ReACt: A Resource-centric Access Control System for Web-app Interactions on Android

Xin Zhang
xzhang99@binghamton.edu
SUNY Binghamton
Binghamton, New York, USA

Yifan Zhang
zhangy@binghamton.edu
SUNY Binghamton
Binghamton, New York, USA

ABSTRACT

We identify and survey five mechanisms through which web content interacts with mobile apps. While useful, these web-app interaction mechanisms cause various notable security vulnerabilities on mobile apps or web content. The root cause is lack of proper access control mechanisms for web-app interactions on mobile OSes. Existing solutions usually adopt either an origin-centric design or a code-centric design, and suffer from one or several of the following limitations: coarse protection granularity, poor flexibility in terms of access control policy establishment, and incompatibility with existing apps/OSes due to the need of modifying the apps and/or the underlying OS. More importantly, none of the existing works can organically deal with all the five web-app interaction mechanisms. In this paper, we propose *ReACt*, a novel *Re*source-centric *A*ccess *C*on*t*rol design that can coherently work with all the web-app interaction mechanisms while addressing the above-mentioned limitations. We have implemented a prototype system on Android, and performed extensive evaluation on it. The evaluation results show that our system works well with existing commercial off-the-shelf Android apps and different versions of Android OS, and it can achieve the design goals with small overhead.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; Web application security; • **Human-centered computing → Mobile phones**.

KEYWORDS

Web-app interaction, access-control, Android, binder replacement, ART hooking

ACM Reference Format:
Xin Zhang and Yifan Zhang. 2021. *ReACt*: A Resource-centric Access Control System for Web-app Interactions on Android. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449960

1 INTRODUCTION

We are seeing a trend of tighter integration of web-based interfaces and mobile apps/OSes [20, 33, 44]. We identify the following five mechanisms that are commonly used for web-app interactions on mobile devices (*M1* to *M5* below, more details in §2.1): (**M1**): *JavaScript-Java bridges* expose resources owned or accessible by mobile apps to web content and thus provide a convenient way for mobile app developers to integrate their apps with their own or third-party web services [2, 4]. (**M2**): *HTML5 APIs* allow web content to access a wide range of device information and resources [39–42]. (**M3**): The `file` *URL scheme* enables direct accesses to files on device's file system from web content [9, 34, 43]. (**M4**): *Mobile deep linking*, which is an effective and popular way to enhance mobile app user experience, allows users to open specific locations within a mobile app from web pages rather than simply launching the app [1, 6, 12, 28, 38]. (**M5**): *Web event hooking* provides a number of function hooks which allow mobile apps to intercept/track user web activities, and is a useful approach of realizing user-behavior-based features in mobile apps [27].

The above web-app interaction (WAI for short) mechanisms are useful for achieving rich functionalities and good user experience. However, they also cause security vulnerabilities which allow various security attacks, such as sensitive user data leakage [9, 27, 31], code injection and cross-site scripting (XSS) [9, 23, 31], user-interface (UI) spoofing and phishing [25, 26], and web UI event sniffing and hijacking [27]. The root cause of these security vulnerabilities is the lack of proper access control (AC for short) mechanisms for WAIs in mobile OSes. Several solutions have proposed to provide AC protection for WAIs on mobile devices [13, 17, 18, 23–26, 30, 35, 36]. However, these solutions adopt either an *origin-centric* design or a *code-centric* design, both of which suffer notable drawbacks (details later in §2.2).

In this paper, we present the design and implementation of *ReACt*, a system that adopts a novel *Re*source-centric *A*ccess *C*on*t*rol design, which enables a unified way to provide fine-grained, flexible, and practical AC protection for WAI mechanisms on mobile devices.

There are *three* key aspects of our *resource-centric* design, which are summarized as follows (more details in §3).

• First, we observe that, in all the attack scenarios involving the WAI mechanisms, the attacker's goal is to obtain and/or take advantage of certain resource of the device. Here the notion of "resource" is not limited to conventional device resource, such as user data and device location, but also include unconventional ones, such as WebView instances in mechanism *M4*, and web event hooks in mechanism *M5*.

To coherently safeguard all the five WAI mechanisms, we extend the notion of "resource" to include all those targeted by WAI related attacks.

• Second, existing solutions enforce AC policies either when web content origin is determined (i.e., origin-centric) [17, 36], or when certain methods of mobile apps or web code are invoked (i.e., code-centric) [13, 18, 24, 35]. However, both of these timings either do not necessarily link to resource accesses, or cause coarse-granularity of resource protection. By contrast, *ReACt* adopts a resource-centric design, which enforces AC policies when specific resources are accessed. As a result, *ReACt* can provide definitive and fine-grained web content AC for all the five WAI mechanisms coherently.

• Third, resources related to WAI mechanisms are concerned by different participants of the mobile-web ecosystem. For example, resources exposed by the JavaScript-Java bridge mechanism can be either system resources (e.g., SMS, contact list, device location), which are concerned by the device user, or app-defined resource (e.g., home address input by the app user), which the app developer knows the best whether should be protected. Another example is that with mechanism M5 (i.e., web event hooking), web developers should make the decision whether certain apps can track the events of their web content via event hooks. Therefore, *ReACt* supports different ways of establishing resource AC policies: establishment according to user preferences, establishment by mobile app developers, and establishment by web developers.

We also find that current AC solutions for WAIs usually require rewrite/recompilation of mobile apps [13, 17, 18, 23, 24, 35, 36], or modifications to the OSes [18, 24–26, 36]. Thus, they either cannot work for existing installed apps, or require OS re-flashing which harms deployability on mobile devices that are currently in use. We aim to allow *ReACt* to function normally without the need of modifying existing mobile apps or re-flashing mobile OSes. Achieving this goal is not trivial, since *ReACt*'s resource-centric design relies on monitoring and managing device resource accesses, which are OS-level activities and thus are difficult to control without modifying apps or the OS. To address the challenge, we implemented the *ReACt* prototype system following the Boxify app virtualization approach [7], which allows the interaction between unmodified apps and the Android OS to be monitored by a trusted sandbox app. In our system, the role of such a trusted sandbox app is fulfilled by the *ReACt* manager, which is a user-level app and provides a brokered execution environment for the unmodified apps. The brokered execution environment allows the *ReACt* manager to intercept accesses to system resources (via replaced Android binders) and app-defined resources (via ART[1] hooking [16]).

The above implementation strategy also offers great flexibility in AC policy establishment. Most of the existing works require app recompilation and/or OS modification to integrate the AC policies to their system. With our solution, the *ReACt* manager reads WAI AC policies (§3.4), which are written using our *resource-centric policy language* (RCPL), from device flash and enforces them within the brokered execution environment. As a result, the AC policies can be generated/updated dynamically without any app/OS modifications. More details about our system implementation are presented in §4.

In summary, we make the following contributions in this paper:

• We systematically examine the state of the art of WAIs on mobile devices, and identify five mechanisms for such interactions and their security vulnerabilities.

• We propose a resource-centric AC design, which innovatively extend the notion of resource according to the aims of WAI-related attacks. To the best of our knowledge, our solution is the first that can coherently achieve fine-grained and flexible AC for all the five WAI mechanisms.

• We implement a prototype system which applies the proposed resource-centric AC design on Android without modifying the source code of apps and the underlying OS. As a result, our prototype system is compatible with installed apps, which is a feature few existing solutions have.

• We systematically evaluate the *ReACt* prototype system with real-world experiments. The evaluation results suggest that our system is compatible with existing Android apps and OSes, and can achieve the design goals with small runtime overhead.

[1] ART (Android Runtime) is the application runtime environment used by Android.

