Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation

Rui Wang
Microsoft Research
ruiwan@microsoft.com

Luyi Xing
Indiana University
luyixing@indiana.edu

XiaoFeng Wang
Indiana University
xw7@indiana.edu

Shuo Chen
Microsoft Research
shuochen@microsoft.com

ABSTRACT

With the progress in mobile computing, web services are increasingly delivered to their users through mobile apps, instead of web browsers. However, unlike the browser, which enforces origin-based security policies to mediate the interactions between the web content from different sources, today's mobile OSes do not have a comparable security mechanism to control the cross-origin communications between apps, as well as those between an app and the web. As a result, a mobile user's sensitive web resources could be exposed to the harms from a malicious origin.

In this paper, we report the first systematic study on this mobile cross-origin risk. Our study inspects the main cross-origin channels on Android and iOS, including intent, scheme and web-accessing utility classes, and further analyzes the ways popular web services (e.g., Facebook, Dropbox, etc.) and their apps utilize those channels to serve other apps. The research shows that lack of origin-based protection opens the door to a wide spectrum of cross-origin attacks. These attacks are unique to mobile platforms, and their consequences are serious: for example, using carefully designed techniques for mobile cross-site scripting and request forgery, an unauthorized party can obtain a mobile user's Facebook/Dropbox authentication credentials and record her text input. We report our findings to related software vendors, who all acknowledged their importance. To address this threat, we designed an origin-based protection mechanism, called *Morbs*, for mobile OSes. Morbs labels every message with its origin information, lets developers easily specify security policies, and enforce the policies on the mobile channels based on origins. Our evaluation demonstrates the effectiveness of our new technique in defeating unauthorized origin crossing, its efficiency and the convenience for the developers to use such protection.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection – *access controls, invasive software*

General Terms

Design, Experimentation, Security, Standardization.

Keywords

Android, iOS, same-origin policy, mobile platform.

1. INTRODUCTION

The popularity of smartphones, tablets and other mobile devices has brought in a plethora of software applications designed for running on these devices. Such applications, commonly known as *apps*, are typically used to deliver web services (data storage, social networking, web mails, etc.) through their compact user interfaces and simple program logic, which are tailored to mobile platforms. Moreover, other than interactions with their own web services, many of those apps are also built to work with other apps and services, leveraging the third-party's resources to enrich their functionalities. This is a trend that echoes web-API integrations extensively utilized in developing traditional, browser-based web applications. Examples include the apps that support social login and data sharing through the services offered by Facebook, Twitter, Google Plus, etc.

Mobile origin-crossing hazard. Those mobile apps essentially play the same role as traditional web browsers at the client side. However, different from conventional web applications, which enjoy browse-level protection for their sensitive data and critical resources (e.g., cookies), apps are hosted directly on mobile operating systems (e.g., Android, iOS), whose security mechanisms (such as Android's permission and sandbox model) are mainly designed to safeguard those devices' local resources (GPS locations, phone contacts, etc.). This naturally calls into question whether the apps' web resources are also sufficiently protected under those OSes. More specifically, web browsers enforce the *same origin policy* (SOP), which prevents the dynamic web content (e.g., scripts) of one domain from directly accessing the resources from a different domain. When the domain boundary has to be crossed, a designated channel needs to go through to ensure proper mediation. An example is the *postMessage* channel [5], which a domain uses to send a message to another domain by explicitly specifying the recipient's origin, and the browser mediates to ensure that only the content from that origin gets the message and the recipient is also well informed of the sender's origin. Such origin-based protection has become a de facto security standard for a modern browser. However, it is not present on any channels provided by mobile OSes for apps to communicate with each other or the web. As a result, the web content or apps from an untrusted domain may gain unauthorized access to the web resources of other apps/websites through those channels, causing serious security consequences.

As an example, consider the *scheme* mechanism [25][26] supported by Android and iOS, through which an app on phone/tablet can be invoked by a URL once it registers the URL's scheme (e.g., the "youtube" part of "youtube://watch?token=xxx", with "xxx" as the input parameter for the app). What an adversary can do is to post on Facebook a link that points to a malicious script hosted on his website. Once this link is clicked by the victim through the Facebook app on her iOS device, the script

starts to run within the app's *WebView* instance, which is then redirected to a dynamically generated URL with the scheme of another app that the adversary wants to run on the victim's device and the parameters he chooses. As a result, the target app will be invoked to blindly act on the adversary's command, such as logging into his Dropbox account to record the victim's inputs (Section 3.3.2), since the app is given no clue the origin (the adversary's site) of the request. In another case, the Android mobile browser processing a URL with the "fbconnect://" scheme from the Facebook server will deliver the secret token on the URL to an app from an arbitrary origin, as long as it claims to be able to handle that scheme (Section 3.3.1).

Such unauthorized origin crossing is related to the *confused deputy* problem [24] on mobile devices. Prior research on this subject, however, focuses on *permission redelegation* [10], which happens when an app with a permission requires sensitive system resources (e.g., a phone's GPS location) on behalf of another app without that permission. The interactions between the two apps go through an Inter-Process Call (IPC) that delivers a message called *intent* from one app to invoke the other app's *Activity* for services such as getting GPS coordinates. This *intent channel* can also be used to cross origins: for example, it allows an app from one origin to send intents to another app when the latter's related Activity is accidentally made public, a mobile *cross-site request forgery* (CSRF) attack [27]. However, given that those prior studies primarily aim at protecting mobile devices' local resources, the general problem has not been dug deeper: for example, it is not clear whether an app's *private* Activity can still be invoked by the intent message from an unauthorized origin, not to mention the security implications of other channels (such as the URL scheme) that can also be used for crossing domains.

Our findings. To better understand this critical security issue, we conducted the first systematic study on unauthorized origin-crossing over mobile OSes, including Android and iOS. In our study, we investigated all known channels that allow apps to cross domains, such as intent, scheme and utility classes for web communications, by dissecting popular apps like Facebook, Dropbox, Google Plus, Yelp, and their SDKs, to understand how they utilize these channels to serve other apps on different mobile OSes. Our study found 5 generic cross-origin weaknesses in those high-profile apps and SDKs, which can be exploited through CSRF, login CSRF and cross-site scripting (XSS). Many of those problems affect multiple apps and web services. They are unique to the communication channels on mobile OSes, which are fundamentally different from those within the browsers. The root cause of the vulnerabilities is the absence of origin-based protection. More specifically, due to missing origin information, an app or a mobile web service is often left with little clue about the true origin of an incoming message, nor does it have any control on where its outgoing message will be delivered to.

The consequences of these cross-origin attacks are dire. They allow a malicious app to steal the mobile device owner's Facebook, Dropbox authentication credentials, or even directly perform arbitrary operations on her Dropbox account (sharing, deleting, etc.) on Android. On iOS, a remote adversary without direct access to any app on the victim's device can stealthily log the phone into the adversary's Dropbox account through Google Plus, Facebook apps. As a result, the victim's text input on iPhone and iPad, her contacts and other confidential information are all exposed to the adversary once she uses popular editing and backup apps (e.g., PlainText, TopNotes, Nocs, Contacts Backup, etc.) that integrate Dropbox iOS SDK. We reported those

problems to related parties, who all acknowledged the importance of our discoveries. We received over $7000 bounty for these findings, most of which were donated to charity. The details of such recognition together with demos of our attacks are posted online [31].

Origin-based defense. Without any OS-level support, not only does app development become highly error-prone, but software manufacturers can also have hard time fixing the problems after they are discovered. As examples, both Dropbox and Facebook need to spend a significant amount of effort to fix the security problems we discovered, which involves changing software architecture (Section 3.2.1) or deprecating some core features within their apps and SDKs (Section 3.3.1). To address these issues and facilitate convenient development of securer apps, we present in this paper the design of the first mobile origin-based defense mechanism, called *Morbs*. Our approach mediates all the cross-origin channels on Android and iOS, including intent, scheme and the utility classes for web communications, and enables a developer to specify to the OS authorized origins her app/website can receive requests from and/or send responses to.

