Finding 10: Most of malicious XPM behaviors target OAuth. In our results, 18 out of 21 apps with malicious XPM behaviors attack OAuth, indicating that OAuth is the mostly targeted Web service.

Finding 11: Malicious XPM behaviors have affected a large number of users. For the 21 apps with malicious XPM behaviors, we collect their installation count in Google Play. We find that these 21 apps have total installations ranging from 29,885,000 to 131,220,000, which means a lot of users are affected.

5 Implications on Mitigation

Our empirical study shows that the Web resource manipulation capability of WebView brings huge risks to service providers. This section studies the awareness of such risks to service providers and reviews the defensive mechanisms in securing Web service integration.

5.1 Risk Awareness to Service Providers

We study five popular Web service providers (Facebook, Twitter, Google, Weibo and QQ) on whether they prohibit users from accessing login and OAuth pages in WebView. The result is shown in Table 8.

Table 8: Experiments on loading login/OAuth pages of major Web service providers in WebView.

Service providers	Allow login in WebView	Allow OAuth in WebView
Facebook	Y	Y
Twitter	Y	Y
Google	Y	N
Weibo	Y	Y
QQ	Y	Y

We find that these providers all support user login and OAuth in WebViews, except Google who blocks OAuth in embedded WebViews [3]. However, our further study find that Google only uses "USER-AGENT" header to identify WebViews, which can be easily manipulated by host apps. For example, in Android, apps can use *setUserAgentString* API to change the "USER-AGENT" header to any value such as "Google Chrome". We conduct such an experiment and successfully load Google OAuth page in our controlled WebView. Thus, we draw the following conclusion.

Finding 12: Most Web service providers are unaware of risks in Web resource manipulation, and can not effectively prevent users from accessing sensitive pages in WebView.

5.2 Evaluating Defensive Techniques

To secure Web service integration, several techniques have been proposed. Based on our measurement results, we rethink their solutions and conclude several findings.

Finding 13: Complete isolation of WebView is not compatible to most apps. Complete isolation is a common way to protect host program from untrusted code. LayerCake [41] protects the in-app WebView by running WebView in a separate process and seamlessly sharing UI display and events between the host app process and the WebView process. Similarly, AdSplit [44] and AdDroid [37] use process-level isolation to run WebView-based advertisements in separate processes. Although complete isolation is achieved between the host app process and the WebView process, it can not further support WebView manipulation which requires accessing

WebView resources directly in the host process. In our study, we find that most of XPM behaviors are necessary to improve the usability for mobile users (see Findings 6). Thus, though complete isolation improves security, it is hard to apply to existing apps.

Finding 14: Fine-grained access control is a must for regulating Web resource manipulation APIs. Access control is the fundamental way to regulate API usage. To regulate Web resource manipulation APIs, WIREFRAME [20] uses binary rewriting to replace default WebView instances in apps with isolated and mediated WIREFRAME instances. It further provides origin-based access control policy, in which each app is treated as a standalone origin and policies can be expressed as whether an app from origin X can access the Web resources of origin Y. In theory, WIREFRAME is quite useful in preventing the abuse of Web resource manipualation APIs found in our case studies. However, we find the access control mechanism in WIREFRAME is not fine-grained enough because they make the whole app as a single origin, while our Finding 3 shows that more than 60% of XPM behaviors are from libraries. Thus, without fine-grained access control, systems like WIREFRAME are hard to effectively protect Web resources from being abused.

6 Discussion

The cross-principal manipulation problem proposed in this paper is similar to the one faced by Web browser extensions [27, 25], since both mobile apps and browser extensions can manipulate Web resources. The common challenge is how to identify suspicious ones. The most significant difference we observe is that mobile apps may manipulate content from their own servers or others, while most browser extensions are designed to operate on web content of others. Thus, different to vetting suspicious browser extensions, a new challenge met by our work is that we need a fine-grained analysis to recognize whether the host app manipulates his own resources or resources of other parties. Our work makes non-trivial efforts by leveraging static analysis, code similarity and search engines.

Currently, our work has a few limitations. Since our static analyzer is based on several existing static analysis tools [28, 11, 30], XPMChecker inherits limitations of these tools. Besides, XPMChecker can not prevent determined attackers from evading our analysis. For example, they can hide the invocations of Web resource manipulation APIs using Java reflection, or obfuscate the identifiers for recognizing Web principals and app principals. To handle this case, XPMChecker can adopt more sophisticated techniques [31, 14, 13, 39] which is an orthogonal research direction. In this paper, XPMChecker is designed to perform an empirical study rather than to be a detection tool. Our evaluation and study show that it is effective to draw several insightful findings.