Table 1: The five web-app interaction mechanisms

Web-app interaction mechanism	Notation	Interaction direction
M1: **J**ava**S**cript-**J**ava **b**ridge	JJB	web ⟶ app
M2: **H**TML**5** **A**PIs	H5A	web ⟶ app
M3: **f**ile **U**RL **s**cheme	FUS	web ⟶ app
M4: **M**obile **d**eep **l**inking	MDL	web ⟶ app
M5: **W**eb **e**vent **h**ooking	WEH	app ⟶ web

Listing 1: Enabling JavaScript-Java bridge in an Android application (Java code).

```java
WebView wv = ...; // Initialize a WebView object
...
public class AppMngtWebIntf {
    ...
    @JavaScriptInterface
    public String GetDeviceIMEI() {
      // return device IMEI as a string
    }
    @JavaScriptInterface
    public void GetUsrSSN() {
      // obtain user SSN provided to the app
    }
    public void InternalMethod() {
      // JavaScript code cannot access this method
    }
}
wv.addJavaScriptInterface(new AppMngtWebIntf(),  "
     AppMngtObj");
```

Listing 2: Invoking JavaScript-Java interface in web content (JavaScript code).

```javascript
    <script>
    devIMEI = appMngtObj.GetDeviceIMEI();
    appMngtObj.GetUsrSSN();
    ...
    </script>
```

2 BACKGROUND AND RELATED WORK

2.1 Background

WebView [3, 5, 29] is a type of mobile app UI element which can be considered as a mini web browser embedded in mobile apps to display web contents. WevViews support different mechanisms for web content to interact with the content rendering apps. We introduce these WAI mechanisms and the possible attacks as follows (a notation summary of the mechanisms is given in Table 1).

(1) JavaScript-Java bridge (JJB). The JJB mechanism provides a means for web content to access resources exposed by mobile apps.

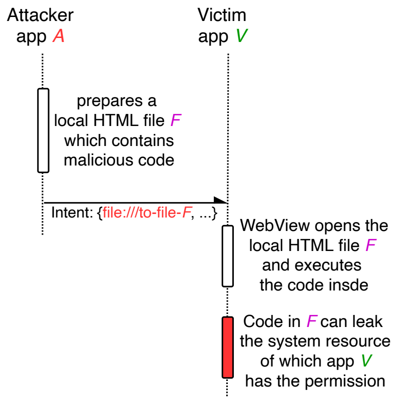
Attacker app A
Victim app V
prepares a local HTML file F which contains malicious code
Intent: {file:///to-file-F, ...}
WebView opens the local HTML file F and executes the code insde
Code in F can leak the system resource of which app V has the permission

Figure 1: Intent-based attack scenario of using file URL scheme.

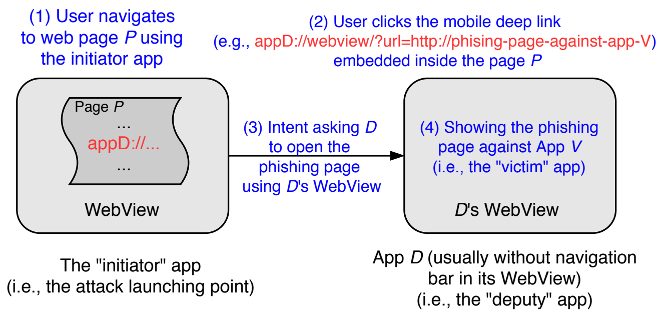
(1) User navigates to web page P using the initiator app
(2) User clicks the mobile deep link (e.g., appD://webview/?url=http://phising-page-against-app-V) embedded inside the page P
Page P
...
appD://...
...
WebView
(3) Intent asking D to open the phishing page using D's WebView
(4) Showing the phishing page against App V (i.e., the "victim" app)
D's WebView
The "initiator" app (i.e., the attack launching point)
App D (usually without navigation bar in its WebView) (i.e., the "deputy" app)

Figure 2: Mobile-deep-liking-based phishing attack scenario.

It allows app developers to register Java object in their apps' WebView instances, so that JavaScript code loaded into the WebView instances can invoke the Java object's methods within the Java objects. Listings 1 and 2 present an example in Android. Listing 1 shows the code snippet from an android app. In this example, the app contains one Android WebView instance `wv` (defined and initialized at line 1). The app developer implements a Java class named `AppMngtWebIntf` (line 3-16), where three methods are defined: the first (i.e., `GetDeviceIMEI()`) is to obtain the device's IMEI; the second (i.e., `GetUsrSSN()`) is to obtain the SSN that the user has previously input to the app; and the third (i.e., `InternalMethod()`) is a method for internal use. Here the device IMEI is a type of system resource, and user SSN is a type of app-defined resource. The developer then adds an `AppMngtWebIntf` object to the WebView instance `wv` using `WebView` class's `addJavaScriptInterface()` method (line 17). Then the `AppMngtWebIntf` object can be referenced through the object name "`appMngtObj`" by JavaScript code running inside the WebView instance. Listing 2 shows how JavaScript running in WebView can invoke the aforementioned Java object's methods to obtain device IMEI and user SSNs.

The major vulnerability here is that exported Java methods (e.g., the `GetDeviceIMEI()` and the `GetUsrSSN()` in the above example) can be accessed by any web code running in WebView, and therefore can be exploited by malicious web code to steal private user information. There have been multiple efforts in address this vulnerability [13, 17, 18, 24, 30, 35, 36]. But they suffer from different drawbacks as will be discussed later in §2.2.

(2) HTML5 API (`H5A`). `H5A` allows JavaScript in web content to call a set of APIs defined by the HTML5 specification to access system resources, such as device geolocation [32, 40], file system [39], and media stored on device [42]. Most platforms support reporting the origins of web content that makes resource access requests using `H5A` to applications. It is the app developers' responsibility to properly implement AC functionality for those requests. However, a recent study has shown that handling such AC is cumbersome and error-prone [35]. Therefore, it will be beneficial to enable an automatic AC mechanism for resource accesses initiated by `H5A`.

(3) `file` URL scheme (`FUS`). `FUS` enables web content to access device local files. Before Android 4.1 (API Level 16), JavaScript running in the context of a `file` scheme URL can access content from any origin [19], which allows attackers to load malicious JavasScript embedded in local files by including `file` scheme URLs in intents sent to apps with WebView or in local or public domain web pages. Since Android 4.1, the default values of `setAllowFileAccessFromFileURLs` and `setAllowUniversalAccessFromFileURLs` have been changed to `false` [19], which by default prevents attackers from launching attacks by using web domains. However, file access (i.e., `setAllowFileAccess`) is enabled by default with the latest version of Android, and thus attackers can still use intent to trigger victim app to open and execute malicious JavaScript existing in local files using `file` scheme URLs. Figure 1 illustrates such attack scenario. The attack would allow the attacker to obtain system resource, such as device location, which it does not have the permission to access.

(4) Mobile deep linking (`MDL`). `MDL` is a useful mechanism that allows web content to access a specific location within a mobile app other than simply launching the app [1, 12]. However, recent studies have shown that the `MDL` mechanism can be exploited to launch UI spoofing and phishing attacks [25, 26]. Figure 2 summarizes the attack process. First, a user uses the initiator app and navigates to a web page which contains an attack mobile deep link targeting the deputy app *D* (step-1). Here the initiator app is benign, and can be an app with WebView or a web browser. The User then clicks the attack link (step 2), which leads the initiator app to send an intent to the deputy app *D* asking it to open the phishing page (step-3). The deputy app is also benign, and is being exploited by the attacker to show a phishing page which impersonates the UI of another app (i.e., the victim app). Finally, the deputy app displays the phishing page (step-4). Please note that the deputy app *D* is chosen because its WebView activity does not contain any UI element (e.g., title bar) which would allow the user to be aware of the attack by noticing the victim app UI is displayed as a web page in a WebView. More details of the attack can be found in the work by Li et al. [25].