We implemented Morbs on Android in a way that fully preserves its compatibility with existing apps. Moreover, we show that through our mechanism, the developers can easily gain controls on all cross-origin events, avoiding the ordeal experienced by Facebook, Dropbox, and other companies. Our evaluation on the implementation also shows that it is both effective, stopping all the exploits we found, and efficient, incurring only a negligible impact on performance (< 1% overhead).

The source code of Morbs is publicly available on GitHub [40].

Contributions. We summarize the paper's contributions here:

- *New problem.* Up to our knowledge, the research reported here is the first attempt to systematically understand unauthorized origin crossing on mobile OSes. The discovery made by our study brings to light the presence of such vulnerabilities in high-profile apps and more importantly, the seriousness and pervasiveness of the problem.

- *New techniques.* We developed new origin-based protection for existing mobile OSes, which works with apps/websites to oversee the communication channels on these systems.

- *Implementation and evaluation.* We implemented our design on Android, and evaluated its effectiveness, efficiency, compatibility, and usability to the app developer.

Roadmap. The rest of the paper is organized as follows: Section 2 describes the mobile channels used for apps to communicate with each other or the web; Section 3 elaborates our study on mobile cross-origin problems and our findings; Section 4 presents our defense mechanism Morbs, Section 5 reports the evaluation of our techniques; Section 6 compares our work with other related research; Section 7 concludes the paper and discusses the future research.

2. MOBILE CHANNELS

Today's mobile OSes (including Android and iOS) provide various channels for apps to communicate with each other or the web. Those channels include *intent*, *scheme*, and *web-accessing utility classes* (which we elaborate later in the section). As shown in Figure 1, an app communicates with other apps through the intent or scheme channel. It can also invoke the browser to load a webpage using an intent and be triggered by the web content rendered in the browser through a URL scheme. Moreover, the

app can simply acquire and display any web content through the WebView class, which embeds a browser-like widget within the app, and directly talks to a web server using the methods provided by the HttpClient classes. Note that the intent channel is Android-specific, while others are also available on iOS. Unlike the domain-crossing mechanisms within a browser (e.g., postMessage), these mobile channels are not under the origin-based protection: messages exchanged do not carry any origin information that the sender/receiver can inspect, and are completely unmediated with regard to where they come from.
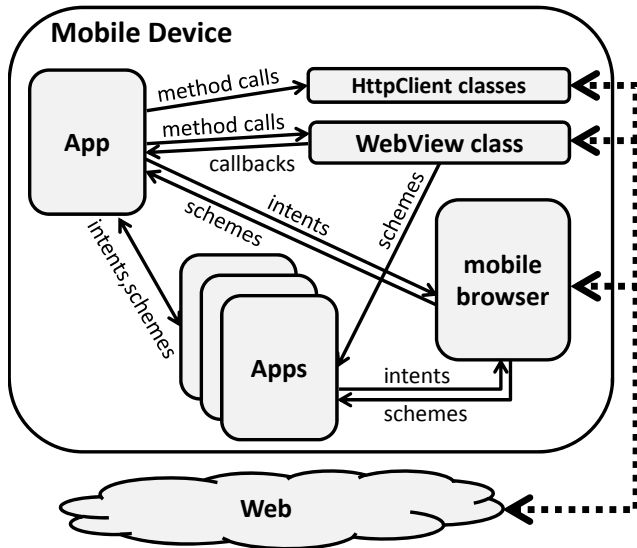

Figure 1 Mobile communication channels

Here we elaborate how those channels work:

- *Intent*. An intent is an inter-process message delivered through an IPC. It is a channel only supported by Android. Through intent messaging, one app on Android can activate the background Services, Activities (application components with user interfaces) or Broadcast-Receivers of another app, as well as the Activities/Services of its own. Intent invocation is conducted through APIs such as *startActivity*, *startActivityForResult*, and *startService*. An app developer can specify a set of intents the app can receive from other apps in its manifest file. However, the intent channel never labels the origin of each message (i.e., who created it). This causes the problem we elaborate in Section 3.2.1.

- *URL scheme*. As discussed before, scheme is supported by both Android and iOS, which allows an app or website to use a URL to invoke another app (on iOS) or its Activity/Service components (on Android) that claim the scheme of that URL. For example, the URL "youtube://watch?token=xxx" can be used to start the YouTube app to play the video "xxx". When such a URL is loaded in the mobile browser or a WebView instance, the OS will launch the target app with this URL as input. In addition, an app can also invoke other apps through the schemes they registered. On Android, scheme invocation is implemented through the intent channel: a scheme URL is wrapped in an intent instance, and invoked by an app through the same set of APIs that also serve intent messages, such as *startActivity*. On iOS, this is done through *openURL* API. Again, the OSes do not mediate the scheme-based invocations using origins.

- *Web-accessing utility classes.* Mobile platforms provide several utility classes for apps to communicate with the web. We call them *web-accessing utility classes*. For example, both

Android and iOS support the WebView class (called UIWebView on iOS), which an app can embed for displaying webpages. An app can interact with its WebView instance through a set of method calls or callbacks. For example, it can call *loadURL* on Android (*loadRequest* on iOS) with a URL to load a page into the instance; it can also register events, like URL loading, to inspect every URL its WebView instance processes through a callback *shouldStartLoadWithRequest* (iOS) or *shouldOverrideUrlLoading* (Android). In addition, the mobile platforms provide utility classes for an app to directly talk to a web server without loading its web content. HttpClient [36] or HttpURLConnection [37] (Android) and NSURLConnection [38] [39] (iOS) are such examples. We call those classes (for direct communication with web servers) *HttpClient classes*. Origin-based protection is not in the picture here: e.g., a WebView/HttpClient instance never labels which app is the origin of an HTTP request.

3. ATTACKS
In this section, we elaborate our study on unauthorized origin crossing on mobile OSes. What we want to understand here are whether the ways real-world apps utilize those channels for cross-origin communications indeed expose them to attackers, and whether those apps have proper means to mitigate such a threat and safeguard their operations over those channels. For this purpose, we systematically analyzed high-profile apps on both Android and iOS, including the official apps of Facebook and Dropbox and their SDKs, and the official Google Plus and Yelp app. Note that these SDKs are very popular. They have been integrated into many real-world apps. Problems discovered there may have a broad impact. In our research, we looked into how those apps use the aforementioned cross-origin channels to interact with other apps, or the web. The study reveals the pervasive presence of subtle yet serious cross-origin vulnerabilities, allowing an unauthorized party to activate an app remotely with arbitrary input parameters, call its internal Activities, steal user's authentication credentials and even directly manipulate its operations.

Such discoveries were made through an in-depth analysis on the code and operations of those apps. Specifically, for Android apps, we decompiled the binary code of their latest versions using apktool [33] and AndroChef Java Decompiler [34] in order to analyze their program logic related to the mobile channels. When it comes to iOS apps, decompiling their executables is often hard. Therefore, we resorted to a black-box traffic analysis to understand those apps' interactions with other parties (apps, web services, etc.). We also studied the SDKs provided by Facebook and Dropbox, whose source code is publically available. In the rest of the section, we report our findings. The demos of our exploits on those apps and other supplementary materials are posted on the web [31].

3.1 Adversary Model
Our adversary model describes practical threats to different mobile platforms. On Android, we consider an adversary who can trick a user into installing a malicious app on her device. That app, however, may not have any permission considered to be dangerous by Android. Also, threats to Android can come directly from the web, when the victim uses her mobile app or browser to view malicious web content posted by the adversary on a website. On iOS, we only consider this remote threat (from a malicious website), not the malicious app, given the fact that Apple's vetting process on iOS apps is more restrictive than that of Android apps, and few malicious apps have been reported so far.

Note that we treated Android and iOS differently to respect the realistic threats those systems face: we could have found more issues had we assumed the presence of malicious apps on iOS. Finally, we do not consider an adversary with OS-level controls.

