Web-to-Application Injection Attacks on Android: Characterization and Detection[⋆]

Behnaz Hassanshahi, Yaoqi Jia, Roland H. C. Yap, Prateek Saxena, and Zhenkai Liang

School of Computing
National University of Singapore
{behnaz,jiayaoqi,ryap,prateeks,liangzk}@comp.nus.edu.sg

Abstract. Vulnerable Android applications are traditionally exploited via malicious apps. In this paper, we study an underexplored class of Android attacks which do not require the user to install malicious apps, but merely to visit a malicious website in an Android browser. We call them web-to-app injection (or W2AI) attacks, and distinguish between different categories of W2AI side-effects. To estimate their prevalence, we present an automated W2AIScanner to find and confirm W2AI vulnerabilities. Analyzing real apps from the official Google Play store – we found 286 confirmed vulnerabilities in 134 distinct applications. Our findings suggest that these attacks are pervasive and developers do not adequately protect apps against them. Our tool employs a novel combination of static analysis and symbolic execution with dynamic testing. We show through experiments that this design significantly enhances the detection accuracy compared with an existing state-of-the-art analysis.

1 Introduction

In this paper, we present a detailed study of an underexplored class of application vulnerabilities on Android that allow a malicious web attacker to exploit app vulnerabilities. It can be a significant threat as no malicious apps are needed on the device and the remote attacker has full control on the web-to-app communication channel. A successful attack can exploit web APIs (WebView) and native APIs on Android.

The Android platform provides a *web-to-app communication bridge* which enables web-to-app interaction. The web-to-app bridge is used in Android to facilitate installed applications to be invoked directly via websites. This feature has many benign uses and it is actively used by many popular applications on the official Google App Store, e.g., the Google Maps app can seamlessly switch to the Phone app when phone numbers of businesses displayed on Google Maps are clicked, without starting the Phone app.

The web-to-app bridge exposes Android apps to unvetted websites when the user visits them in a browser. Without proper sanitization on the URI or "extra parameters" derived from the URI, a vulnerable app ends up using these values to start a malicious web page in a WebView or abuse Android native APIs. While it is known that the web-to-app bridge can lead to vulnerabilities [34], in this work, we study whether existing apps are susceptible to attacks from this channel in any significant way, and if so, to

[⋆] This work has been supported in part by Huawei.

what extent. We systematically study and classify attacks which we call *Web-to-App Injection (W2AI)* Web-to-App Injection attacks are different from other recently disclosed vulnerabilities. Such vulnerabilities arise either in the implementations of hybrid mobile application frameworks, or on application code written on top of such frameworks which access external device interfaces (e.g. camera) [9, 14, 17, 26]. In contrast, attacks studied in this paper also affect Android applications via the web-to-app bridge. Furthermore, W2AI attacks can be easily combined with existing app-to-app attacks.
W2AIScanner. To enable detection for W2AI on a large scale, we describe a tool that analyzes for W2AI in Android apps. Existing static analysis techniques alone are insufficient for conducting such analysis as the complexity and size of applications limits the precision of static analysis. Dynamic analysis, such as random testing and unguided symbolic execution, face the complementary problem of path space explosion, leading to expensive analysis. In this work, we employ a refinement-based static analysis combined with dynamic testing to overcome the challenges of these individual techniques. W2AIScanner is able to automatically analyze APK files and produce working exploits (0-days). Thus, it shows a significant enhancement of the accuracy over the results generated by purely static state-of-the-art analysis. It constructs a witness exploit, a URI, to be subsequently used by security analysts or app store managers to construct specific attack payloads for determining the severity of discovered vulnerabilities.
Results. First, to measure the prevalence of W2AI vulnerabilities, we analyze applications on the official Google App Store. Of the 12,240 apps, 1,729 of them expose browsable activities. From these, we automatically constructed 286 confirmed vulnerabilities in 134 vulnerable apps (7.75% of 1,729 apps). Our findings suggest that developers often neglect the risk posed by web-to-app vulnerabilities to Android users, taking insufficient countermeasures. We contacted the Android security team to disclose the vulnerabilities to the vulnerable app vendors. The Tencent security team [3] has confirmed our reported vulnerabilities in Tencent Android SDK (2.8).

Second, we find that W2AI attacks introduce a broad range of possible exploits in installed Android applications which are analogous to vulnerabilities commonly known to occur in web applications — such as open redirect, database pollution, file inclusion, credential theft, and so on. Further, these vulnerabilities are not specific to implementations of certain application frameworks (or SDKs), as they can arise in application written in different SDKs. Third, we demonstrate that our analysis technique provides significantly higher precision than state-of-the-art static analysis techniques, at an acceptable analysis cost.

2 Overview

In Android, intents are the primary ways for an Android app to share data with other apps and to access system services [1]. An intent object carries information that Android uses to determine which component to start execution, plus information that the recipient component uses to properly perform the action, e.g., the email app can be invoked to send an email via an intent by any other app.

A web page can invoke a component in an installed app if the target app declares one or more of its activities as being `BROWSABLE` in the app's manifest. When a user clicks a web hyperlink, in a certain format, Android translates it into an intent. We call the intents created from such hyperlinks as *URI intents*. We use *intent hyperlink* to refer to the link or its string, while URI intent refers to the workings of the mechanism in Android. Intent hyperlinks carry parameters contained in the hyperlink, the fragment identifier, and information about the target activity specified as a tuple $(scheme, host, path)$, etc.