The above attack can be addressed if app *D*'s developer takes extra caution, for example, by setting the WebView of her app as private, or validating the content of incoming intents before serving them. However, such security measures may not be adopted or correctly implemented by app developers due to their inexperience. Thus, a mandatory AC mechanism is needed to protect `MDL` WAIs.

(5) Web event hooking (`WEH`). The `WEH` mechanism enables WAIs from mobile device to web content. With `WEH`, a list of web event

Table 2: Comparison of access control solutions for web-app interactions on mobile devices

	WAI mechanism targeted					Fine-grained control	Require no app modification	Require no OS modification	Dynamic AC policy establishment
	`JJB`	`H5A`	`FUS`	`MDL`	`WEH`				
Morbs [36]	√	√	√						
Jin et al. [24]	√	√				√			
NoFrak [17]	√	√						√	
PowerGate [18]	√	√				√			
Draco [35]	√	√			√	√	√	√	
WIREframe [13]	√	√				√		√	√
HybridGuard [30]	√	√				√		√	
Li et al. [25]				√			√		
Liu et al. [26]				√			√		
ReACt (this work)	√	√	√	√	√	√	√	√	√

function hooks provided by WebView are made available for Android apps. These event hooks are invoked when certain web activities happen, such as when the web page finishes loading and when the web page loads certain resource (e.g., image and video). App developers can establish their own handlers for these event hooks. Although `WEH` allows mobile apps to closely interact with web content, it also allows attackers to abuse the event hooks to launch attacks like web UI event sniffing and hijacking [27]. The cause of the problem is that there is no restriction for the hosting app to update individual event hooks. Therefore, a potential solution to the problem would be an effective AC mechanism for using the event hooks.

2.2 Related work: origin-centric & code-centric designs for WAI access control

Existing solutions on addressing the AC problems of different WAI mechanisms user either the *origin-centric* design or the *code-centric design*, which are introduced as follows.

2.2.1 ***Origin-centric design***. With this design, resource access requests from web content are granted based on origins of the content using whitelisting and/or blacklisting methods [17, 36]. Here the origin of web content refers to the URI of the content, which consists of scheme, host name, and port number [21].

The drawbacks of the origin-centric design arise from its coarse-granularity of protection. On one hand, if certain web content passed the origin filtering, it would be granted the same permissions as the hosting app, which, however, could open the door for different permission-related attacks [8, 15, 22]. On the other hand, if the web content were blocked, it would be impossible for the web content to obtain any permission-protected device resource, which could obstruct useful and benign services. For example, location-based advertising (LBA), which is an important part of mobile ecosystem [14] relies on user location information to deliver proper ad content. However, it is unlikely for LBA web content to obtain such permission-protected information with mobile web apps adopting the origin-centric design, since the origins of advertisers are different from those of the services associated with the apps, and thus are usually blocked by origin-centric policies.

2.2.2 ***Code-centric design***. To address the coarse-grained protection problem, solutions adopting the *code-centric* design [13, 18, 24, 30, 35] focus on the following two aspects: one is analyzing and adapting the code of existing mobile apps and/or web content such that proper AC policies can be applied to individual Java and/or JavaScript methods that *may* lead to device resource accesses; the other is providing a means for app/web developers to define AC policies for the methods in their newly developed code.

The code-centric design achieves finer AC granularity because it can treat requests from web content with the same origin differently. However, the code-centric approach has its own limitations.
• First, this approach may not be able to correctly prevent unwanted accesses to device/app resources, since method invocations in mobile apps may not necessarily lead to resource accesses. Therefore, code-centric AC solutions need to determine whether and what resources are eventually accessed by different methods. For instance, Draco [35], a recent code-centric AC solution on `JJB`, infers potential resource accesses from exposed Java methods by performing static code analysis, which however may not always generate the correct results. One possible solution is to involve app developers into the AC design, as they have the full knowledge of the relationship between app code methods and resource accesses. But such solution requires redevelopment of apps and thus is not compatible with already-installed apps.
• Second, the code-centric approach cannot treat different resource accesses originating from the same method invocation differently.
• Third, it is difficult to extend the code-centric approach to naturally work with other WAI mechanisms that do not generate accesses of conventional resource (such as `MDL`).

More details of the above two designs will be provided when we compare them with our resource-centric design in §3.3.

3 SOLUTION DESIGN

3.1 Adversary model

The attackers aim to exploit the vulnerabilities of the five WAI mechanisms to launch the attacks as introduced in §2.1. We make the following realistic assumptions about the attackers and the mobile apps in this work. **(1)** For the web to app interaction mechanisms (i.e., `JJB`, `H5A`, `FUS`, and `MDL`), we assume the attacker has control over web domains into which malicious web content can be inserted. **(2)** We assume malicious web content is reachable from user's apps with WebView or web browsers, for example, by careless domain control implementations in individual apps, or by web code's usage of `iframe` which can load untrusted web content. **(3)** We assume apps under *ReACt*'s protection are not malicious (in other words, these apps will not try to detect and reverse *ReACt*'s protection). **(4)** For JJB, we assume the attacker has the information of exported Java methods, which can be achieved by downloading and reverse-engineering the victim apps. **(5)** For FUS, we assume the attacker can place files with malicious JavaScript in victim device's file system. This can be achieved, for example, by providing benign-looking attack apps which can access device's file system.

3.2 Design goal

Table 2 shows the limitations of the existing solutions, which are summarized as follows. **(1)** First and most importantly, the existing solutions target a subset of the five WAI mechanisms. To the best of our knowledge, there has been no solution that can work with all the five mechanisms coherently. **(2)** The protection granularity is coarse due to the origin-centric design code-centric approaches. **(3)** Most existing solutions are not compatible with already-installed mobile apps and OSes, as they require app and/or OS modifications. **(4)** Most existing solutions require app/OS changes to integrate AC policies into their systems. As a result, they do not support generating new or changing existing AC policies during runtime.

Our goal is to address the above four limitations. The first two limitations are addressed by our resource-centric AC design which is to be introduced next, and the latter two are solved by our implementation strategy which will be presented later in §4.

3.3 *ReACt*: *Resource-centric Access Control*

3.3.1 ***Motivating observations.*** Our resource-centric design is inspired by the following two key observations. **(1)** The goal of performing AC on WAIs is to protect device resources from being accessed maliciously. Therefore, the most effective timing to perform AC is when device resource is actually accessed. **(2)** We observe that in all the WAI attack scenarios, the attacker's goal is to obtain or take advantage of certain resources, which can be conventional resources, such as system/app-defined resources, or unconventional ones, such as a "deputy" app's WebView instances and web content's event hooks. Therefore, protection from the perspective of resource accesses can be a promising way to coherently safeguard all the five WAI mechanisms,

3.3.2 ***The resource-centric design.*** Given the above two observations, the idea of *ReACt* is to monitor resource accesses and enforce AC policies when actual accesses occur at system level. Figure 3 compares the origin-centric and the code-centric designs with our resource-centric design for the JJB mechanism on Android.