3.2 Exploiting the Intent Channel

The security implication of the intent channel on Android has been studied in prior research [10][27]. All existing work, however, focuses on how such a channel can be leveraged by a malicious app to invoke a legitimate app's Activities that are accidentally made public by the app's developer. In our research, we found that even the private Activities not exposed to the public, which is meant to be called only by the app itself, can be triggered by an app from an unauthorized origin. This problem has a serious consequence, letting the malicious app gain great control of the victim app. We discovered this vulnerability on both the Facebook app and the Dropbox app. Here we use the Dropbox app as an example to explain where the problem is.

3.2.1 Next Intent (Android)

An Android app can have two types of Activities, private or public. By default, an Activity is private, meaning that only the code inside the app can invoke it. When the app developer sets the "exported" property of the Activity to true, or she declares at least one intent for the Activity in the manifest of the app, the Activity becomes public, in which case other apps can invoke the Activity with an intent instance as an argument.

Our analysis on the Dropbox app reveals that the app exposes a few Activities, such as login, which is meant to be public. An interesting observation is that when any of its public Activities are invoked by an intent instance, the Activity first needs to check whether the user is in a logged-in status. If not, it redirects him to the login Activity before proceeding with its own task. Specifically, the Activity creates a new intent instance, in which the current intent, the one it receives from another app, is saved under the key "com.dropbox.activity.extra.NEXT_INTENT" (called "NEXT_INTENT" here). The new intent instance is then issued by the app itself to invoke LoginOrNewAcctActivity (the login Activity). Once the user completes her login, the login Activity retrieves the original intent instance from "NEXT_INTENT", and uses it to invoke the unfinished public Activity to fulfill its task.

The cross-origin exploit. It turns out that this next-intent feature can be exploited by a malicious app to cross origins and invoke the Dropbox app's private Activity. Since the login Activity is public, a malicious app can trigger it with an intent instance, in which the attacker puts another intent instance under the "NEXT_INTENT" key. The second instance points to a private Activity of the Dropbox app. This login intent will not be noticed by the user if she is already in the logged-in status, and cause little suspicion if she is not, simply because it is the authentic Dropbox app that asks the user to log in. Either way, once the login is done, LoginOrNewAcctActivity retrieves the intent content under the "NEXT_INTENT" key and use it to call the startActivity API. Since startActivity is now invoked by the Dropbox app itself, all of its Activities, including those private ones, can be executed, even though the next-intent actually comes from a different origin, the malicious app. The root cause of this problem is that the startActivity API never checks (and also has no means to check) the provenance of the intent under the "NEXT_INTENT" key, due to the lack of origin-based protection on the mobile OS. In the absence of the origin information (here, the app creating the

intent), even an app's private Activity can be exposed to unauthorized parties.

The problem goes beyond a single app. In the Facebook app, we discovered the same problem in a public Activity called UriAuthHandler. The Facebook app also checks the login status, and directs the user to the login Activity, and uses "CALLING_INTENT" (equivalent to "NEXT_INTENT") as a key to store the current intent instance. This channel is equally vulnerable and can be abused in the same way, as found in our study. We suspect that other apps with this type of next-intent feature are also subject to the same exploit.

Attacks and consequences. Once the origin is crossed illegitimately, the door is open to all kinds of abuses. In our research, we implemented two attacks (one against the Dropbox app, another one against the Facebook app) to demonstrate the serious security consequences of the problem.

Our attack on the Facebook app leverages a private Activity LoggedOutWebViewActivity. The Activity takes a URL as an input parameter and loads the content pointed by the URL into a WebView instance. What can happen here is that a malicious app running on the same device can drop a Javascript file onto its SD card (Secure Digital memory card) and exploit the next-intent feature to run LoggedOutWebViewActivity with the URL pointing to that Javascript file. Since the SD card is viewed as a local storage by Android, the script is allowed to access contents from all Internet domains [32]. Specifically in our attack, the script injected can make arbitrary AJAX requests to facebook.com and read the contents of the responses. Given that all such requests carry the user's Facebook cookie, this cross-origin scripting ends up allowing the adversary to perform arbitrary operations on the user's account, and obtain all private data.

For the Dropbox app, we exploited a private Activity VideoPlayerActivity, which has an input parameter "EXTRA_ METADATA_URL" that specifies a URL from which to fetch the metadata for a video file. In a normal situation, this URL points to a file kept by dropbox.com. However, our next-intent exploit enables a malicious app to set the URL to arbitrary web domain, such as "http://attacker.com". When the Dropbox app makes a request with that URL, it always assumes the recipient to be dropbox.com and attaches to the request an authentication header, as opposed to applying the conventional origin-based cookie policy. Since right now, EXTRA_METADATA_URL points to "http://attacker.com", the adversary gets the header and can use it to gain a full access to the user's Dropbox account.

Vendor responses. Fixing this problem turns out to be much more complicated than it appears to be. Specifically, the Dropbox security team told us they were "*working on changing the architecture in our Android app to make that API secure*", but the next-intent feature is "*unfortunately also very useful for us*". Facebook also said that this problem "will take some time to fix". As an acknowledgement to the importance of our work, Facebook awarded us $5000 bounty for finding this vulnerability, which we donated to charity. Dropbox also awarded us 100GB free storage for each author, and included our names on their special thanks page. The details of those software vendors' responses can be found here [31]. From our communications with the vendors, it can be seen that addressing this next-intent problem from the developer side alone can be hard. In Section 4, we show how a well-thought-out OS-level support can make this type of flaws more convenient to fix.

3.3 Abusing the Scheme Channel

As discussed in Section 2, scheme is an important cross-origin channel supported by both Android and iOS. Through this channel, an app on those OSes can be invoked by a URL (with the scheme the app claims) from another app or from a webpage in a WebView instance or a browser (see Figure 1 in Section 2). In our research, we found that this channel can be easily abused for unauthorized origin crossing, enabling a malicious app to acquire a user's authentication token with Facebook or perform a login CSRF on iOS, as described below.

3.3.1 Fbconnect (Android)

Facebook provides a *Dialog mechanism* [35] through its apps and SDKs for both Android and iOS. Using the mechanism, an app can send through the Facebook official app a Dialog request to query the Facebook server for sensitive user data (e.g., access token) or operate on the user's account (e.g., sharing a post). Among all the arguments carried by the Dialog request are *client_id*, the ID assigned to the sender app by Facebook, and *redirect_uri*, which is set to "fbconnect://success". In the case that the user's access token is requested, the Facebook server displays a dialog within Facebook app's WebView instance to ask for the user's consent, and upon receiving it, the server redirects the WebView instance to "fbconnect://success#...", where the secret token is attached to the "…" part of the message. This token is then extracted by the Facebook app, which later dispatches it to the target app (i.e., the sender of the Dialog request) associated with the *client_id*.

The URL "fbconnect://success#..." is just used for delivering data from the Facebook server to its official app. However, if this URL is loaded in the mobile browser, a serious attack can happen. More specifically, a malicious app on the device first registers this fbconnect:// scheme, and then invokes the browser to load a Dialog URL, in an attempt to request the sensitive data of another app (e.g., the TexasHoldem app) from the Facebook server. This can be easily done by setting *client_id* in the URL to that of TexasHoldem because an app's client_id is public. Also, within the browser, the dialog may not even show up to alert the user, if it is already in the logged-in status. As a result, Facebook will redirect the browser to "fbconnect://success#...". Unlike the Facebook app, the browser treats this URL as a scheme invocation, and therefore will trigger the scheme's handler (i.e., the malicious app) with the URL as an argument. This exposes to the malicious app the victim's Facebook secret token for the TexasHoldem app. We tested the attack on an Android device (Galaxy Tab 2) and confirmed that the malicious app can get the user's access token, authorization code and other secrets. In this case, we can see that although the Facebook server is the sender of the scheme message, it cannot control which app to receive the message through the mobile browser. This is different from what happens within a web browser: for example, if a script from a.com sends a message to b.com through the postMessage API, it can specify the recipient domain and the browser then guarantees that only b.com gets the message. On today's mobile OS, however, there is no way that the Facebook server can specify the authorized receiver of its scheme URL, not to mention any mechanism to enforce this security policy.

Also note that the fbconnect problem here is present on both Android and iOS. However, given that iOS malware is rare, the risk it poses is mainly to Android (see our adversary model).