Although our empirical study is performed on Android apps, the ideas proposed in this paper also work on iOS platform. Finally, in our study, manual effects are involved to classify XPM behaviors. In the future work, we plan to automatically label the types of XPM behaviors with heuristic rules and learning techniques.

7 Related Work

The interplay between mobile app, embedded browser, and embedded web content is complex and fraught with security concerns. Prior work have discussed these problems in several aspects.

Web-to-App Security. A large number of these works focus on how Web code can attack native apps. Several works point out that malicious JavaScript code from unauthorized Web origin can get sensitive data from the host apps through several ways, including abusing the JavaScript bridge (exported Java functions using *addJavascriptInterface* API) [32, 17, 36, 23], accessing file system [17, 23, 45], abusing HTML5 geolocation API [23] or postMessage API [24]. To detect such malicious Web code, BridgeScope [48] is proposed to precisely and scalably vet JavaScript Bridge vulnerabilities in hybrid apps. Rastogi et al. [40] try to detect and find the provenance of attacks from ad libraries to host apps. Jin et al. [26] study the channels for malicious JavaScript to be loaded by HTML5-based mobile apps. Further more, some defensive mechanisms are also proposed. NoFRAK [22] enforces access control rules for the Web code in Cordova framework, with the help of unforgeable capability tokens from the Web server. Draco [46] provides a uniform and fine-grained access control framework to regulate Web code.

App-to-Web Security. An opposite research direction is to study how host apps can attack Web resources. Luo et al. [32] show that malicious apps can attack Web pages by injecting JavaScript code or sniffing and hijacking Web navigation events. In [33], they also demonstrate that malicious apps can hijack touch events of the web pages. Shehab et al. [43] and Chen et al. [16] focus on the security issues of a certain kind of Web service, i.e. OAuth in mobile apps. When using WebView as the user-agent in OAuth, Shehab et al. [43] show that user credentials and authorization interface may be attacked, while Chen et al. [16] point out that access token sent in redirection URI may be leaked by the host app. However, none of existing work seeks to find such attacks in real-world apps. This paper firstly phrases this threat as cross-principal

Web resource manipulation, then overcomes several non-trivial challenges to design a detection tool, and finally confirms this kind of attack in not only Android apps but also iOS apps.

Furthermore, XPMChecker leverages techniques from several related fields, including static analysis, library detection, and text similarity. The static analyzer module is based on state-of-the-art static analysis tools, including Soot [28], Flowdroid [11] and IccTA [30]. Specifically, we use the intermediate representations provided by Soot [28], build an ICFG for each APK based on Flowdroid [11], and extract inter-component information provided by IccTA [30]. Our method to distinguish library code and app code is inspired by some library detection work [19, 47, 34, 49]. Furthermore, search engine is utilized by the XPMClassifier module to recognize XPM behaviors. Besides, search engine is also widely used in the context of short-text semantic similarity, such as in [38, 21, 42, 15].

8 Conclusion

This paper conducts the first empirical study on Web resource manipulation with large-scale apps. We define the threats in Web resource manipulation as XPM problems. To support automatically recognizing XPM behaviors, we design XPMChecker which overcomes several non-trivial challenges. With a study of 80,694 top Google Play apps, we find that 49.2% of manipulation points are XPM, 4.8% of apps contain XPM behaviors, and more than 70% XPM behaviors manipulate top popular Web sites. More importantly, we confirm the threat of XPM behaviors with obvious malicious intents in both Android and iOS apps. Our further studies actuate us to rethink existing defensive mechanisms and propose new suggestions for future defense design. Besides, to facilitate further research in XPM behaviors, we release the dataset at `https://xhzhang.github.io/XPMChecker/`.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U1636204, 61602123, 61602121, U1736208) and the National Program on Key Basic Research (NO. 2015CB358800). Yuan Zhang was supported in part by the Shanghai Sailing Program under Grant 16YF1400800 and a research gift from Ant Financial. The IU author is supported in part by the NSF 1408874, 1527141, 1618493 and ARO W911NF1610127.

References

[1] Apache Cordova. `https://cordova.apache.org/`.

[2] Building Web Apps in WebView. `https://developer.android.com/guide/webapps/webview.html`.

[3] Modernizing OAuth Interactions in Native Apps for Better Usability and Security. `https://developers.googleblog.com/2016/08/modernizing-oauth-interactions-in-native-apps.html`.

[4] OAuth 2.0 for Native Apps. `https://tools.ietf.org/html/rfc8252`.

[5] Public Suffix List. `http://publicsuffix.org/`.

[6] Stop Words List from Glasgow Information Retrieval Group. `http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words`.

[7] The Chromium Projects. `http://www.chromium.org/`.