Before discussing the details, a brief introduction of the process of web content accessing device resources through the JJB mechanism is necessary. The JJB mechanism on Android involves four entities: an Android app, the app's WebView instance (whose functionalities are provided by Chromium [10]), the Android OS, and the hardware. When the WebView instance loads the web content (i.e., the step-1 in Figure 3), the renderer thread of it parses the web content (step-2). Upon seeing Java object method invocations when parsing, the

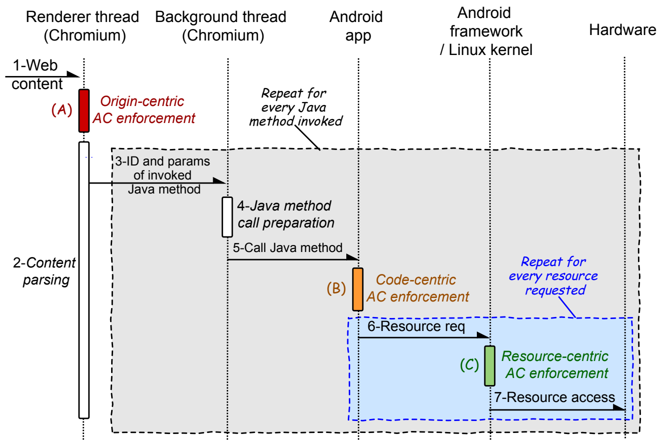
Renderer thread (Chromium)
Background thread (Chromium)
Android app
Android framework / Linux kernel
Hardware
1-Web content
(A)
Origin-centric AC enforcement
Repeat for every Java method invoked
3-ID and params of invoked Java method
4-Java method call preparation
2-Content parsing
5-Call Java method
(B)
Code-centric AC enforcement
Repeat for every resource requested
6-Resource req
(C)
Resource-centric AC enforcement
7-Resource access

Figure 3: Comparison of the *origin-centric*, the *code-centric*, and the *resource-centric* access control approaches for the Java-JavaScript bridge mechanism in Android.

renderer thread passes the IDs[2] and the parameters of the invoked methods to the background thread of the WebView instance[3] (step-3). Then the background thread performs preparation work for calling the Java object's method (step-4), and invokes it (step-5). The Java object's method eventually makes request(s) to access certain system or app-defined resource (steps 6 & 7).

With the *origin-centric* design, AC policies, which look like "*Web requests from origin X are allowed (or not allowed)*", are enforced right after the renderer thread parses and obtains the origin of the web content. Therefore, the AC is applied to all the resources requests from the same origin regardless user's preference.

With the *code-centric* design, better protection granularity can be achieved because its AC policies, which look like "*Web requests from origin X are allowed (or not allowed) to access resource Y*", are enforced based on the invoked Java methods. However, the connection between a method invocation and resource access need to be determined by rewriting the app code (which is incompatible with installed apps) or by inference (whose correctness is not guaranteed). In addition, if different resources are accessed in one method, they cannot be treated differently (i.e., the AC granularity is still coarse).

With our *resource-centric* design, AC policies, which look like "*Accesses to resource X can (or should not) be granted if the resource consumer is Y*", are enforced when specific resource is accessed at system level so that we can treat individual resource accesses differently even if they come from the same origin or the same method. Here the notions of "resource" and its "consumer" differ for different WAI mechanisms, and are summarized in Table 3. More details of "resource" and "consumer" are presented next in §3.4.

3.4 The resource-centric policy language (RCPL)

We define RCPL, a declarative resource-centric policy language which is used to establish resource AC policies in our system. The

[2]Chromium assigns IDs for methods of registered Java objects, and uses these IDs to identify and reference those methods internally.
[3]Chromium uses the background thread to enable concurrent JavaScript compilation. Interaction between JavaScript and Java objects also takes place in this thread.

Table 3: "Resource" and its "consumer" in *ReACt*'s AC policies.

	Resource	Resource consumer
JJB	Resources accessible from an app $\mathcal{A}$'s exposed methods	Web domains which make use of app $\mathcal{A}$'s exposed methods
H5A	Resources accessible from the HTML5 APIs	Web domains which host the HTML5 code
FUS	Resources accessible by an app $\mathcal{A}^1$	Sender app^2 of the intents which request $\mathcal{A}$ to open a `file` scheme URL
MDL	An app $\mathcal{A}$'s^3 WebView instances(s)	Domains of the web content to be displayed^4 in $\mathcal{A}$'s WebViews
WEH	Web content's event hook(s)	An app $\mathcal{A}$ which displays the web content

1: In the context of Figure 1, the victim app V is the app $\mathcal{A}$ here.
2: In the context of Figure 1, the attacker app A is the sender app here.
3: In the context of Figure 2, the deputy app D is the app $\mathcal{A}$ here.
4: In the context of Figure 2, the phishing page is the web content to be displayed here.

syntax of RCPL is described using the Backus-Naur Form (BNF) notation [37]. An RCPL rule is defined as:

<RCPL rule> ::= <resource> "-" <consumer> "-" <permission>

3.4.1 ***Resource definition.*** The "resource" in the above rule represents the resource being protected by the rule, which can be (1) device system resource, (2) app-defined resource, (3) app WebView instances, or (4) web event hooks. Thus the rule of "resource" is defined as:

<resource> ::= <system resource> | <app-defined resource> |
<webview resource> | <EH resource>

System resource protected by *ReACt* is similar to those guarded by Android's permission system. Currently our prototype system supports four types of resource: device's location, phone number, contact list, and calendar. The rule of system resource is defined as follows (more can be added when the corresponding implementation is added to the system).

<system resource> ::= "gps" | "phone_number" | "contacts" |
"calendar" | ...

Among the five WAI mechanisms, JJB, H5A, and FUS are able to expose system resource to their users.

App-defined resources are those created by individual apps. For example, app $\mathcal{A}$ collects user home addresses and exposes them to the web via the JJB mechanism. In this case, the user home addresses collected are a type of resource defined by $\mathcal{A}$. Since app-defined resource can only be obtained through exported Java methods in JJB, they are represented by the corresponding Java methods in RCPL's syntax:

<app-defined resource> ::= <exported method> | <exported
method> "," <app-defined resource>
<exported method> ::= <package name> : <class name> ":" <method
name>
<package name> ::= <token> | <token>"."<package name>
<token> ::= "*" | *string*
<class name> ::= *string*
<method name> ::= *string*"()"

In the above syntax, "package name" refers to the package name of the app which defines the resource.

WebView resource is targeted in MDL-related attacks. For example, the WebView instance of the "deputy" app (i..e, the app D in Figure 2) is exploited by attackers to display spoofed web content which mimic the UI of a victim app. Since WebView instances and their hosting apps are uniquely related, we use package name of the app to represent WebView instances(s) in RCPL's syntax. We support the asterisk wildcard character (*) in the representation of package names, such that an RCPL policy rule can be used to describe the WebView resource of a group of apps. The rule of webview resource is defined as follows.

<webview resource> ::= <wv> | <wv> "," <webview resource>
<wv> ::= "webview@"<package name>

Web event hook (EH) resource is targeted in WEH related attacks. Specifically, an attack app can exploit the web event hooks to track user activity or hijack events when web pages of a targeted web domain are being displayed in the app's WebView. Therefore, in RCPL's syntax, web event hook resource is represented by the combination of web domain and a specific web event function hook. The asterisk wildcard character is used in the syntax to support protecting a range of web content with a single rule. The rule of EH resource is defined as follows.

<EH resource> ::= <web domain> "+" <event hook>
<web domain> ::= "web:"<domain>
<domain> ::= <subdomain>"."<domain> | *string*
<subdomain> = "*" | *string*
<event hook> = "*" | "onPageStarted" | "onPageFinished" |
"onReceivedLoginRequest" | "onLoadResource" | ...

The definition of web domain in the above syntax can be replaced by URL to support finer control granularity for web content. There are 20 web event function hooks supported in Android's WebViewClient class, four of which are shown above, as of Android API level 24.

3.4.2 ***Consumer definition.*** The "consumer" in an RCPL rule represents the entity that can potentially make access request to the "resource" specified in the rule. As summarized in Table 3, there are two types of consumer: *web consumer* and *app consumer*. Web consumer is represented by the web domain name (defined above), and app consumer is represented by app name in RCPL's syntax.

<consumer> = <web domain> | <app name>
<app name> ::= "app:"<package name>

To express the notion of "all but" for consumer representation, we use the notation "!{*subject*}" to represent "all the consumers except the one enclosed".

3.4.3 ***Permission definition.*** The "permission" in an RCPL rule specifies what type of access requests the mentioned "consumer" can be made to the mentioned "resource" in the same rule.