Vendor response. Without the OS support, this problem turns out to be even harder to fix than the next-intent issue. We reported it

to Facebook on Sept. 11, 2012. On Jan. 22, 2013, Facebook security team told us that they took steps to "ensure that popular app stores block apps that attempt to register this URI schema". Moreover, they were "crafting a formal deprecation plan for the fbconnect schema", but this plan needs a "several month deprecation period" because "all of our embedded SDKs currently depend upon this functionality". On March 20, 2013, Facebook informed us that they "crafted a plan for the deprecation of the fbconnect schema in the next major release". They expect to "see this disappear entirely as users continue to upgrade". They awarded us a bounty of $1500 for this finding, which we donated to charity.

3.3.2 Invoking Apps from the Web (Android and iOS)

In this section, we elaborate a new security threat that comes from a malicious website the user visits with a mobile device. The root cause of the problem has been confirmed to exist on both Android and iOS. For the simplicity of presentation, here we just use iOS as an example to explain the problem.

Mobile apps typically use their WebView instances to render web content. Such content could come from less trustworthy web sources, such as public posts on Facebook and restaurant reviews from the strangers on Yelp. In our research, we found that during such rendering of web content, whenever the WebView instance of an app is directed by the content to a URL with a scheme registered by another app on the same device, the latter will be automatically invoked, without being noticed by the user, and act on the parameters given by the URL. This is dangerous because the app receiving the scheme message which carries the URL cannot distinguish whether this message comes from the sender app itself or from the web content within the app's WebView instance, which causes the confusion about the message's true origin. Here we use two examples to show the consequences of this confusion.

Login CSRF attacks on Dropbox iOS SDK. We studied the latest version (v.1.3.3) of Dropbox iOS SDK, which enables a 3rd-party app on iOS to link to the device owner's Dropbox account, using Dropbox as the app's storage. Here, we use PlainText [41], a popular text-editing app, as an example to explain what can go wrong, though apps using Dropbox iOS SDK are all vulnerable. Specifically, after the mobile user authorizes this account linking, the Dropbox app delivers to the PlainText app a scheme URL, which is in the following format: db-<APP_ID>://1/connect?oauth_token&oauth_token_secret&uid.
The URL includes 3 arguments, *oauth_token*, *oauth_token_secret*, and *uid*, which the PlainText app uses to communicate with the Dropbox server to complete the account linking. However, we found that the linking process can be exploited to launch a serious login CSRF attack, *without* any malicious app running on the user's device. Specifically, in our attack, the adversary uses the 3 URL arguments collected from his own device to build a URL: db-<APP_ID>://1/connect?oauth_token'&oauth_token_ secret' & uid', where *oauth_token'*, *oauth_token _secret'*, and *uid'* are the adversary's Dropbox credentials, and APP_ID identifies the PlainText app. The attacker shares a malicious web URL (e.g. pointing to attacker.com) on his Google Plus status updates or comments. Once the victim user clicks it within the Google Plus app on her device, attacker.com is loaded in the app's WebView instance, and redirects the WebView to the scheme URL. As a result, the PlainText app is invoked with the URL as input. The app treats the URL as part of the scheme message from the Google Plus app, without knowing that it actually comes from the

web content of attacker.com rendered in the Google Plus app's WebView. It is then unknowingly linked to the attacker's Dropbox account. When this happens, the app automatically sends the user's text input to the attacker's account. A demo of this attack is posted online [31]. We also checked a few other popular iPad apps using Dropbox SDK, including TopNotes, Nocs, and Contacts Backup to Dropbox. They are all found vulnerable.

Bypassing Facebook's app authentication mechanism. Many apps using Facebook iOS SDK, such as Yelp and TripAdvisor, may also render untrusted web contents within their WebView instances. Below we show that an attacker who posts a malicious link on Yelp can bypass an important mechanism Facebook uses to authenticate 3rd-party iOS apps. Specifically, when app A invokes the Facebook app through schemes such as "fbauth://", the Facebook app sends the app ID specified by app A and its bundle ID retrieved from the OS to the Facebook server for authentication. This prevents app A from impersonating another app to communicate with the Facebook server because it cannot manipulate the bundle ID. However, this protection does not work when a malicious page is loaded to the WebView instance of the Yelp app because the Facebook app cannot distinguish whether an incoming scheme message is from the Yelp app or a webpage in its WebView (the bundle ID from the OS always points to the Yelp app). Therefore, whoever posts a comment on Yelp acquires the same privilege as Yelp has on the victim's Facebook account.

Vendor response. We reported these problems to Dropbox, Google, and Facebook. For the first problem (login CSRF through Dropbox SDK), Dropbox started its investigation immediately after receiving our report. They have implemented a fix that needs to change both the SDKs and the Dropbox official apps on all platforms (including Android and iOS). Facebook mitigated the threat by deploying a whitelist inside the WebView instance of its official app, which only allows http, https, and ftp schemes. Google has not taken any actions so far [31]. Facebook awarded us $1000 for this finding. We also reported to Facebook the second case (bypassing its app authentication mechanism on iOS), which is still under investigation.

3.4 Attacks on Web-Accessing Utility Classes
As shown in Figure 1, besides intent and scheme, origins can also be crossed on a mobile OS when an app directly calls the methods of the WebView/HttpClient classes or registers their callback events. Here we show how this channel can be abused.

3.4.1 Exploiting Callbacks (iOS)
On iOS, we studied a WebView callback method the Facebook app registers, *shouldStartLoadWithRequest*, which is triggered each time the app's WebView instance is navigated to a link. If this link is in the form "fbrpc://appID=xyz&foo=123", the callback method (provided by Facebook) creates a new URL "fbxyz://foo=123" to invoke an app with the appID "xyz" and set its input argument to "123". Note that this operation is different from the scheme-based invocation (from a web domain) described in Section 3.3.2, as in that case, a website directly runs a URL to invoke the target app on the mobile device (the sender of the scheme message is the website), while here such a URL is actually created by the callback method, which is implemented by the Facebook app (the sender is the Facebook app).

This mechanism can be exploited when a malicious link such as attacker.com is clicked by the user through her Facebook app. When this happens, the malicious content loaded to the WebView

instance redirects to the fbrpc URL. Then the callback of the Facebook app generates a new scheme URL to launch any app the adversary wants to run on the victim's device with any argument value he sets. For example, we found that in this way, a popular app *Pinterest* can be activated by the adversary to sign onto the adversary's account on the victim's device, so as to dump the user's data with Pinterest into the adversary's account.

3.4.2 Exploiting Header-attachment (Android)
We also studied the HttpClient class, which is used by Android apps for direct HTTP communications with web servers. HttpClient allows a developer to specify the URL of a request and an HTTP header. The header is attached to the request. In the absence of origin-based protection, any header, including the one used for authentication, can be attached to a request sent to any website. A prominent example is the attack case described in Section 3.2.1: the adversary invokes the Dropbox app's Activity VideoPlayerActivity, which utilizes an HttpClient instance to load metadata from a URL with the user's authentication header attached. Since the URL is manipulated by the adversary to point to attacker.com, without origin checks, the authentication header goes to the adversary.

Note that this header-attachment issue by itself is a security flaw, as admitted by the Dropbox security team ("*Attaching the Authorization header to non-Dropbox URLs was definitely a serious security bug*" [31]). Actually the attack on a phone user's Dropbox account as described in Section 3.2.1 is built upon two security flaws, i.e., the next-intent and header-attachment issue.

4. ORIGIN-BASED DEFENSE
As described in the prior section, unlike web browsers, today's mobile OSes (i.e., Android and iOS) do not offer origin protection to the channels used by apps to communicate with each other or the web. As a result, cross-origin interactions on those systems can be easily abused to undermine the user's security and privacy, which even happens to highly popular apps built by security-savvy developers. Moreover, even after the problems were reported, the developers still had hard time in fixing them. This makes us believe that a generic solution to the problem should be built into mobile OSes, which have the observations of messages' origins, and the means to mediate the communication over those channels. In this section, we elaborate the first design for such protection, called *Morbs* (mobile origin based security), and its implementation on Android. We released the source code of the Android implementation on GitHub [40].