2.1 Web-to-App Injection Attacks

URI intents expose a new channel of attacks targeting installed apps. We present the first comprehensive study of web-to-app injection (W2AI) attacks in Android.

Threat Model. In a W2AI attack, we assume that the adversary is a standard web attacker [4], who controls a malicious website. To expand the coverage of victims, the attacker can disseminate the shortened URL of the malicious site through emails, social networks, ads, etc. We make the following conservative assumptions. We assume that the victim, Alice, only installs legitimate apps from Google Play on her Android device. We also assume that at least one app with adequate permissions on her device is benign but buggy, hence a W2AI vulnerability exists.

W2AI Attacks. Analogous to a conventional web attack, when Alice directly visits the malicious website or clicks a link redirected to the site, a W2AI attack occurs. A generic scenario for W2AI follows: 1) The attacker crafts and publishes a malicious intent hyperlink in a social network; 2) A user clicks the malicious link redirecting to the attacker's site in her mobile browser; 3) The site loads the malicious intent hyperlink in an iframe or a new tab; 4) The browser parses the hyperlink, generates the URI intent and launches the corresponding activity in the vulnerable app; and 5) Hence, the payloads derived from the URI intent running in the app can access the user's private information or perform privileged operations on behalf of the app.

2.2 Categories of W2AI Vulnerabilities

Android applications typically use data derived from the intent hyperlink with Android API interfaces which can be divided into two categories, WebView and native interfaces. If the attacker-controlled data are used in these interfaces without any validation, the attacker can feed payloads to abuse them. We divide the W2AI vulnerabilities into: (i) abusing WebView; or (ii) abusing Android native app interfaces.

W2AI vulnerabilities arise due to dataflows in the native Android code, rather than in application logic written in HTML5 code [9, 14, 17, 26]. Unlike other vulnerabilities that exploit app-to-app communication interfaces [8, 24, 40, 42], we emphasize that W2AI attacks do not need an installed malicious app on the device to launch attacks.

Abusing WebView Interfaces. WebView is a browser component provided by Android, which provides the basic functionalities of normal browsers (e.g., page rendering and JavaScript execution) and enables access to various interfaces (e.g., HTML5 APIs and JavaScript-to-native bridge). Certain applications take parameters in the intent hyperlink and treat them as web URLs, thereby loading them into WebView during execution. When this happens, the attacker's HTML code runs in the WebView. Furthermore, if the vulnerable application enables execution for JavaScript in the WebView, the attacker can run JavaScript in its HTML page, and can access all interfaces exposed to it by WebView. We classify the vulnerabilities arising from unfettered access to the exposed interfaces into 4 sub-categories:

1) Abusing JavaScript-to-Native Bridge. JavaScript code loaded in the WebView can access native methods via the `android.webkit.JavascriptInterface`. The accessible native methods are specific to each application. In our experiments, we have found up to 29 distinct JavaScript-to-native interfaces accessible by a single app, e.g., many apps enable access to interfaces that retrieve the device's UUID, version and name, thereby opening up the threat of privacy-violating attacks. Furthermore, several interfaces allow reading and modifying the user's contact list and app-specific local files.

2) Abusing HTML5 APIs. WebView enables access to standard HTML5 APIs, akin to normal web browsers, e.g., if the vulnerable app has the proper permissions and WebView settings, the attack web page running in the WebView can use JavaScript to call the HTML5 geolocation API. We found 29 apps with such tracking vulnerabilities.

3) Local File Inclusion. When the user visits the malicious site, the site can trick the browser to automatically download an HTML file into the user's SD card by setting the HTML file as not viewable. When the site triggers the browser to parse the intent hyperlink that refers to the downloaded HTML file, e.g., `file:///sdcard/Downloads/attack.ht-ml`, it launches the vulnerable app to load the HTML file in its WebView. If the vulnerable app has certain WebView settings, the malicious JavaScript code in the HTML file can read any files under the app's directory or the readable system files (e.g., `/etc/hosts`) and send them to the attacker.

4) Phishing. The attacker's web page can impersonate or phish the user interface of the original application. Since there is no address bar displayed by WebView that users can use to inspect the current page's URL, users cannot distinguish the phishing page from the normal page. Such attacks via the web-to-app bridge are harder for users to discern than the conventional phishing attack on the web [12].

Abusing Android Native App Interfaces. Android Apps, even if not using WebView, can expose native Android interfaces to URI intents if input is not properly sanitized. These lead to the following four categories of exploits:

1) App Database Pollution. Android provides native interfaces for apps to execute SQL query statements to manage the app's database. Therefore, if the SQL queries are derived from the URI intent, it allows the web attacker to pollute (e.g., add or update the table's fields) the vulnerable app's database.

2) Persistent Storage Pollution. Android native interfaces enable apps to store persistent states, e.g., authentication tokens, in the persistent storage (e.g., `SharedPreferences` and local files). Many vulnerable apps directly treat the parameters from the URI intent as the content to add or update the persistent storage. An attack URI intent can pollute the target app's persistent storage.