<permission> = "read" | "write" | "read_write" | "call" | "none"

The "read", "write" and "read_write" permissions above are applied to system resources, and the "call" permission is applied to app-defined, WebView, and EH resources.

3.5 Establishing AC policies and dealing with resource accesses in WAIs

ReACt AC policy establishment is performed by users, app developers, or web developers, depending on the type of the resource. In the following, we elaborate the reasons, and describe how AC policies are established for each of the four types of resource.

3.5.1 ***AC policy establishment for system resource.*** AC policies for system resource are established based on **user preferences**.

The reason is two-fold. *First*, mobile device users are best suited to decide if certain system resources, such as device location and contact list, should be accessed by an external web domain or an app on their devices. *Second*, this approach is in accordance with existing Android usage experience that system resource is granted to individual apps based on user's choice during runtime. With *ReACt*, when certain system resource is about to be accessed, such an access attempt is intercepted and the resource consumer is identified. If the existing AC policies do not contain information about how to handle the access, the user is prompted whether the resource should be granted to the consumer. Based on the user's feedback, the *ReACt* runtime system automatically generates an AC policy, which applies to the pending access attempt, as well as the same type of resource requests in the future.

Recall that among the five WAI mechanisms, system resource can be exposed by `JJB`, `H5A`, and `FUS`. Next, we make three examples to illustrate how accesses to system resources, which are exposed by the three mechanisms respectively, are dealt with in *ReACt*.

Example 1: system resource exposed by `JJB`. Imagine that a web-based group-study app *GS* exports a Java method, which accesses app user's calendar, to JavaScript interface using the `JJB` mechanism. When the user uses the app *GS* to visit some web content on the app's corresponding web domain (i.e., `studytogether.com`), the web content requests to read/write user's calendar. Then the *ReACt* system intercepts such a request, and asks the user whether to grant such access to the said web content. On positive answer, the following AC rule is automatically generated and applied to the current and future requests.

```
calendar − web:studytogether.com − read_write
```

Example 2: system resource exposed by `H5A`. Continuing the example 1, while browsing the web domain `studytogether.com` in the app *GS*, the user accidentally clicks an ad link, which leads to another web domain `olinegamead.net`. This web domain calls the HTML5 API to request the user device's GPS location. A properly developed *GS* app would handle the HTML5 API call and ask the user if the location request should be granted. However, the *GS* app may not implement or illy implemented such an AC logic (recall that handling similar AC is cumbersome and error-prone [35]). With *ReACt*, the request to access GPS location is automatically intercepted, and the user's consent is asked for the request. Suppose that the user does not want to reveal her location to the web domain `onlinegamead.net`, the following AC rule is generated and applied.

```
gps − web:onlinegamead.net − none
```

Example 3: system resource exposed by `FUS`. Another example is that a simple calculator app sends an intent to another app *V* to obtain, via the `FUS` mechanism, device location to which app *V* has the permission. The *ReACt* system is able to intercept the location access attempt and identify the source of the attempt (which is the calculator app), and prompts the user about it. If the user chooses to deny, the following AC rule is generated and applied.

```
gps − app:com.simple.calculator − none
```

3.5.2 *AC policy establishment for app-defined resource.*

AC policies for app-defined resource is established by **app developers**, because app developers know the best about whether and how these resources should be protected. These AC policies are stored in regular files which are read by *ReACt* runtime to be populated into *ReACt*'s AC policy DB (details later in §4).

Example 4: app-defined resource exposed by `JJB`. In the example shown in Listing 1 previously, user SSN number is an app-defined resource and the method `GetUsrSSN()`) is defined to allow external web code obtain such an resource. If the app developer wants only her web domain (i.e., `mydomain.com`) can obtain user SSN by calling the method, she can define the AC rules in the listing below.

```
Listing.One.App:AppMngtWebIntf:GetUsrSSN() − web:
      mydomain.com − call
Listing.One.App:AppMngtWebIntf:GetUsrSSN() − !{web:
      mydomain.com} − none
```

3.5.3 *AC policy establishment for WebView resource.*

AC policies for WebView resource is also established by **app developers**. This is because the app developers have the best knowledge about who should (or should not) be using their apps' WebView instances for displaying web contents.

Example 5: WebView resource exposed by `MDL`. In the example shown in Figure 2 (which illustrates the `MDL`-related attack), to prevent her app from being exploited to display unwanted web content, the app developer can define an AC rule in the below listing for her app's WebView such that the WebView only loads content from her own web domain (i.e., `mydomain.com`).

```
webview@Figure.Two.Deputy.App − web:mydomain.com −
      call
webview@Figure.Two.Deputy.App − !{web:mydomain.com} −
      none
```

3.5.4 *AC policy establishment for EH resource.*

AC policies for web event hook (EH) resource is established by **web developers**, because web developers have the knowledge of which apps can or should not track events when displaying their content. Similar to AC policies established by app developers, these AC policies are also stored in regular files and are populated to *ReACt*'s AC policy DB during runtime.

Example 6: EH resource exposed by `WEH`. Suppose a web developer develops web content for `mydomain.com`. She can define the two rules shown in the listing below if she wants only the app with package name `My.Domain.App` can track the events happening while the app is displaying the web content.

```
web:mydomain.com + ∗ − app:My.Domain.App − call
web:mydomain.com + ∗ − !{app:My.Domain.App} − none
```

4 SYSTEM IMPLEMENTATION

We have implemented a prototype *ReACt* system on Android AOSP 7.0.1. Although the implementation is based on Android, the resource-centric idea to coherently address the AC problem for different WAI mechanisms is applicable to other mobile OSes.

There are two main goals for our *ReACt* system implementation. *First*, the prototype system should be compatible with installed apps

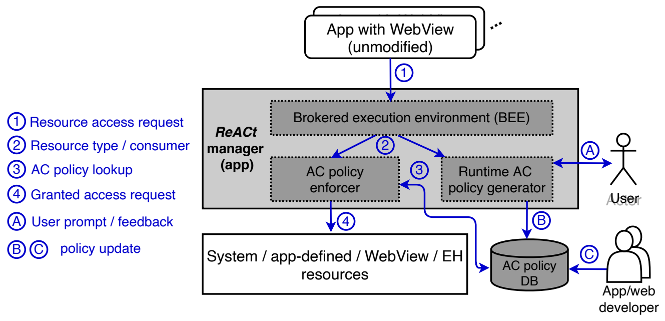
App with WebView (unmodified)
① Resource access request
② Resource type / consumer
③ AC policy lookup
④ Granted access request
Ⓐ User prompt / feedback
Ⓑ Ⓒ policy update
ReACt manager (app)
Brokered execution environment (BEE)
AC policy enforcer
Runtime AC policy generator
User
System / app-defined / WebView / EH resources
AC policy DB
App/web developer

Figure 4: ***ReACt*** **runtime system overview.**

and existing Android OS, meaning it should not require modification, compilation or installation of apps and the Android OS. *Second*, the implementation should correctly reflect *ReACt*'s resource-centric design discussed previously. The following discussion of the system implementation is centered around how we achieved these two goals.

4.1 Overview of the *ReACt* runtime system

The core component of our system is the *ReACt* manager, a regular Android app which does the following operations. **(1)** *ReACt* manager provides a brokered execution environment (BEE) for running unmodified android apps. Here "brokered" means all interactions between the unmodified apps and the Android system, such as system service requests and resource requests, are routed to the BEE such that the *ReACt* manager can execute the resource AC control design described previously. **(2)** *ReACt* manager performs the resource AC based on resource type/consumer and AC policies. **(3)** *ReACt* manager generates AC policies based on user feedback during runtime. Figure 4 shows an overview of our *ReACt* prototype system. In the system, unmodified apps which are under *ReACt* system's monitoring interact with the four types of resource (i.e., system resource, app-defined resource, app WebView instances, web event hooks (EH)) through the BEE of which the details are deferred later (step ① in Figure 4). For each resource access request, the BEE identifies the type of resource being requested and its consumer, which are passed to the AC policy enforcer and the runtime AC policy generator (step ②). In case that the resource being requested is system resource and there is no AC policy regulating whether the consumer can access the resource, the AC policy generator asks for user's preference (step Ⓐ), generates a new policy accordingly, and adds it to the AC policy DB (step Ⓑ). The above two steps are not needed if the resource being requested is not system resource because the AC policies for non-system resource are provided by app/web developers (step Ⓒ). The AC policy enforcer consults the AC policy DB to determine whether the request should be granted (③). On a positive decision, the *ReACt* manager requests the resource and returns it to the unmodified app (step ④).