4.1 Design
Overview. Morbs is generic to iOS and Android. It is designed to achieve browser-like origin protection: 1) it exposes to the developers the true origins of the messages their apps/websites receive, enabling them to build protections based on such information; 2) it allows the developers to specify their intentions, in the form of whitelists of origins their apps/websites can get messages from and send messages to, and enforces policies transparently within the OS.

More specifically, an app or a web service that asks for origin protection first communicates its intended sender/recipient origins (the whitelists) to the OS. These policies are enforced by a reference monitor that mediates different mobile channels. The reference monitor is triggered by the messages delivered through these channels. Once invoked, it identifies the origins of the messages, which are either apps or web domains, and checks their

policy compliance against the whitelists. Those running against the policies are blocked by Morbs.

A unique feature of Morbs is its capability to connect web activities (within WebView instances or the mobile browser) to the events that happen within the OS. For example, it exposes the true origin of a message to a recipient app when confusion arises on whether the message comes from another app or the web domain visited by that app's WebView instance. It also helps a web server specify to a mobile device a designate app on the device that can receive the server's scheme message. This capability is crucial for defeating unauthorized origin crossing on mobile devices. Following we elaborate our design.

Defining mobile origin. For web content, an origin is defined as a combination of scheme, host name, and port number [4]. However, this definition is insufficient for the origin protection on mobile platforms: here we need to consider not only web origins but also app identities and other local origins. To maintain the consistency with the web origins, we adopted a URL-like representation for those new origins, such as "app://<appID>", where <appID> is an app's package name (Android) or bundle ID (iOS), for example, "app://com.facebook.katana" for the Facebook app on Android and "app://com.getdropbox.Dropbox" for the Dropbox app on iOS. Likewise, messages from trusted sources like the OS are given a local origin "local://". For web domains, we adhere to the traditional origin definition [4].

Exposing origins. When a message is created by an app/website, Morbs sets the *origin* attribute (added by our approach) of the message to its creator (i.e., an app, a web domain, or local). This attribute always goes with the message within the OS, until it gets to its recipient app/website, where we remove the attribute. To help developers build their own protection, our design exposes the origin of a message through existing and new APIs. For example, on iOS, we can enhance the API for retrieving the bundle ID of the sender of a scheme message by returning the true origin of the message, which could be the domain of the web content within that app's WebView instance. In this way, Facebook will be able to find out that the scheme message it gets from the Yelp app actually comes from a webpage Yelp displays, not the app itself. Therefore, the exploit described in Section 3.3.2 will be defeated.

Default policy. It is well known that the browser by default enforces the SOP to the web content it hosts, but the same policy cannot be applied to all the apps on mobile platforms as it may disrupt their legitimate cross-origin communications. Our strategy is to implement the SOP only on the totally unexpected and insecure channel. An example is the next-intent communication described in Section 3.2.1, which is unexpected, since the private Activity of an app should only be invoked by the app's own intent when calling the startActivity API. Therefore, in this scenario, the SOP is always enforced.

Setting policies. Morbs allows a policy to be specified on a channel between an app and a web domain (*web policy*), as well as between two apps (*app policy*). An app policy defines legitimate inter-app communication, which goes through intent or scheme. A web policy is about app-web interactions, through scheme or web-accessing utility classes. An app or a website sets a policy on a specific channel to notify Morbs the list of senders authorized to send messages to it, and the list of recipients allowed to receive the messages it sends.

Setting a policy can be done through a new API *setOriginPolicy*, which an app can directly call. Here is its specification:

```
void setOriginPolicy(type, senderOrRecipient, channelID, origins)
```

Here, *type* identifies the type of the channel (intent, scheme, or utility class), *senderOrRecipient* specifies sender or recipient, *channelID* indicates the channel ID, and *origins* is the whitelist. Once invoked, setOriginPolicy first identifies a channel by *type*, and *channelID*, which is an OS-wide unique string. For example, the ID for the intent channel that triggers LoginActivity within the Facebook app is "com.facebook.katana.LoginActivity", in which "com.facebook.katana" is the Facebook app's package name. For a scheme, its channel ID is the corresponding scheme field on a URL. For web-accessing utility classes, they are identified by their class instances within an app. After a channel is identified, the API then extracts the whitelist that regulates the sender or the recipient (specified in *senderOrRecipient*) through this channel from the *origins* parameter.

Although *setOriginPolicy* offers a generic interface for policy specification, it cannot be directly invoked by a website to set its policies. To address this problem, Morbs provides mechanisms for indirectly accessing this API, including a JavaScript API *setMobileAllowedOrigins*, through which the dynamic content from a website can set policies within the mobile browser or a WebView instance, and a header *mobile-allowed-origins* in HTTP responses that inform the browser or a WebView instance of the parties on the device allowed to get the message. The app developer can also leverage other indirect mechanisms to define her policies whenever she is building the app's functionality over a mobile channel. Specifically, Morbs allows the developer to set her policies regarding a scheme/intent her app claims within the app's manifest file (for Android) or .plist (for iOS), under a new property *allowedOrigins*. In this way, she can turn on our origin protection for her app without changing its code. Other ways for policy setting include a new argument *allowedOrigins* for the API that delivers scheme/intent messages, and a new API *setAllowedOrigins* used to define policies for utility classes such as WebView and HttpClient. An advantage of using these indirect ways is that they only require one argument (i.e., *origins*) from the developer because other arguments are set by default.

Enforcing policies. Morbs runs a reference monitor to enforce security policies on different channels. Whenever a message is delivered over a channel, the reference monitor is triggered to identify its origin and calls a function *checkOriginPolicy* to check its policy compliance. The function's specification is as follows:

```
bool checkOriginPolicy(type, senderOrRecipient, channelID, from, to)
```

Intuitively, the function searches Morbs policy base to find out whether the current sender (specified in the *from* argument) is allowed to deliver the message to the recipient (*to*) through the specific channel (*type and channelID*). Note that *checkOriginPolicy* needs to be called twice (one for checking the sender origin against the recipient's policy and the other for checking the recipient origin against the sender's policy). The message is allowed to go through only if both checks succeed.

Both *setOriginPolicy* and *checkOriginPolicy* operate on the Morbs policy base that keeps all policies. *setOriginPolicy* inserts a policy into the database and *checkOriginPolicy* searches the database for an applicable policy, checking whether a sender/recipient origin is on the whitelist included in the policy. The performance of this compliance check is critical because it needs to be invoked for every message going through these

channels. To make it efficient, Morbs leverages the hash-table search to quickly locate a target within the database.

4.2 Implementation

We implemented our design on Android (Figure 2). At the center of our system are the *setOriginPolicy* API and the ReferenceMonitor class, in which the most important function is *checkOriginPolicy*. They were built into the Thread class of the Dalvik Virtual Machine. The *setOriginPolicy* API is open to all apps, while ReferenceMonitor is kept for the OS, which is only accessible to the code running inside the Android OS kernel.
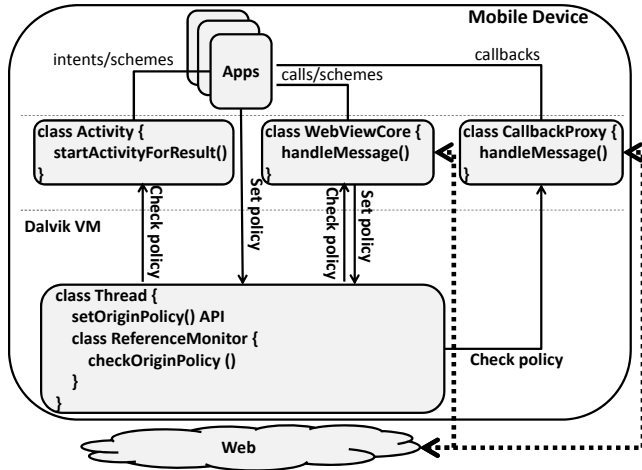

Figure 2 The framework of Morbs on Android

In the presence of the centralized policy compliance check (*checkOriginPolicy*), the task of ReferenceMonitor (i.e., policy enforcement) becomes trivial: all we need to do here is pulling the arguments, invoking *checkOriginPolicy* and raising an exception to drop a message when the check fails. In our implementation, the ReferenceMonitor class is used in the OS components related to these channels to conduct mediation. Specifically, for intent and scheme, the enforcement code was placed within the API *startActivityForResult*, which needs to be called by *startActivity*, when a message delivered through those channels attempts to start an Activity. Note that we chose not to do the security checks within the IPC mechanism: Android does not recommend inspecting intent data in IPC because the intent instances are serialized for high performance data transport [29]. For mediating web-app communications, we changed the *handleMessage* method within both the WebViewCore class and the CallbackProxy class. The two methods are the focal point of mobile browsers and WebView instances: all method invocations and callback handling from apps need to go through them. In addition, the *execute* method of HttpClient class was used to mediate apps' direct communication with web servers.