3) Open Re-delegation. Android native interfaces provide the ability to launch specific activities addressed by name. If the name parameter is derived from URI intent, it allows the web attacker to invoke any in-app private activities directly, which are not required to be marked browsable. Moreover, attacker might embed an additional intent hyperlink as a parameter to the original intent hyperlink and force the benign app to invoke another app. This leads to a broad range of problems such as permission redelegation [13]. Permission re-delegation is a confused deputy problem whereby a vulnerable app ac-

cesses critical resources under influence from an adversary. Though these attacks are previously known to be possible via the app-to-app [13], we show that they can be affected under influence of a website through the web-to-app bridge.
4) App-Specific Logic Flaws. Android enables apps to perform various operations (e.g., popping up messages) via native interfaces. Due to the app-specific logic flaws, the vulnerable app directly uses the data from the URI intent as parameters to these operations, e.g., we found that an attacker can exploit vulnerable apps to display a fabricated PayPal transaction status.

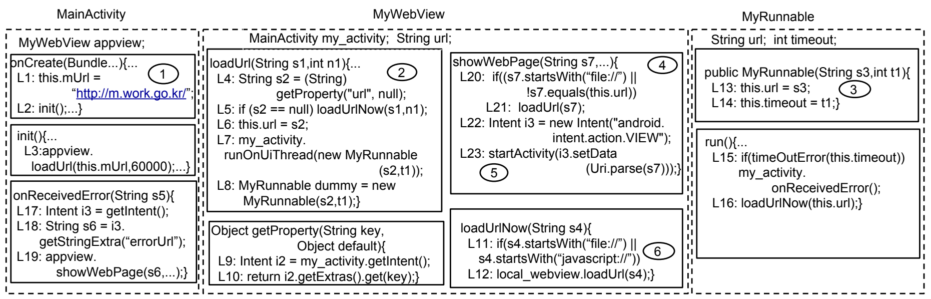
Now, we use a real app as an example to explain how the W2AI attack works. The example app is WorkNet (3.1.0), a Korean information app with 1-5M downloads. It has a browsable activity, `kr.go.keis.worknet.WorknetActivity`, which loads arbitrary URLs in URI intents and is vulnerable to the following W2AI attacks: abusing JavaScript-to-native bridges, abusing HTML5 APIs, local file inclusion and phishing. The attack's life cycle is as follows: (1) The attacker hosts a malicious website, which loads an intent hyperlink (`"intent://#Intent;scheme=worknet;action=android.intent.action.VIEW;S.url=http://attacker.com;end;"`) into a new tab using `window.open`. The attacker posts the site's shortened link on social networks, e.g., Facebook. (2) When the user visits the attacker's site (by clicking the link on social networks, ads, and so on) in her Android browser, the site loads the hyperlink in a new tab. (3) The user's browser parses the hyperlink to the URI intent which contains extra parameters and launches the `WorknetActivity` activity with the intent. (4) The activity loads the URL (`http://attacker.com`) derived from the malicious URI intent into the WebView without proper validation. Now the attacker's site is loaded with its JavaScript code running in the WebView. The attacker can utilize whatever is available to this activity, e.g., abusing JavaScript-to-native bridges.

We find that WorkNet has 21 such interfaces, e.g., accessing contacts, local files, device information, etc. Furthermore, being a WebView app, the attacker's site can mimic the UI of the original page. In the background, the scripts access the user's private data (e.g., device information and contacts), sending them to the attacker's server. In addition to abusing the JavaScript-to-native interfaces, the web attacker can also abuse HTML5 APIs to track Alice's geolocation and leak the content of local files via file inclusion in this app. From this example, we can see that the W2AI attacker can not only mount conventional web attacks (e.g., unvalidated redirect in the example), but can also hijack the vulnerable app to perform privileged operations on sensitive resources (e.g., local files and contacts) without any installation of malware in the user's device.

2.3 Detection Challenges

Our aim is to design a system which both *detects* and also *confirms* W2AI vulnerabilities. The target of W2AI attacks are *sinks* defined as sensitive/critical Android and Java APIs used to inject data which make the application vulnerable. API calls which fetch intent objects containing data under the control of the attacker are called *sources*.

At the high level, detecting W2AI vulnerabilities can be considered a source to sink reachability analysis for Android apps. In general, static analysis techniques would give potential reachability and existing techniques for Android apps [5, 16, 25] are no different. However, there can be many potential source-sink flows detected with possibly



```
MainActivity
MyWebView appview;

onCreate(Bundle...){...
  L1: this.mUrl =
        “http://m.work.go.kr/”;
  L2: init();...}

init(){...
  L3:appview.
    loadUrl(this.mUrl,60000);...}

onReceivedError(String s5){
  L17: Intent i3 = getIntent();
  L18: String s6 = i3.
      getStringExtra(“errorUrl”);
  L19: appview.
          showWebPage(s6,...);}
```

```
MyWebView
MainActivity my_activity;  String url;

loadUrl(String s1,int n1){...
  L4: String s2 = (String)
            getProperty("url", null);
  L5: if (s2 == null) loadUrlNow(s1,n1);
  L6: this.url = s2;
  L7: my_activity.
    runOnUiThread(new MyRunnable
                         (s2,t1));
  L8: MyRunnable dummy = new
        MyRunnable(s2,t1);}

Object getProperty(String key,
                 Object default){
  L9: Intent i2 = my_activity.getIntent();
  L10: return i2.getExtras().get(key);}

showWebPage(String s7,...){
  L20:  if((s7.startsWith(“file://”) ||
                !s7.equals(this.url))
         L21:  loadUrl(s7);
  L22: Intent i3 = new Intent("android.
                    intent.action.VIEW");
  L23: startActivity(i3.setData
                      (Uri.parse(s7)));}

loadUrlNow(String s4){
  L11: if(s4.startsWith(“file://”) ||
    s4.startsWith(“javascript://”))
  L12: local_webview.loadUrl(s4);}
```