We highlight three aspects in the above design. *First*, the BEE provided by the *ReACt* manager is able to launch unmodified apps and redirect all the resource access requests through it. *Second*, the implementation of the BEE does not require modifying Android framework or the Linux kernel. *Third*, the AC policy DB works as an external DB to both the *ReACt* manager and the Android OS. Therefore, AC policies in our system can be updated dynamically.

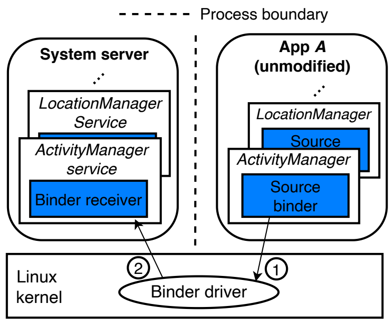

Figure 5: App-to-system-service comm. via binder in Android.

4.2 The brokered execution environment (BEE)

The main goal of the BEE is to intercept resource access requests for the four types of resource made by the unmodified apps which are under *ReACt* system's monitoring. We adopted a strategy similar to the Boxify app virtualization approach [7] in enabling the BEE: redirecting the communications between the protected apps and the Android system, such as system resource access and intent passing, through an environment which is under our control, such that those communications can be monitored as needed. The key technique to realize this strategy is Android binder replacement, which is described next.

4.2.1 ***Android binder replacement.*** The system resources concerned in this paper are provided by different Android system services. When an app is started, Android system creates *local instances* of different system services such that the app can request certain system service through the corresponding local instance of the service. Most system services are running as individual threads with the System Server process. The mechanism that connects an app's local instances of system services and their counterparts in the System Server process is binder, which is a form of inter-process communication (IPC) mechanism in Android. Figure 5 shows an example of how an app *A* communicates with the `ActivityManager` Service using binder. The app *A*'s local instance of `ActivityManager` has a "source binder" through which the app can send intents (i.e., a form of message in Android) to the `ActivityManager` Service which receives intents from apps through a "binder receiver". The communication from the source binder to the binder receiver is facilitated by the binder driver located in the Linux kernel.

To allow the *ReACt* manager to intercept system resource access requests from mobile apps, the binder replacement technique replaces the source binders in the local instances of system services with different ones, such that the replaced source binder would send requests to the *ReACt* manager app instead of the actual system services. Figure 6 demonstrate an example of the above process: the source binder in app *A*'s `ActivityManager` local instance is replaced with one which would communicate with the binder receiver for the `ActivityManager` within the *ReACt* manager app. As a result, all `ActivityManager`-related resource requests sent from app *A* will be intercepted by the *ReACt* manager, which grant the requests only if they pass the AC policies.

In our implementation, the replacement of source binder does not require modifications to the app because we use Java reflection, which is a way of examining or modifying the behavior of Java methods, classes, interfaces at runtime without the source code, to

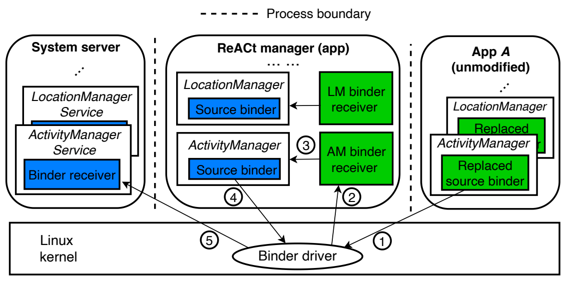

Figure 6: *ReACt* manager app intercepts binder-based app-to-system-service communication using binder replacement.

replace the original source binders with ours. To obtain the original source binder objects, our operations are different based on whether a local static binder object is available. For system services which use static references for source binders in local instances, such as `ActivityManager` service and `NotificationManager` service, we use Java reflection to directly replace the source binders. For the remaining services, such as `LocationManager` which don't have such local static binder reference, we obtain the reference to the source binders by calling `ServiceManager.getService()` to get the corresponding binder interface, which is stored in a static hashmap called `sCache` inside the `ServiceManager` local instance, while applying Java reflection.

The Android binder replacement approach is useful to *ReACt* runtime system not only because it helps to redirect system resource requests to *ReACt* manager without changing app/OS, but also because it is able to intercept intent sending (by replacing binders in `ActivityManager`) such that we can identify resource consumers in `FUS` and `MDL` mechanisms (details later).

4.2.2 ***ART hooking****.* Android Runtime(ART) is the runtime used by apps in Android 5.0 and later. The ART hooking technique gets Java method's native representation inside ART and modify ART's internals to change a Java method's behaviour.

We used ART hooking to intercept app-defined resources, which are represented by the corresponding `get`/`set` Java methods, and web event hooks resources, which are essentially Java methods. Due to the space limit, we skip the implementation details regarding ART hooking, which are largely introduced in work [11].

4.2.3 ***Setting up the BEE (i.e., ReACt manager launching unmodified apps)****.* We set up the BEE and launch unmodified apps in the following way. All apps under monitoring are launched through the *ReACt* manager. The *ReACt* manager carries the code of resource access monitoring and AC enforcement, which is to be injected into the runtime environment of the unmodified apps using either Android binder replacement or ART hooking. To properly launch an unmodified app *A* with the *ReACt* manager, the app is started, bound to and running in an empty Activity of *ReACt* manager. We redirect app `A`'s `ActivityManager`-related requests to the *ReACt* manager using binder replacement as introduced previously. The *ReACt* manager manages all app `A`'s `ActivityManager`-related requests on behalf of the app. The code of the new functionalities (i.e., code for resource access monitoring and AC enforcement) is carried in the `Application` class of the *ReACt* manager. The new empty Activity, in which the app `A` is running, shares the same `Application` class with the *ReACt* manager. As a result, we use process IDs inside the `Application` class to determine the correct running context (i.e., the *ReACt* manager's context or the app `A`'s context). Once the app `A` is started, we use the binder replacement approach to monitor app `A`'s other system service requests using binder replacement, and use the ART hooking approach to intercept invocations of app-defined resource and web event hooks.

4.3 Implementing the resource-centric design

The resource AC workflow for each of the five WAI mechanisms consists of two important steps: *resource type identification* and *resource consumer identification*, which we described as follows.

4.3.1 ***Resource type identification****.* For *system resource*, we can identify the specific type using the binder replacement approach introduced previously (because all system resource accesses are done through certain Android System Services). Our current implementation supports four types of system resource: device location, phone number, contact list, and calendar. Device location is obtained through the `LocationManager` service. Phone number is obtained through the `TelephonyManager` service. Contact list and calendar are obtained from the corresponding Content Providers each of which are associated with their own Activity, and Activity is managed through the `ActivityManager` service. Therefore, we can know what specific type of resource is being accessed by checking which binder receiver was used when receiving intents in the *ReACt* manager.

For *app-defined resource*, which is represented by the `get`/`set` methods defined by app developers, we can know what specific resource is accessed by looking at what customized Java method, which was hooked on using the ART hooking technique.

For *WebView resource*, it is used in `MDL` related attacks where attackers request certain apps to use their WebView instances to display attack contents by embedding mobile deep links in intents. Therefore, we can know if certain apps' WebView instances are about to be exploited by monitoring intent communication, which is achieved by replacing `ActivityManager`-related binders.