Challenge I: origin identification. Morbs attaches an *origin* attribute to every message exchanged through the mobile channels. On Android, both intent and scheme channels utilize the intent messages (Section 2). The constructor for generating an intent instance is hooked in our implementation to label an intent message with its app origin. Specifically, we added an *origin* property to the intent class. When the constructor is creating a new intent instance, it retrieves the package name of the app initiating the intent and fills in the *origin* property with the package name. For example, when the initiator is the Facebook app, the *origin* property should be marked as "app://com.facebook.katana", in which "com.facebook.katana" is the package

name of Facebook app. However, this origin is not easy to identify in practice, since there is no API to help us find out the initiator directly. A simple solution is to get the whole call stack from the OS through *getStackTrace* API, and then inspect it to find out the caller. This approach turns out to be very expensive: in our test, extracting the call stack takes 1.35 ms in average. Our solution is to add an *origin* property to each thread that hosts an app. When the thread is created, the app's origin is attached to the property. Once an intent is initiated, the constructor then copies the origin information from the thread to the intent instance.

Challenge II: response inspection. To enable a web server to set its policies to a mobile device, Morbs needs to inspect the HTTP response to find the header *mobile-allowed-origins*. The response is processed by Android's native C++ libraries. Morbs (written in Java) cannot directly access it. In our implementation, we managed to get access to the header using Java Native Interface (JNI) [30]. JNI provides an API called *onReceivedData* through which C++ code can send messages to Java code. To inform Morbs of the content of the header, we modified the C++ code to identify the header *mobile-allowed-origins* within HTTP responses, and then call *onReceivedData* to deliver all policies described there to WebViewCore, where Morbs uses *setOriginPolicy* to complete this policy setting process.

5. EVALUATION

We evaluated the general design of Morbs and its implementation on Android to understand its effectiveness, performance, compatibility and utility to the app developers.

5.1 Effectiveness

We ran our implementation against the aforementioned cross-origin attacks (Section 3). Specifically, our experiment was conducted on Android 4.3 with Morbs running within an emulator. We installed both the vulnerable apps discovered in our research and the attacker apps. In the experiments, we first ran the attacker apps, and then checked whether the exploits were blocked by Morbs or not. Note that in some situations, we also need the developers to explicitly specify their whitelists of origins within their apps, in addition to the default policies. In the absence of those apps' source code, we had to directly enter those app-specific policies (whitelists) into the OS.

Preventing the exploits on intent. As described in Section 3.2.1, a malicious app can use the next-intent trick to invoke any private Activities of the victim app (the Dropbox app and the Facebook app). The content saved under the NEXT-INTENT key is essentially an intent, which needs to be first created by the malicious app before it is embedded into another intent (the one delivered to Dropbox app's login Activity). Under Morbs, the intent constructor sets the origins of both intents to the malicious app, which cannot be changed by the app. As a result, when *startActivity* is called to start the target private Activity, our reference monitor immediately finds that the origin of the next-intent is not the victim app itself, and stops this invocation according to the default policy (the SOP). Our tests confirmed that the vulnerabilities in both Dropbox app and Facebook app are fixed in this manner, without changing the apps' code.

Defeating the attacks on scheme. For the fbconnect problem described in Section 3.3.1, what Facebook wants to do is to return the data (e.g., secret tokens) from its server to the app associated with the *client_id* parameter within the Dialog request, not anyone that claims the fbconnect:// scheme. This intention is

communicated by the Facebook server to the mobile OS through a list of legitimate recipient origins specified using its HTTP response header or the JavaScript API provided by Morbs. Specifically in our experiment, we inserted the header "mobile-allowed-origins: app://com.facebook.katana" into the HTTP response from the Facebook server, indicating that only the Facebook app can receive the data, and observed that the scheme invocation was stopped when the app that registered the fbconnect:// scheme was not the Facebook app. A video demo about this case can be found at [31].

When it comes to the apps using Dropbox iOS SDK (the first vulnerability described in Section 3.3.2), it is clear that their schemes associated with Dropbox are only supposed to be invoked by the Dropbox app. Using Morbs, the Dropbox SDK embedded in the apps specifies "app://com.getdropbox.Dropbox" (i.e., the Dropbox app) as the only legitimate sender origin for these schemes. As a result, our reference monitor ensures that any invocation of the schemes comes only from the Dropbox app, which defeats the attacks from a malicious webpage (through Facebook app or the Google Plus app). In the case of the attack using the Yelp app (second vulnerability described in Section 3.3.2), the problem comes from that the recipient of the scheme message, the Facebook app, cannot tell whether the origin of the message is indeed Yelp or a malicious website visited by the Yelp app's WebView instance. In the presence of Morbs, the true origin of the message is revealed as the website's domain, which enables Facebook to thwart the attack. Note that we did not actually run those fixes as the problems were found on iOS.

Mediating issues in web-accessing utility classes. For the case of the WebView callback within the Facebook app (Section 3.4.1), this callback should only respond to the event (i.e., processing the fbrpc URL) when this URL comes from the domains under Facebook's control. Let's assume Facebook app specifies "https://*.facebook.com" as the whitelist associated with the callback class UIWebViewDelegate, an operation that can be easily done using Morbs. As a result, the event initiated from "attacker.com" is ignored by the WebView, without triggering the callback *shouldStartLoadWithRequest*, while those from facebook.com continue to be handled. We further evaluated our implementation against the exploit on the HttpClient class (Section 3.4.2). This time, we set "https://*.dropbox.com" as the legitimate origin on the whitelist for the instance of the HttpClient class used in the Dropbox app. After that, we found that even after the adversary crossed the origins through the next-intent channel, he still cannot steal the Dropbox authentication header by sending requests to a non-dropbox.com URL, because it was blocked by our reference monitor according to the whitelist.

5.2 Performance

We evaluated the performance of our implementation on a Nexus 4 development phone. We compared the overhead of Morbs with the overall delay the user experiences in the absence of Morbs, to understand the performance impact that our approach can have on cross-origin communications. In the experiments, we call a Java API *nanoTime* to collect timing measurements at a precision of 1 nanosecond (i.e., 10^{-9} s). To measure the performance of a Morbs operation, we repeated it 10 times to get its average execution time. The operations we studied include setting policies and checking policy compliance. Among them, the compliance check is the focus of our evaluations, as the policy setting is just a one-time task. More specifically, we measured the delays for sending messages through intent, scheme, and utility classes in the absence of Morbs, and then compared them with the time spent on

a policy compliance check. In all the cases, the impact of Morbs was found to be negligible (below 1%).

Performance of Morbs operations. On the Android OS with our Morbs implementation, we ran a test app to invoke the *setOriginPolicy* API, and measured the time for setting a policy. On average, this operation took 0.354 ms, which involves storing the content of the policy to a policy database maintained by the OS. To check the compliance with the policies, Morbs needs to search the database to find out whether the origin of the current sender or recipient is whitelisted. As described in Section 4.1, we leverage the hash-table search to quickly locate the policies. To understand the performance of this operation, we utilized a test app to invoke another test app through an intent message, which triggered the *checkOriginPolicy* function. We found that the whole compliance check process took 0.219 ms on average. Note that policy enforcement over other channels all utilizes the same ReferenceMonitor class, which is expected to bring in similar average delay.