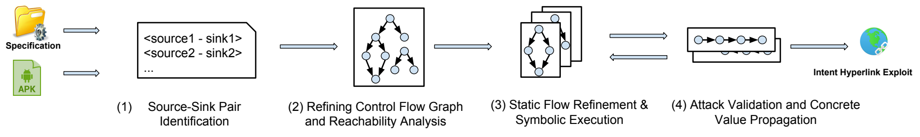
```
MyRunnable
String url;  int timeout;

public MyRunnable(String s3,int t1){
  L13: this.url = s3;
  L14: this.timeout = t1;}

run(){...
  L15: if(timeOutError(this.timeout))
        my_activity.
              onReceivedError();
  L16: loadUrlNow(this.url);}
```

Fig. 1: An execution sequence which retrieves malicious parameters from an intent hyperlink. There are three classes separated by dashed lines: `MainActivity`, `MyWebView` and `MyRunnable`. `MainActivity` is the browsable activity, `MyRunnable` is an inner class of `MainActivity` which implements `Runnbale` interface. Methods are shown in boxes.

many false positives (i.e., potential vulnerability is signaled as a flow, even though it can never occur during execution) as we show in Sec. 4. Our goal is to not only analyze for potential W2AI vulnerabilities but also to confirm the vulnerabilities. For this more ambitious goal, we want to be able to generate intent hyperlinks which actually reach and also affect a sink, in other words, automatically generate a form of *0-day W2AI attack*. This makes the task of understanding and confirming a vulnerability significantly easier for security analysts or the app developers.

The complexity of the Android environment and apps also raises practical challenges. Figure 1 shows a simplification of the code of the WorkNet app, explained in Sec. 2.2, which has W2AI vulnerabilities. The browsable activity that is triggered by intent hyperlinks is `MainActivity`. When the user clicks on the malicious intent hyperlink in the default browser, the system generates an intent, launching `MainActivity`. Unlike Java programs, Android apps do not have a main method. When an intent invokes an activity, the Android runtime invokes the `onCreate()` or `onNewIntent()` callback methods. Next, the `getIntent()` and `onNewIntent()` methods obtain the intent messages. Once an intent is sent to an activity, any invocation of the `getIntent()` method throughout the activity yields the same intent until `setIntent(Intent)` is called. Thus, the intent objects at lines L9 & L17 will refer to the same intent hyperlink.

We explain the possible execution paths in Fig. 1 where the browsable activity loads malicious parameters in a malicious intent hyperlink clicked by the user: (1) The `MainActivity` is launched and `onCreate()` is invoked storing the default URL in `this.mUrl` used by `loadUrl()` at L3. (2) However, the application does not load the default URL into the WebView immediately. Instead, `getProperty()` is called which invokes `getIntent()` at L9. This method looks for the “extra parameter” (i.e., the parameter returned by `get[type]Extra()` API with type string), having the key `"url"`. If this parameter exists in the URI intent, `runOnUiThread()` at L7 is called which runs the `MainActivity`’s UI thread. (3) Next, `MyRunnable` class is instantiated storing the malicious URL in `this.url` and `run()` method is invoked by the Android runtime. Line L15 in `MyRunnable` forks a thread (not shown) to check whether the network connection times out within `timeout` limit. In case of timeout, it calls `onReceivedError()` in the `MainActivity` which looks for another extra parameter with key `"errorUrl"` at line L18. (4) If the string conditions at line L20 are met, a



Specification
APK
<source1 - sink1>
<source2 - sink2>
...
(1) Source-Sink Pair Identification
(2) Refining Control Flow Graph and Reachability Analysis
(3) Static Flow Refinement & Symbolic Execution
(4) Attack Validation and Concrete Value Propagation
Intent Hyperlink Exploit

Fig. 2: Architecture of W2AIScanner

string from the malicious URL is eventually loaded to the WebView (path 1 with sink 1, `loadUrl()`, at line L12). (5) Otherwise, the string will be incorporated into a new intent and attack suceeds to start another app (path 2 with sink 2, `startActivity()`, at line L23). (6) Alternatively, the malicious URL obtained at line L4 is loaded by the WebView (path 3 with sink 1, `loadUrl()` at line L12).