For *web event hook resource*, similar to app-defined resource, we can know what specific web event hooks, which are essentially Java methods, are being utilized by checking which hook method is being invoked.

4.3.2 ***Resource consumer identification****.* There are two types of resource consumers: *web consumer* and *app consumer*. The identity of web consumer, which is represented by the origin of web content, is obtained by instrumenting the Android system WebView app, which can be changed, compiled and installed like regular Android apps. The Android system WebView app provides the web engine support for WebView instances in different apps. By instrumenting the Android system WebView app, we can identify the origins of different web users. We adopt two approaches of instrumenting the Android system WebView app to identify web consumers. **(1)** *The thread ID based approach.* This approach is based on the observation that resource accesses with the `JJB` mechanism takes place in Chromium's background thread, in which no other background resource access would happen. Therefore, we use the ID of the thread in which a resource access happens to decide if the access is caused by a JJB WAI. **(2)** *The mark-and-check-later approach.* This approach is suitable for a WAI mechanism if resource accessed needs to be returned via the interaction initiating point. For example, in an H5A WAI, resources accessed by the HTML5 APIs will be returned through Chromium's renderer thread which is also the starting point of the WAI. Therefore, we can mark the access with its resource type in the access-monitoring phase, and enforce the AC policies when the resource is returned to the renderer thread where the resource consumer is identified.

The identify of app consumer is relatively straightforward: it can be obtained through intent passing monitoring as described previously.

5 DISCUSSION AND LIMITATIONS

Comparison to Android permission system. Similar to Android permission system, *ReACt* provides AC for system resources. It is noteworthy that the difference between our resource AC and the AC by Android's permission system is that Android permissions deal with system resource requests from individual apps, whereas in our case the consumers of resources are web domains or apps that are *different* from the apps making the access request. Also, Android permissions cannot protect other forms of resource, such as app-defined resource, WebView resource, and EH resource.

The brokered execution environment (BEE) and the *ReACt* manager. BEE allows for monitoring and interception of communications between the protected apps and the Android system without modifying apps or the OS. In our prototype system, the BEE is provided by the *ReACt* manager app. It is worth noting that although (unmodified) protected apps are launched by the *ReACt* manager, they are not running inside the *ReACt* manager's context. Instead, individual protected apps run in their own processes, whose communication with the Android system are redirected through the *ReACt* manager app. Therefore, the process-level app isolation in Android is retained with our prototype system. However, we do need to trust the *ReACt* manager app which oversees all the resource accesses related to the protected apps.

Limitations. The advantage of requiring no app/OS modification by our prototype system is at the cost of the following limitations. *First*, since intent passing that involves protected apps goes through the *ReACt* manager, communication between non-protected apps (which are not launched by the *ReACt* manager) and the protected apps is not possible. *Second*, normal Android user experience is broken with our prototype system because all the protected apps need to be launched via the *ReACt* manager rather than the Android launcher. *Third*, the binder replacement approach introduces overhead (albeit small, details next). To address these limitations, we could implement the proposed resource-centric design as part of the Android OS.

6 SYSTEM EVALUATION

System Effectiveness. To evaluate the effectiveness of *ReACt* system, we use five commercial apps which expose JJB interfaces: WPS, CVS, Twitter, Walmart and TikTok. We developed web code to take advantage of the interfaces and check if our *ReACt* system can capture the wanted accesses. we also developed several apps each of which takes advantage of an individual WAI mechanism to acquire private user and device information, and ran them with our *ReACt* prototype system. The experiment results show that our *ReACt* system can accurately detect resource accesses which violated the AC policies in all the cases.

Table 4: Web content loading time overhead

WAI mechanism	Overhead in milliseconds	Overhead in pert.
JJB	2.8	6.2%
H5A	4.6	7.2%
FUS	18.3	6.7%
MDL	8.4	3.1%

Meanwhile, since our solution relies on intercepting and examining system level operations, it is expected to introduce some amount of overhead. Therefore, in the following, we focus on the overhead caused by the *ReACt* system. All the experiments were carried out using a Google Nexus 5 smartphone running the Android AOSP 7.0.1 OS.

Web content loading time. We first evaluated the overhead for web content loading time. We developed our own test app which employed the WAI mechanisms so that we can instrument the app to accurately measure the web page loading performance with and without *ReACt*. Specifically, our test app uses WebView to load web pages, and it also has Activities for resolving incoming intents. The test app has all the necessary permissions of the device's resources to trigger each WAI mechanism. In addition to the test app, we also developed different web code for triggering different WAI mechanisms:
– For JJB, we implemented a Java method which requests the device's geolocation, and then expose this method through the JJB mechanism.
– For H5A, the web code calls the HTML5 Geolocation APIs to get the devices location information.
– For FUS, we used another app to send an intent containing a `file` scheme URL, which points to an HTML file on the device's local storage, to the test app, which parses the intent and loads the local HTML file specified in the intent. This local HTML file contains the code to read a private file of the test app to simulate the attack which exploits the FUS mechanism.
– For MDL, we also used another app to send an intent to the test app. The intent contains a deep link which is to request one of the test app's WebView instances to load a web page.

The web content loading time for different WAI mechanisms were measured as follows. For JJB and H5A, the loading time was calculated as the time between the occurrence of WebView event `onPageStarted()` and the event `onPageFinished()`. For FUS and MDL, the loading time was the time between the invocation of `startActivity()` to the intent and the occurrence of the Webview event `onPageFinished()`.

In each experiment, to eliminate the impact of network transmission time fluctuation, the web code is located in a local HTML file instead. Each of the page loading experiments was repeated 200 times, and the average values are reported. Table 4 shows the web content loading overhead in both milliseconds and percentage for each mechanism. The percentage value is calculated as $\frac{(T_w - T_{wo}) \times 100}{T_{wo}}\%$, where T_w and T_{wo} are the loading times with and without *ReACt* enabled respectively. From the result we can see that,

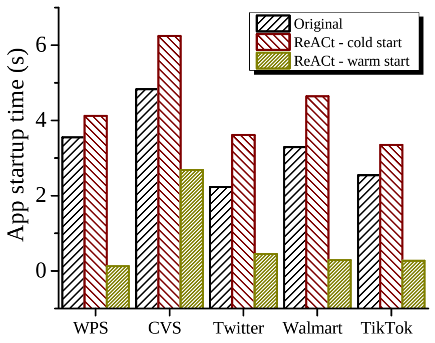

Figure 7: Application startup time

in the cases of JJB, H5A, FUS, and MDL, *ReACt* only introduced less than 20 ms of additional time to load the web content, which was unnoticeable. Considering that WAI is usually not a high frequency event, the loading time overhead is essentially negligible.

Mobile app startup time. With *ReACt*, protected apps are started through the *ReACt* manager app instead of the Android launcher. Therefore, we also evaluated how our system would introduce overhead to the app startup process. In this experiment, we selected WPS, CVS, Twitter, Walmart and Tiktok for our evaluation. Each of these apps is within the most popular apps, which make use of WebView, from their own categories (i.e., business, health, social, shopping and photography) in Google Play. We define app startup time as the time from the user click the icon from the Android Launcher or from our *ReACt* manager to the time that the app is started. For *ReACt*, we also define the startup of a protected app $\mathcal{P}$ as "cold start" if it is the first time that $\mathcal{P}$ is loaded by the *ReACt* manager app since the manager was launched; otherwise the startup of $\mathcal{P}$ is defined as "warm start". We repeated this experiment for different apps for 20 times and report the average value here.