Impacts on mobile communications. As described above, the performance impact of setting policies should be minimum, since it just incurs a one-time cost. Also for the policies declared within a manifest file, they are set when the app is installed, which does not affect its operations. Therefore, our focus was policy compliance check.

In the study, we measured the overall delays for sending a message through intent, scheme, and web-accessing utility classes without the policy compliance check. Table 1 shows the average delays for such communication, and their comparison with the overhead for a compliance check (0.219 ms). This gives a pretty good picture about the impact the check can have on such channels. Specifically, for the intent channel, we measured the time interval between the invocation of *startActivity* and the execution of *performCreate* (the first API the target Activity needs to call). After repeating the operation for 10 times, we observed an average delay of 42.142 ms when the sender and recipient were the same app, and 46.267 ms when they were not (see Table 1). On the other hand, the compliance check took only an average 0.219 ms. Therefore, the impact of this mediation on the intent communication was around 0.5%. For the scheme message delivered between two apps, it goes through the same intent mechanism. The mediation impact of Morbs on this communication was found to be 0.3% on average. We further measured the time a webpage takes to invoke an app through scheme, between the event when the method *handleMessage* in WebViewCore class is triggered to process the scheme URL, and when the *performCreate* API for the target test app is called. We found that this whole process took 115.301 ms and the impact of the policy checking there was 0.2%.

When we take into account the delays incurred by web-related operations, particularly those performed by the methods and callbacks of WebView and HttpClient class, the extra time spent on the policy compliance check can be comfortably ignored. Specifically, we measured the waiting time for loading a URL (specifically, google.com) through HttpClient, and WebView. For HttpClient, we measured the time interval between the creation of a class instance and the point when the instance receives the HTTP response, which took 225.035 ms on average. For WebView, we measured the interval between the start of page loading (*onPageStarted* is called) and its completion (*onPageFinished* is called), which took 692.955 ms. By

comparison, the time Morbs spends on the compliance check (0.219 ms) become unnoticeable.

Table 1 Impact of policy compliance check

Channel	Type of Communication	Communication Delay w/o Morbs (ms)	Impact of Morbs policy checking
intent	in-app	42.142	0.52%
	cross-app	46.267	0.47%
scheme	app-app	64.077	0.34%
	web-app	115.301	0.19%
utility classes	HttpClient	225.035	0.10%
	WebView	692.955	0.03%

5.3 Compatibility and Developer's Effort

An important goal of Morbs is to maintain compatibility when possible and minimize the developer's effort to use its protection. Following we elaborate our study on these two issues.

Compatibility. To see whether our implementation can work with existing apps, we loaded Android with Morbs into a Nexus 4 development phone and evaluated the operations of top 20 free apps downloaded from Google Play market. Those apps were first analyzed: we disassembled their binary code and found that all of them use intent, 12 claim various schemes and all need to use the web through WebView and HttpClient classes. We then analyzed their functionalities in the presence or absence of Morbs mediation, by clicking on all buttons and using all of the services we could find. During the test, we did not observed any deviation of those apps' behaviors with and without our mechanism.

Developer's effort. As discussed before, to use Morbs, the developer only need to specify her whitelists through the interfaces (e.g., the *setOriginPolicy* API) we provide, which is straightforward for them to act on. This is compared to the case-by-case fixes that app developers are currently doing in response to our vulnerability reports. In Table 1, we give a comparison with regard to the vulnerabilities described in Section 3. The ways they are fixed (or to be fixed) ("Fix w/o Morbs") come from our conversations with corresponding software vendors. Here, how to fix the problem 4 (the exploit through Yelp app) and 5 (the callback loophole) is still unknown.

As we can see from the table, these vulnerabilities are much easier to fix with the support of Morbs. Specifically, for the next-intent problem (Section 3.2.1), both Dropbox and Facebook informed us that an effective fix takes time to build. Particularly, Dropbox explained that they need to "change the architecture" of their app, which involves non-trivial effort. In the presence of our origin-based protection, however, this next-intent cross-origin loophole is fixed without requiring any modification to the apps. As another example, for the fbconnect issue described in Section 3.3.1, Facebook chose to deprecate the use of fbconnect, which is a core feature in all of its native SDKs and official apps. This effort needs "a several month deprecation period", according to Facebook. Using Morbs, however, Facebook could easily fix the problem without touching any of its SDKs and apps, by simply adding an extra header, including the origins of the apps supposed to receive its message, to the HTTP response its server sends to mobile devices. Overall, as shown in the table, the current fixes to these problems are all case by case, while our solution is consistent in the way to set origin-based security policies (whitelist of authorized origins) and enforce the policies.

Table 2 Comparison of current fixes and the fixes with Morbs

Problems	Fix w/o Morbs	Fix w. Morbs
next-intent (Section 3.2.1)	Change architecture of the Dropbox app and the Facebook app	No modification
fbconnect (Section 3.3.1)	Deprecate this feature (affecting all apps with Facebook SDKs, and taking several months)	Facebook server specifies recipient whitelist by setting a header in HTTP response "mobile- allowed-origins: app://com.facebook.katana"
Dropbox iOS SDK (Section 3.3.2)	Change both the Dropbox apps and SDKs	Dropbox SDK specifies sender whitelist by adding an entry "allowedOrigins: app://com.getdropbox.Dropbox" under "URL scheme" in .plist file.
Yelp issue (Section 3.3.2)	Unknown	No modification
callback exploit (Section 3.4.1)	Unknown	Facebook app specifies sender whitelist by calling WebViewClient:setAllowedOrigins("https://*.facebook.com")
HTTPClient exploit (Section 3.4.2)	Change to the Dropbox app, adding code for checking whether a URL is from dropbox. com when attaching authorization header	Dropbox app specifies recipient whitelist by calling HTTPClient.setAllowedOrigins(" https://*.dropbox.com").

6. RELATED WORK

Origin-based protection in web browsers. Origin-based protection is a cornerstone for browser security. All modern web browsers enforce the same-origin policy (SOP) [4] to protect the web content from one origin against unauthorized access from a different origin. Always at the center of browser security is the attacks that circumvent this protection, such as XSS, CSRF, login CSRF, and the defense that reinforces the browser and makes the protection hard to bypass [1][2][3]. Our research shows that serious cross-origin attacks can also happen on mobile platforms and therefore the origin-based protection is equally important to mobile security.

Under the SOP, cross-origin communication needs to go through designated channels with proper mediation. A prominent example is the *postMessage* channel [5], through which the web content of one origin can send messages to another domain, and the browser ensures that the recipient knows the true origin of the sender. However, the web developer of the recipient domain still needs to come up with her own policy enforcement logic, which could be error-prone. Alternatively, the browser can act on whitelisted origins specified by the developer. An example is the Cross-Origin Resource Sharing mechanism [6], through which the content from a.com can request resources from b.com server using XMLHttpRequest [7]. The server authorizes this cross-origin activity to the browser by attaching to its HTTP response a header "Access-Control-Allow-Origin: a.com", a whitelist for the requestor a.com. The browser then enforces this policy, sending the message only to a.com webpages.

The design of Morbs is pretty much in line with those browser-based security mechanisms. We bring in this origin-based protection to mobile platforms, making the true origin of each message observable to app/web developers and also helping them enforce their policies at the OS level.

Security on mobile platforms. The security framework of Android is built on i) the sandbox model [8], which separates an app's data and code execution from that of the rest of the system, and 2) the permission model [9], which grants each app different level of privileges to access system resources under the user's consent. Prior studies mainly focus on circumventing such protection to obtain private user data (e.g., GPS location, phone contacts) or perform privileged operations (e.g., sending SMS messages) without proper consents from the user [10][11][12][13][14][27]. Most related to our work here is permission re-delegation [10], in which an unprivileged app sends an intent to another app with a proper permission to act on its behalf, operating on the resources (e.g., GPS, user contacts, etc.) it is not supposed to touch. However, this problem has been studied mainly for understanding the threat to mobile devices' *local* resources. What we investigated is the protection of an app's *web* resources, which has not been explicitly included in Android's security models. Luo et al. conducted two studies specifically about security issues related to WebView: in [42], they categorized existing issues raised by other researchers and a number of issues discovered by them. Many of these issues were shown to affect Android applications that use the open-source package DroidGap; in [43], they proposed a type of attack called "touchjacking", which targets the weaknesses of WebView's handling of touch events.