In this example, there are 2 vulnerable sinks at lines L12 and L23 with 3 paths to reach them. However, analyzing these vulnerabilities require dealing with challenges that are not currently dealt with satisfactorily in existing systems. The main reason is limitations in constructing the control flow graph (CFG) from the Dalvik code. We saw that paths 1 and 2 occur due to (nested) inner threads. The app also uses `runOnUiThread` which changes execution to the main UI thread of the activity. Existing tools such as FlowDroid [5] do not report this path since the generated CFG lacks the edges necessary for the vulnerable paths. We remark that we have found 818 browsable apps in our dataset which use threads. In this example, the `getIntent()` invocations happen to give the same intent message in all the code. In general, analysis needs to determine what intent `getIntent()` refers to. The example also shows that the analysis needs to be field-sensitive, since the malicious URL is stored in `this.url` field and also object-sensitive to refer to the correct instance of `MyRunnable` class.

Our analysis not only aims for accuracy in finding the paths for the source-sink flow but also needs to generate instances of intent hyperlinks to confirm the vulnerability. We use symbolic reasoning on strings and other constraint solving in our analysis (lines L11, L20) to this end. In addition, the operations on intent parameters can be dependent on the intent filters in the app manifest. Hence, in addition to the bytecode analysis, intent filters from the app manifest need to be taken into account in the analysis.

3 Detecting and Confirming W2AI vulnerabilities

We describe a tool, W2AIScanner, which can automatically detect and confirm W2AI vulnerabilities. In order to deal with the challenges described in Sec. 2.3 and also automatically confirm vulnerabilities, W2AIScanner works in phases as shown in Fig. 2. We now describe each of these phases.

3.1 Source-Sink Pair Identification

Our design starts with a less precise analysis continued with an on-demand refinement of the analysis. The more efficient but less precise analysis identifies potentially vulnerable areas in the program that benefit from a more precise analysis which can reduce false alarms but may suffer from state explosion problems.

We start with a specification including a set of sources and sinks. The sources are the `getIntent()` and `onNewIntent()` methods that fetch the intent objects which start the activity and provide data inputs to the app. We choose a subset of the sinks provided by Susi [29] and add more if needed based on the categories described in Sec. 2.2.

The initial CFG used by our analysis is the inter-procedural control flow graph in Soot [21] constructed based on the call graph created by SPARK [22]. In the first step, we generate pairs of source and sink program points for the given specification. We have two design choices: (i) locating all possible program points in the initial CFG by comparing the method signatures in the reachable methods; and (ii) using an existing information flow analysis system like FlowDroid [5] to collect source-sink pairs with data dependency on inputs.[1]

We have observed that using FlowDroid, we need to perform the analysis for fewer source-sink pairs and remove some of the irrelevant pairs, thereby decreasing the analysis time. Hence, we use FlowDroid with a conservative setting. For instance, it is possible to choose the flow-sensitivity of the backward alias search and conservatively, we choose it to be flow-insensitive. Starting from the browsable activities,[2] FlowDroid finds pairs of source and sink program points using dataflow analysis. In the next step, we utilize these source-sink pairs for reachability testing and refining the initial CFG constructed by FlowDroid.

3.2 Refining the Control Flow Graph & Reachability Analysis

The less precise dataflow analysis in the previous step may introduce false alarms and the constructed CFG may also miss edges (informally, we call them as gaps). We compensate for this inaccuracy by a subsequent on-demand refinement and symbolic execution. We start with the initial CFG from the previous step. Given a source method, S_c, and a sink method, S_k, W2AIScanner starts traversing and refining the CFG with S_c being the starting node using depth-first search. We resolve virtual method and interface calls using a backward variable type analysis which considers assignments between callsite and class object instantiation program points. In our motivating example, a node for method `run()` in `MyRunnable` class is added because the CFG misses the edge from L7 to this method. In this example, the class implementation for the `Runnable` interface at L7 is resolved to `MyRunnable` class. Then, the `run()` method is loaded and its nodes are added to the CFG. Space does not permit discussion about other cases to make the CFG more accurate.

While constructing the CFG, a reachability analysis is also performed to reduce the state explosion problems in the symbolic execution phase. When a branching node is visited, it examines whether the S_k is reachable from each of the branches and caches the reachability result. The CFG construction stops in this step if the S_k is reached and continues for the next source-sink pair. If a new sink is detected, it will be added to the source-sink pairs to be examined later by the symbolic execution. In Sec. 4, we show

[1] FlowDroid [5] is a static state-of-art analyzer for Android built upon Soot [21] (based on the Interprocedural Finite Distributive Subset (IFDS) algorithm [30]) and incorporates the Android component lifecycle.

[2] We have modified the entry point selection implementation to pick the browsable activities.

that accurately handling threads helped us to find interesting vulnerabilities that could not be detected by FlowDroid [5].

One problem is that S_c can be invoked anywhere in the program. Therefore, the caller of the method where S_c resides might not be known (e.g., line L9 in Fig. 1). Our analysis is conservative, thus, it returns to all possible callsites to continue the analysis. Note that a path might have more than one sink. In that case, the analysis continues until it reaches the S_k sink.

In order to deal with backward edges caused by loops and recursive calls, a node in a specific calling context is visited within a bounded number of times. Later we unify all the reachability results for nodes visited in different calling contexts and use them in the symbolic execution. If a path does not include any backward edges but is too long, we enforce a depth limit for the depth first search.