Figure 7 shows the experiment result. From the result we can see that *ReACt* caused around half a second to one second more time, which is an acceptable amount of overhead, to start each of the five apps if it was the first time for the *ReACt* manager to start the app (i.e. cold start). Interestingly, if a protected app $\mathcal{P}$ was started by the *ReACt* manager before, it would take much less time for the *ReACt* manager to start $\mathcal{P}$again (i.e., warm start). For all the cases the warm start time with *ReACt* is even smaller than that of using the original Android launcher. The reason is that with our implementation, the *ReACt* manager still keeps some data structure for a protected app after the app is closed. For example, each protected app is running in a child process of the *ReACt* manager. Our current implementation does not terminate the hosting process after a protected app exists. Therefore, when the same app is started next time, some time can be saved due to reuse of the process. Given the above observations, we are currently planning an optimization on *ReACt* manager, which is to flush out cached data structures only when an app is not frequently used. This way, we will be able to strike a good balance between app startup time and memory usage.

Overhead on normal resource accesses. Since our solution needs to impose on sensitive resource accesses, some overhead is expected to caused for normal resource accesses, such as those triggered by explicit user operations. To quantify this overhead, we performed experiments to measure the invocation times of Android framework APIs, such as `getLastKnownLocation()` of the `LocationManager` class and `getAccounts()` of the `AccountManager` class, with and without *ReACt* enabled. The result shows us that *ReACt* usually caused 10% to 25% of overhead depending on the complexity of the APIs (APIs requesting more system services suffer a higher percentage of overhead). Considering the invocations of Android APIs are usually very fast (e.g., less than 100 ms), the overhead caused by *ReACt* is negligible.

7 CONCLUSION

We comprehensively examined five WAI mechanisms on Android. We presented *ReACt*, a resource AC system for such mechanisms. *ReACt* adopts a novel resource-centric design such that it can provide fine-grained resource AC protection for all the five WAI mechanisms in a unified way, which is not possible with the existing solutions. We implemented a prototype *ReACt* system and evaluated it with real-world experiments. The evaluation results show that our system achieves the design goals with little overhead.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendously valuable feedback. This work was supported in part by NSF Award #1566375.

REFERENCES

[1] Alexandre Jubien. Use deep linking as a powerful acquisition and retention channel. https://www.apptamin.com/blog/mobile-deep-linking/.

[2] Andrew Lee-Thorp. Android WebViews and the JavaScript to Java Bridge. https://www.synopsys.com/blogs/software-security/android-webviews-and-javascript-to-java-bridge/.

[3] Android Developers. Building Web Apps in WebView. https://developer.android.com/guide/webapps/webview.html.

[4] Android Guru. Binding JavaScript and Android Code: an Example. http://programmerguru.com/android-tutorial/binding-javascript-and-android-code-example/.

[5] Apple Developer Documentation. UIWebView - UIKit. https://developer.apple.com/documentation/uikit/uiwebview.

[6] Azim, T., Riva, O., and Nath, S. ulink: Enabling user-defined deep linking to app content. In *ACM MobiSys* (2016).

[7] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security Symposium* (2015).

[8] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *ACM MobiSys* (2011).

[9] Chin, E., and Wagner, D. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications (WISA)* (2013).

[10] chromium.org. The Chromium Projects. http://www.chromium.org/Home.

[11] Costamagna, V., and Zheng, C. Artdroid: A virtual-method hooking framework on android ART runtime. In *International Workshop on Innovations in Mobile Privacy and Security (IMPS)* (2016).

[12] Dan Kosir. 7 Key Benefits of Mobile App Deep Linking. https://clearbridgemobile.com/7-benefits-of-mobile-app-deep-linking/.

[13] Davidson, D., Chen, Y., George, F., Lu, L., and Jha, S. Secure integration of web content and applications on commodity mobile operating systems. In *ACM AsiaCCS* (2017).

[14] Dhar, S., and Varshney, U. Challenges and business models for mobile location-based services and advertising. *Communications of the ACM 54*, 5 (2011), 121–128.

[15] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium* (2011).

[16] Frumusanu, A. A Closer Look at Android RunTime (ART) in Android L. *AnandTech: http://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/* (2014).

[17] Georgiev, M., Jana, S., and Shmatikov, V. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS* (2014).

[18] Georgiev, M., Jana, S., and Shmatikov, V. Rethinking security of web-based system applications. In *WWW* (2015).

[19] Google Developers. Android WebSettings. https://developer.android.com/reference/android/webkit/WebSettings.

[20] Greg Sterling. Nielsen: More Time On Internet Through Smartphones Than PCs. http://marketingland.com/nielsen-time-accessing-internet-smartphones-pcs-73683.

[21] Internet Engineering Task Force (IETF). RFC6454: The Web Origin Concept. https://tools.ietf.org/html/rfc6454.

[22] Jiang, X., and Zhou, Y. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy* (2012).

[23] Jin, X., Hu, X., Ying, K., Du, W., Yin, H., and Peri, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *ACM CCS* (2014).

[24] Jin, X., Wang, L., Luo, T., and Du, W. Fine-grained access control for html5-based mobile applications in android. In *Information Security Conference (ISC)* (2013).

[25] Li, T., Wang, X., Zha, M., Chen, K., Wang, X., Xing, L., Bai, X., Zhang, N., and Han, X. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *ACM CCS* (2017).

[26] Liu, F., Wang, C., Pico, A., Yao, D., and Wang, G. Measuring the insecurity of mobile deep links of android. In *USENIX Security Symposium* (2017).

[27] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on webview in the android system. In *ACM ACSAC* (2011).

[28] Ma, Y., Liu, X., Hu, Z., Liu, Y., and Xie, T. Aladdin: Automating release of deep-link api on android. In *WWW* (2018).

[29] Microsoft Docs. Web view - UWP app developer. https://docs.microsoft.com/en-us/windows/uwp/controls-and-patterns/web-view.

[30] Phung, P. H., Mohanty, A., Rachapalli, R., and Sridhar, M. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. In *IEEE Security and Privacy Workshops - Mobile Security Technologies (MoST)* (2017).

[31] Rizzo, C., Cavallaro, L., and Kinder, J. Babelview: Evaluating the impact of code injection attacks in mobile webviews. *arXiv preprint arXiv:1709.05690* (2017).

[32] Ruadhan O'Donoghue. HTML5 for the Mobile Web - a guide to the Geolocation API. https://mobiforge.com/design-development/html5-mobile-web-a-guide-geolocation-api.

[33] The Guardian. Smartphone now most popular way to browse internet: an Ofcom report. https://www.theguardian.com/technology/2015/aug/06/smartphones-most-popular-way-to-browse-internet-ofcom.

[34] TRUSTLOOK. Android WebView Class Poses Significant Security Risk. https://blog.trustlook.com/2018/01/19/android-webview-class-poses-significant-security-risk/.

[35] 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 *ACM CCS* (2016).

[36] Wang, R., Xing, L., Wang, X., and Chen, S. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM CCS* (2013).

[37] Wikipedia. Backus-Naur form. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form.

[38] Wikipedia. Mobile deep linking. https://en.wikipedia.org/wiki/Mobile_deep_linking.

[39] World Wide Web Consortium (W3C). File API: Directories and System. https://dev.w3.org/2009/dap/file-system/file-dir-sys.html.

[40] World Wide Web Consortium (W3C). Geolocation API Specification. https://w3c.github.io/geolocation-api/.

[41] World Wide Web Consortium (W3C). HTML5 compatibility on mobile and tablet browsers with testing on real devices. http://mobilehtml5.org/.

[42] World Wide Web Consortium (W3C). Media Capture and Streams API. https://w3c.github.io/mediacapture-main/getusermedia.html.

[43] Wu, D., and Chang, R. K. Analyzing android browser apps for file:// vulnerabilities. In *Information Security Conference (ISC)* (2014).

[44] Yoni Heisler. Mobile devices become most popular way to access internet. http://nypost.com/2016/11/03/mobile-devices-become-most-popular-way-to-access-internet/.