[8] UIWebView, Apple Development Documentations. `https://developer.apple.com/documentation/uikit/uiwebview`.

[9] WebView, Android Developers. `https://developer.android.com/reference/android/webkit/WebView.html`.

[10] WKWebView, Apple Development Documentations. `https://developer.apple.com/documentation/webkit/wkwebview`.

[11] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices 49*, 6 (2014), 259–269.

[12] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 217–228.

[13] BICHSEL, B., RAYCHEV, V., TSANKOV, P., AND VECHEV, M. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 343–355.

[14] BODDEN, E., SEWE, A., SINSCHEK, J., OUESLATI, H., AND MEZINI, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)* (May 2011), pp. 241–250.

[15] BOLLEGALA, D., MATSUO, Y., AND ISHIZUKA, M. Measuring semantic similarity between words using web search engines. *www 7* (2007), 757–766.

[16] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 892–903.

[17] CHIN, E., AND WAGNER, D. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications* (2013), Springer, pp. 138–159.

[18] CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. Precise analysis of string expressions. In *International Static Analysis Symposium* (2003), Springer, pp. 1–18.

[19] CRUSSELL, J., GIBLER, C., AND CHEN, H. Scalable semantics-based detection of similar android applications. In *Proc. of ESORICS* (2013), vol. 13, Citeseer.

[20] DAVIDSON, D., CHEN, Y., GEORGE, F., LU, L., AND JHA, S. Secure integration of web content and applications on commodity mobile operating systems. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 652–665.

[21] FITZPATRICK, L., AND DENT, M. Automatic feedback using past queries: social searching? In *ACM SIGIR Forum* (1997), vol. 31, ACM, pp. 306–313.

[22] GEORGIEV, M., JANA, S., AND SHMATIKOV, V. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS symposium* (2014), vol. 2014, NIH Public Access, p. 1.

[23] HASSANSHAHI, B., JIA, Y., YAP, R. H., SAXENA, P., AND LIANG, Z. Web-to-application injection attacks on android: Characterization and detection. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 577–598.

[24] HUANG, J., GU, G., MENDOZA, A., ET AL. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications*, IEEE, p. 0.

[25] JAGPAL, N., DINGLE, E., GRAVEL, J.-P., MAVROMMATIS, P., PROVOS, N., RAJAB, M. A., AND THOMAS, K. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 579–593.

[26] 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), ACM, pp. 66–77.

[27] KAPRAVELOS, A., GRIER, C., CHACHRA, N., KRUEGEL, C., VIGNA, G., AND PAXSON, V. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 641–654.

[28] LAM, P., BODDEN, E., LHOTÁK, O., AND HENDREN, L. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)* (2011), vol. 15, p. 35.

[29] LI, D., LYU, Y., WAN, M., AND HALFOND, W. G. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ACM, pp. 661–672.

[30] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 280–291.

[31] LI, L., BISSYANDÉ, T. F., OCTEAU, D., AND KLEIN, J. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 318–329.

[32] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 343–352.

[33] LUO, T., JIN, X., ANANTHANARAYANAN, A., AND DU, W. Touchjacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security* (2012), Springer, pp. 227–243.

[34] MA, Z., WANG, H., GUO, Y., AND CHEN, X. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion* (2016), ACM, pp. 653–656.

[35] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87* (1987), Springer, pp. 369–378.

[36] MUTCHLER, P., DOUPÉ, A., MITCHELL, J., KRUEGEL, C., AND VIGNA, G. A large-scale study of mobile web app security. *Mobile Security Techologies* (2015).

[37] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. Addroid: Privilege separation for applications and advertisers in android. In *Proc. of AsiaCCS '12*.

[38] RAGHAVAN, V. V., AND SEVER, H. On the reuse of past optimal queries. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval* (1995), ACM, pp. 344–350.

[39] RASTHOFER, S., ARZT, S., MILTENBERGER, M., AND BODDEN, E. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Network and Distributed System Security Symposium (NDSS)* (Feb. 2016).

[40] RASTOGI, V., SHAO, R., CHEN, Y., PAN, X., ZOU, S., AND RILEY, R. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS* (2016).

[41] ROESNER, F., AND KOHNO, T. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium* (2013), pp. 97–112.

[42] SAHAMI, M., AND HEILMAN, T. D. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web* (2006), AcM, pp. 377–386.

[43] SHEHAB, M., AND MOHSEN, F. Towards enhancing the security of oauth implementations in smart phones. In *Mobile Services (MS), 2014 IEEE International Conference on* (2014), IEEE, pp. 39–46.

[44] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. Adsplit: Separating smartphone advertising from applications. In *Proc. of USENIX Security'12*.