3.3 Symbolic Execution and Static Flow Refinement

Static analysis is generally not sufficient to confirm vulnerabilities. Rather, concrete execution is needed for such confirmation. However, concrete execution requires input, in the form of an (attack) intent hyperlink. We employ symbolic execution [15, 19] commonly used for automated test generation to help in generating the input. The final generated intent hyperlink is a combination of the symbolic execution and validation phases. Our symbolic executor does not require any initial inputs and there are optimizations to reduce the number of paths that need to be explored using the sink reachability analysis conducted in the previous step.

The initial dataflow analysis used for reachability analysis in the first step might produce a large number of flows, many of which may be false positives. Thus, a strategy is required to reduce the number of initial flows. W2AIScanner achieves an initial reduction by removing infeasible paths using symbolic execution. A path is feasible if there exists a program input for which the path is traversed during program execution, otherwise the path is infeasible [20]. So we immediately remove the infeasible paths.

The symbolic executor works on a worklist of statements. Analysis picks a source-sink pair, S_c-S_k, starts from the source statement S_c and symbolically executes the program until it reaches the sink S_k. The reaching definition analysis starts simultaneously and the intent object returned by the S_c method is marked as data dependent on input. From this point, any parameter extracted from the source intent object that has string, numeric, URI or boolean types are stored in a symbolic variable. A URI can be decomposed into many substrings. We model the URI class and convert it to string and integer compartments. Symbolic variables are stored in a symbolic variable pool which is updated when a definition statement is translated. If a statement has a call invocation, it has to decide whether to enter the method or not. Execution enters a method if the sink reachability result shows that entering the method will lead to a program point where S_k is invoked. If the method is available (i.e., it is not external method) and the method callsite has a definition of a variable, the flow fact in the callsite is updated when analysis returns from the method. Otherwise, the method call is considered dependent on inputs if any of its use variables (arguments or the instance variable) are dependent.

For IF statements, the sink reachability result is examined for each of the branches. If none of the branches are reachable to the S_k, no new job will be added to the worklist and the next path will be traversed. If only one of the branches is reachable, that branch will be taken. Finally, if both branches are reachable to the S_k, W2AIScanner will search for the immediate postdminator (ImPodm). Based on the CFG of a method, if W2AIScanner finds an ImPodm inside the method, a new pending merge state will be added to the merge stack.

An optimization arises during the analysis: if there is no ImPodm inside the method, we cannot merge the two branches which are reachable to the same sink S_k. If one of these branches does not contain input-dependent variables, forking it will produce spurious paths whose constraints will not be used for generating inputs. To avoid these paths, we introduce a dummy ImPodm: (1) we add a merge state to the merge stack when execution reaches an always feasible conditional statement and the mergepoint can be any of the exit statements of the method. An exit statement is a program point where execution exits a method; (2) when execution reaches any exit statement, it does not exit the method. Instead, if the method contains another distinct path, that path is added to the worklist; (3) finally, when all paths inside the method are traversed and execution is exiting the method, the states at all of the exit statements which are data dependent on input and the constraints for class fields are merged and there will be only one merged state for all exit statements. In order to choose the program counter for this dummy ImPodm, we create a dummy exit statement. The data dependency results are also utilized to remove irrelevant constraints on the path formula if possible.

This step also involves a reaching definition analysis performed together with the symbolic execution. This analysis is field-sensitive and distinguishes objects originating at different allocation sites but reaching the same program point. We use symbolic values to point to a heap object. In an execution path, there may be variables whose values are used but not resolved, we employ an (on demand) backward copy constant search for more accuracy. The values are over-approximated in two ways: (i) The variable is a method parameter where we consider all possible callers of the method. Therefore, the result might be a set of possible values which will be considered one by one; (ii) The variable is a class field, so we do an over-approximation by considering all of the definition statements for this field variable in its declaring class.

In practice, symbolic execution on real world applications on a large codebase face some additional challenges. The backward edges due to loops and recursions or long paths may lead to scalability issues. In particular, loops pose many challenges in the analysis since even the Android activity lifecycle itself is a large loop. W2AIScanner employs a bounded symbolic execution and models iteration blocks of code (e.g., `Iterator` class in Java) to address these obstacles.

Threads. One challenge in supporting threads is passing arguments. Usually threads are initialized with arguments that are stored in class fields. Later, these class fields are queried in the body of the `run` methods and a field-sensitive analysis is required to keep track of them. Method arguments can also be passed to threads in specific ways (e.g., `AsyncTask`) which are more complicated than binding method arguments in the callsite for normal method invocations (where there exists a one-to-one mapping between actual parameters at callsite and formal parameters of the method). We model different ways provided by Android to use threads and also support binding arguments for them.

Once we get the abstract description for all the sinks and external methods in terms of formulas, we solve them and check the feasibility of each path. For feasible paths, a solution to the constraints is a witness which can be used to construct an intent hyperlink to drive the execution down this path. These intent hyperlinks are used at the last step to dynamically execute the program. We employ the CVC4 SMT solver [23] which deals with string, integer and boolean constraints. We provide intermediate formulas for string operations that are not directly supported by CVC4 such as (startsWith and split). Once the solver has instantiated some/all of the symbolic variables, we use them to instantiate an intent hyperlink. In order to incorporate the generated inputs to the intent hyperlink, our analysis resolves the key-value mappings (explained shortly) in the intent hyperlink syntax.