To address those problems, numerous defense mechanisms have been proposed [17][18][19][20]. Particularly, information-flow techniques, such as TaintDroid [15] and Vision [16], are used to track the propagation of sensitive user data across a suspicious app at the instruction level. Different from those existing techniques, our protection mechanism is designed to keep track of the origin of the message exchanged between the initiator and the recipients for origin-based mediation. For this purpose, we only need to work on the API level (given that the OS is trusted), which is much more efficient. A related technique called Quire [21] enables Android to trace and sign the whole IPC chain observed by the OS during intent messaging, so that the recipient of an intent can find out its initiator. However, this approach is not designed to determine a request's *web* origin: for example, when an app is activated through a scheme URL generated by a malicious webpage displayed in the WebView instance of the Facebook app, looking at the IPC chain does not tell the recipient app that it is actually originated from the malicious domain.

Similar call-sequence analyses have been done on iOS to detect information leaks through iOS apps [22][23]. The focus of these analyses is on malicious apps, while our focus is on protecting benign apps.

7. CONCLUSION AND FUTURE WORK
Unlike traditional web applications, which enjoy browser-level origin-based protection, apps are hosted on mobile OSes, whose security models (e.g., sandbox and permission models) are not designed to safeguard resources based their web origins. Our research shows that in the absence of such protection, the mobile channels can be easily abused to gain unauthorized access to a user's sensitive web resources. We found 5 cross-origin issues in popular SDKs and high-profile apps such as Facebook and Dropbox, which can be exploited to steal their users' authentication credentials and other confidential information such as text input. Moreover, without the OS support for origin-based protection, not only is app development shown to be prone to such cross-origin flaws, but the developer may also have trouble fixing the flaws even after they are discovered. This points to the urgent need of building origin-based protection into mobile

platforms. In our research, we designed and implemented the first such protection mechanism, *Morbs*, for mediating cross-origin communications at the OS level. Our evaluation shows that the new technique effectively and efficiently controls the risks that come with the communications, and can also be conveniently utilized by the app and web developers.

Our current implementation is for Android. Building this new protection on iOS is equally important. Also interesting is the effort to automatically analyze existing apps, to identify their cross-origin vulnerabilities and defend them using the origin protection we provided. More generally, given the trend that web services are increasingly delivered through apps, further investigations are needed to understand how to better protect users' web resources on mobile OSes, which were originally designed to safeguard a device's local resources.

8. ACKNOWLEDGEMENTS

9. REFERENCES
[1] Fogie, S., Grossman, J., Hansen, R., Rager, A., & Petkov, P. D. (2007). *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing.

[2] Auger, R. (2008). The cross-site request forgery (csrf/xsrf) faq. *CGISecurity. com. Apr*, *17*.

[3] Barth, A., Jackson, C., & Mitchell, J. C. (2008, October). Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (pp. 75-88). ACM.

[4] Barth, A. (2011). The web origin concept.

[5] Cross-document messaging – HTML standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html#web-messaging

[6] van Kesteren, A. (2010). Cross-origin resource sharing. *W3C Working Draft WD-cors-20100727*.

[7] Garrett, J. J. (2005). Ajax: A new approach to web applications.

[8] Android Developers: Security Tips. http://developer.android.com/training/articles/security-tips.html

[9] Android Developers: Permissions. http://developer.android.com/guide/topics/security/permissions.html

[10] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., & Chin, E. (2011, August). Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium* (Vol. 18, pp. 19-31).

[11] Davi, L., Dmitrienko, A., Sadeghi, A. R., & Winandy, M. (2011). Privilege escalation attacks on android. In *Information Security* (pp. 346-360). Springer Berlin Heidelberg.

[12] Grace, M., Zhou, Y., Wang, Z., & Jiang, X. (2012, February). Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*.

[13] Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., & Wang, X. (2011, February). Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings*

of the 18th Annual Network and Distributed System Security Symposium (NDSS) (pp. 17-33).

[14] Schrittwieser, S., Frühwirt, P., Kieseberg, P., Leithner, M., Mulazzani, M., Huber, M., & Weippl, E. (2012, February). Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*.

[15] Enck, W., Gilbert, P., Chun, B. G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2010, October). TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (pp. 1-6).

[16] Gilbert, P., Chun, B. G., Cox, L. P., & Jung, J. (2011, June). Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services* (pp. 21-26). ACM.

[17] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A. R., & Shastry, B. (2012, February). Towards taming privilege-escalation attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS)* (Vol. 17, pp. 18-25).

[18] Shekhar, S., Dietz, M., & Wallach, D. S. (2012). Adsplit: Separating smartphone advertising from applications. *CoRR, abs/1202.4030*.

[19] Fragkaki, E., Bauer, L., Jia, L., & Swasey, D. (2012). Modeling and enhancing Android's permission system. In *Computer Security–ESORICS 2012* (pp. 1-18). Springer Berlin Heidelberg.

[20] Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., ... & Sadeghi, A. R. (2012, February). MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*.

[21] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., & Wallach, D. S. (2011, August). Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*.

[22] Egele, M., Kruegel, C., Kirda, E., & Vigna, G. (2011, February). PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*.

[23] Werthmann, T., Hund, R., Davi, L., Sadeghi, A. R., & Holz, T. (2013). PSiOS: Bring Your Own Privacy & Security to iOS Devices.

[24] Hardy, N. (1988). The Confused Deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, *22*(4), 36-38.

[25] Hermandroid. "Launching an Android application from a URL". http://androidsmith.com/2011/07/launching-an-android-application-from-a-url/

[26] Apple URL Scheme Reference. http://developer.apple.com/library/ios/#featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html

[27] Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011, June). Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (pp. 239-252). ACM.

[28] Android Developers: WebView.addJavaScriptInterface. http://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface%28java.lang.Object,%20java.lang.String%29

[29] Android Developers: Parcel. http://developer.android.com/reference/android/os/Parcel.html

[30] Android Developers: Java Native Interface. http://developer.android.com/training/articles/perf-jni.html

[31] Supporting materials for this work. http://research.microsoft.com/en-us/um/people/ruiwan/mobile-origin/index.html

[32] A local file loaded from SD card to webview on Android can cross-domain. http://lists.grok.org.uk/pipermail/full-disclosure/2012-February/085619.html

[33] Android-apktool – A tool for reverse engineering Android apk files. http://code.google.com/p/android-apktool/

[34] AndroChef Java Decompiler. http://www.neshkov.com/ac_decompiler.html

[35] Facebook Developers – Dialogs Overview. https://developers.facebook.com/docs/reference/dialogs/

[36] Android Developers – HttpClient. http://developer.android.com/reference/org/apache/http/client/HttpClient.html

[37] Android Developers – HttpURLConnection. http://developer.android.com/reference/java/net/HttpURLConnection.html

[38] iOS Developer Library – NSURLConnection Class Reference. http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSURLConnection_Class/Reference/Reference.html#//apple_ref/occ/cl/NSURLConnection

[39] iOS Developer Library – Making HTTP and HTTPS Requests. http://developer.apple.com/library/ios/#documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/WorkingWithHTTPAndHTTPSRequests/WorkingWithHTTPAndHTTPSRequests.html

[40] The implementation of the mobile origin-based security mechanism (Morbs) on Android is published on GitHub. https://github.com/mobile-security/Morbs

[41] PlainText – Dropbox text editing for iPhone, iPod touch, and iPad. https://itunes.apple.com/us/app/plaintext-dropbox-text-editing/id391254385?mt=8

[42] Luo, T., Hao, H., Du, W., Wang, Y., & Yin, H. (2011, December). Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference* (pp. 343-352).

[43] Luo, T., Jin, X., Ajai, A., & Du, W. Touchjacking attacks on web in android, ios, and windows phone. In *Proceedings of 5TH International Symposium on Foundations & Practice of Security (FPS 2012)*.