[45] SON, S., KIM, D., AND SHMATIKOV, V. What mobile ads know about mobile users. In *NDSS* (2016).

[46] TUNCAY, G. S., DEMETRIOU, S., AND GUNTER, C. A. Draco: A system for uniform and fine-grained access control for web code on android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 104–115.

[47] WANG, H., GUO, Y., MA, Z., AND CHEN, X. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), ACM, pp. 71–82.

[48] YANG, G., MENDOZA, A., ZHANG, J., AND GU, G. Precisely and scalably vetting javascript bridge in android hybrid apps. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2017), Springer, pp. 143–166.

[49] ZHANG, Y., DAI, J., ZHANG, X., HUANG, S., YANG, Z., YANG, M., AND CHEN, H. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 141–152.

A Real-world Malicious Cases

Table 9 lists the detailed information of the 21 malicious apps detected by XPMChecker.

Table 9: Discovered malicious XPM behaviors with different aims: impersonating relying party in OAuth (A1, 2 apps), stealing user credentials (A2, 6 apps), stealing cookies (A3, 14 apps). One app named InstaView exhibits both A1 and A2 behaviors.

Package Name	Installations	Malicious Behavior	Description	APK MD5
com.chatous.chatous	10M-50M	A3	steal Facebook cookies and abuse cookies to send spam messages	d8726437e1f2bbe17257c4eac6707bee
com.chatous.plaza	1M - 5M	A3	steal Facebook cookies and abuse cookies to send spam messages	e5c4ec654e6f97a95ec5eed7afdd961d
com.melonapps.melon	5M - 10M	A3	steal Facebook cookies and abuse cookies to send spam messages	36513949c9fb8dd2f979354ddd058b60
com.chatous.pointblank	10M - 50M	A3	steal Facebook cookies and abuse cookies to send spam messages	a43529aa32363c480abb1cf013d29cdf
com.vendiste.app	100K - 500K	A3	steal Facebook cookies and abuse cookies to send spam messages	5a4c48925fd42f6ee2376f088184e925
com.litefbwrapper.android	100K - 500K	A3	steal Facebook cookies and abuse cookies to receive account's notification	71e290121dddd0099d766685bf89a479
com.instaview.app	1M - 5M	A1 & A2	Impersonating Tinder in Instagram OAuth & inject JavaScript to steal user's Instagram credentials	b354aafb7f86e7ebc629a767d29f886a
com.kingsoft.email	100K - 500K	A2	inject JavaScript to steal user's QQ Email credentials	c3501cbb6f0caa3c2655de2713afad3a
co.kr.adkingkong	500K - 1M	A2	inject JavaScript to steal user's google plus credentials	26fe73ee8a33d2a0112215cf10d97c8b
com.dmf.wall. DMFPanoLwpF	100K - 500K	A3	steal flickr cookies to login automatically	e85a5e17f96ed57c9eb229234f4abaa2
ru.like.vs	100K - 500K	A3	steal vk cookies to request user's information	33c59b3042acc6ffac59bb7e418f7f85
sg.com.singnet .mystorage.android	100K - 500K	A3	use Facebook cookie to reconnect when user logouts	25611dc7d573e43c923f8e51f7835302
com.hlpth.molome	10K - 50K	A2	inject JavaScript to steal Google access token	a3ec6a2e3e5f387a53cdb06a3e48c917
com.weirdlysocial.videoview	10K - 50K	A2	inject JavaScript to steal user's Instagram credentials	7ebccfbd85c8f239b122ea31eb0b318a
com.wierdlysocial.storyview	5K - 10K	A2	inject JavaScript to steal user's Instagram credentials	f4f1f6f644bbca4de637c0b19b94ec1f
com.deltecs.wipro	50K - 100K	A3	use teamgum's clientId in Google SSO and steal access token from cookies	45dfd761e11883c0b225f7dc8edb4b14
com.snapdealhub	500K - 1M	A3	use teamgum's clientId in Google SSO and steal access token from cookies	31139b30a4f92f14a8f9707f74c9b60d
com.ilgan. GoldenDiskAwards2016	100K - 500K	A1	use fengchuanke' clientId for Weibo SSO	78c32007638f64de697dfb473a2a6d0d
com.homedev.locationhistory	100K - 500K	A3	steal Google cookies and save them into sharedpreference	78ca09ff3d1367982c5fb084b8f31734
com.danielstudio.app.wowtu	10K - 50K	A3	steal Weibo cookies and abuse cookie to update photos	aa3dd94becf64647fdf74e2b2aa7b325
com.aol.mobile.aim	1M - 5M	A3	steal Facebook cookies and save them into sharedpreference	80931d076bd5be08e7e1077e31b409e2