3.4 Attack Validation and Concrete Value Propagation

W2AIScanner automatically generates intent hyperlinks that can exploit the W2AI vulnerabilities. An intent hyperlink can be divided into two parts: (i) the scheme part which has to be matched with the intent filter for the activity defined in the manifest file; and (ii) the data inputs which are of key-value forms described below. The first part is collected by the manifest parser component which retrieves the intent filter specification for the source activity. It creates all possible schemes that will match the intent filter. Path is one of the elements in intent filters that will be checked for accepting an intent. Developer can specify a special form of regular expressions to match the intent hyperlink path pattern. We implement the algorithm used in the Android framework to match against intent hyperlink path patterns and we generate counter examples.

The data inputs which make up an intent hyperlink are derived from the Intent class methods. In Sec. 2 we discussed that an intent hyperlink follows a specific syntax. Here is a simplified meta intent hyperlink:

```
intent:HOST/PATH?query=[string1]#Intent;action=[string2];scheme=[string3];S.key
=[string4];end;
```

where data input can be sent through the [$string$] fields to the Android application code. There are several possible ways to send data via an intent hyperlink: (i) a data URI which references the data resources consists of the scheme, host and path as well as query parameters which are the key-value mappings preceded by the "`?`"; (ii) Intent extras, the key-value pairs whose type can also be specified in the Intent URI (e.g., the S in S`.key=[`$string_4$`]` refers to the string extra). Note that an intent hyperlink can have more parameters with other types, e.g, int; (iii) other Intent parameters such as categories, actions, etc., that can be sent as string values.

As we explained in the previous step, there are Intent APIs for each form of the inputs described above that can be utilized in the application code to get the data inputs. For instance, `Intent.getStringExtra(String key)` returns the extras in the intent whose type is string and is mapped to `key`. We infer such types and use them in generating intent hyperlinks. We define such methods as entry methods if they are invoked on an input intent object. These methods are considered as the input methods in the symbolic execution which generates test inputs for variables initialized by these entry methods. While constructing symbolic formulas in the previous step, we also correlate the entry methods with the intent filters in the manifest file to generate more accurate intent hyperlinks. The entry methods return string, integer and boolean types as well as the URI type.

We also need to find keys corresponding to each input parameter. We use constant propagation, explained in the previous step to find the values of the arguments of API calls such as `getStringExtra(String name)`. If it fails to resolve the key names, an arbitrary string value is generated.

Once we have the key-value pairs and other necessary inputs for the source-sink flows and the intent filter specifications for the target browsable activity, these can all be put together to form an intent hyperlink. Therefore a group of paths generated in the symbolic execution phase might contribute to a single intent hyperlink.

Attack Validation. The intent hyperlinks generated in the static phase are used by the dynamic executor explained below to validate whether they exploit the sink methods. The dynamic executor runs the generated inputs and inspects the results. Running the generated inputs, two possible scenarios might happen: (1) the sink method is invoked at runtime and the generated input is accurate enough to cause the exploit; (2) the sink method is invoked but it is not exploited. In this case, first we use the concrete values obtained from the runtime execution path and assign them to the unknown variables which symbolic executor has failed to resolve. The new path formula is passed to the solver again and we generate a new intent hyperlink. This procedure continues until intent hyperlinks do not change any more (i.e., analysis reaches a fixpoint).

In order to run the concrete generated inputs and obtain the execution trace, we chose to use a high-level but standard interface, Java Debug Wire Protocol (JDWP) [2] which is supported by the Android runtime (both Dalvik and Art) and independent of framework releases. One complexity is that the execution is run in Dalvik bytecode but the analysis is in Jimple (a 3-address instruction representation). We re-translate the generated execution trace back to Jimple. Dexpler [6] keeps a mapping between byte code instruction addresses and Jimple statements. We fetch the Jimple statements using these mappings. In order to assign the concrete value of a variable from execution trace to Jimple registers, for each method, we have to find the relation between variables on the execution stack and the Jimple registers in the method Body. After running the generated intent hyperlinks, we will use these register mappings to find out accurately which Jimple registers should be updated to be further processed during the analysis.

The validation component has to verify whether the generated intent hyperlink results in an exploit. This decision is based on the execution trace, concrete values and the attack policies provided by the security analyst. The attack policy consists of rules for each class of vulnerabilities. Depending on the category of the sink method reached on the execution trace, it applies different policy checks. There are two main classes of vulnerabilities: abusing WebView interfaces and abusing native app interfaces.

The first category is validated by sending a malicious website URL through the intent hyperlink parameters. When a vulnerable application loads the malicious URL, the data retrieved from the device is sent to our server and we can confirm the attacks

Table 1: Overall Statistics of Vulnerable Apps in each W2AI Attack Category

Category	Sub-Category	# of Sinks	# of Vulnerable Apps	ID
Abusing WebView Interfaces	Abusing JavaScript-to--Native Bridge	9	52	1
	Abusing HTML5 APIs	10	29	2
	Local File Inclusion	9	63	3
	Phishing	11	84	4
Abusing Native App Interfaces	App Database Pollution	14	10	5
	Persistent Storage Pollution	72	7	6
	Open Re-delegation	39	23	7
	App-Specific Logic Flaws	16	18	8

accordingly.[3] Attacks which abuse native app interfaces are more complex to validate. First we verify if the sink method is reached on the execution trace. But this is not sufficient. We should also check whether the concrete values of the sink method parameters are directly affected by the intent hyperlink fields. For this purpose, we compare the values resolved for the sink method parameters in the symbolic execution phase with the values observed after running the intent hyperlink. Then, according to the policy, we check for other methods (which we call category settings) on the path that should exist so that the exploit is not prevented from occurring. After confirming the sink method to be exploitable, the intent hyperlink will be reported as an exploit.

4 Evaluation

We assess the prevalence of web-to-app injection attacks on a large scale and also assess the detection capabilities of W2AIScanner. We choose the top 100 apps of all categories in Google Play plus the dataset used in [17].[4] We have conducted a systematic analysis on 12,240 apps in Ubuntu 14.04 on an Intel Core i5-4570 CPU PC desktop (3.20GHz) with 16 GB of RAM. Apps are tested both on the Android 4.4 emulator as well as real Android devices, Galaxy Nexus and Nexus 7. W2AIScanner utilizes the `adb` command to launch the activity to validate the exploits that abuse WebView interfaces or native app interfaces and perform privileged operations (e.g., inserting data into the app's database) which is explained in Sec. 3.4.

Prevalence of W2AI vulnerabilities in Apps. We ran W2AIScanner on 1,729 apps and detected 286 W2AI vulnerabilities in 134 apps. This shows that our system is effective as a vulnerability detection tool for W2AI attacks. (We, in fact, process 12,240 apps, first rejecting those without browsable activities).

Table 1 gives a breakdown of the detected vulnerable apps into our 8 categories. The column, # of sinks, gives the number of sinks defined by our specification for that

[3] As an example, if the app has flows that reach the `WebView.loadUrl` sink and enables `setAllowFileAccess`, `setJavaScriptEnabled` and `setAllowFileAccessFromFileURLs` settings, the app is vulnerable to local file inclusion attacks.

[4] Among which 1,729 apps have at least one `BROWSABLE` activity. Since numerous apps were out of the shelf (the dataset in [17] contains 15,510 apps), we could download only 9,877 apps on Google Play on April, 2014.

category. There can be overlaps among the different categories of sinks. For example, `WebView.loadUrl` can be the sink for the first 4 categories. In total, we have 153 distinct sinks for 8 categories. An app may have vulnerabilities from more than one category. For instance, the WorkNet example has vulnerabilities from categories with ID 1 to 4. Thus, the sum of that column is greater than the number of vulnerable apps.The column, # of vulnerable apps, gives the number of apps for which we found a confirmed vulnerability for that class.

For each category, we have found at least one vulnerable application with more than 1 Million downloads. A popular application is Wikipedia (1.3.4) which is vulnerable to categories with ID 2 and 4. We have also detected and confirmed 14 Dropbox applications that are vulnerable to open-redelegation attacks where attacker can force them to invoke other apps hosting on the phone. One app-specific logic vulnerability appears in 587 apps, here we count it as *only one unique vulnerability* to avoid to skewing the results. Once this vulnerability is exploited, attacker can send fake Paypal payment notifications to the phone. Tencent Android SDK (2.8) is also confirmed to be vulnerable to W2AI attacks. More details on 8 representative applications in each attack category is given in Appendix A.

Effectiveness of W2AIScanner in detecting W2AI vulnerabilities. Our objective is effective detection of W2AI vulnerabilities with the following goals: (i) potential vulnerabilities found by the analyser should have only few false positives and the generated intent hyperlinks should be accurate; and (ii) it should find vulnerabilities which may be missed due to imprecision at the initial whole app-level analysis phase.

Fig. 3-(a) depicts the ratio of number of paths generated by W2AIScanner and those reported by vanilla FlowDroid. For most of the apps, there is a considerable reduction in the number of reported flows which means that either most of the false positive flows are rejected or the combination of symbolic execution and data-flow analysis has effectively reduced the number of generated paths. The ratio can also be greater than one as we detect flows not found by FlowDroid. Fig. 3-(c) shows that W2AIScanner is able to effectively detect false positive sinks. Sometimes, our system is even able to find sinks which have been missed by vanilla FlowDroid. In Fig. 3-(c), these sinks are shown as new_sinks. Note that if we don't find any new sinks for one app, we don't put 0 in the chart. In some cases, all of the sinks reported by FlowDroid are false positives while W2AIScanner finds the true positive ones. In total, we find 82 new true positive sinks in 69 applications after refining the CFG constructed by FlowDroid. The new sinks found in 39 applications are due to thread executions. Fig. 3-(b) shows the number of missing edges in FlowDroid CFG for each application in our dataset. In total, we find missing edges in 863 apps which are due to thread invocations.

The total execution time for static analysis phase can be found in Fig. 3-(d). For most of the applications analysis takes less than 30 seconds. The execution time for dynamic analysis phase tends to be higher. We have measured the execution time per flow for 8 applications each representative for each attack category. The average execution time per flow is around 48s. A large portion of the cost for the dynamic phase is due to operations such as networking.

In Fig. 3-(a), it can be observed that for the first 200 apps, the number of paths reported by vanilla FlowDroid is much higher than W2AIScanner (the ratio is less than

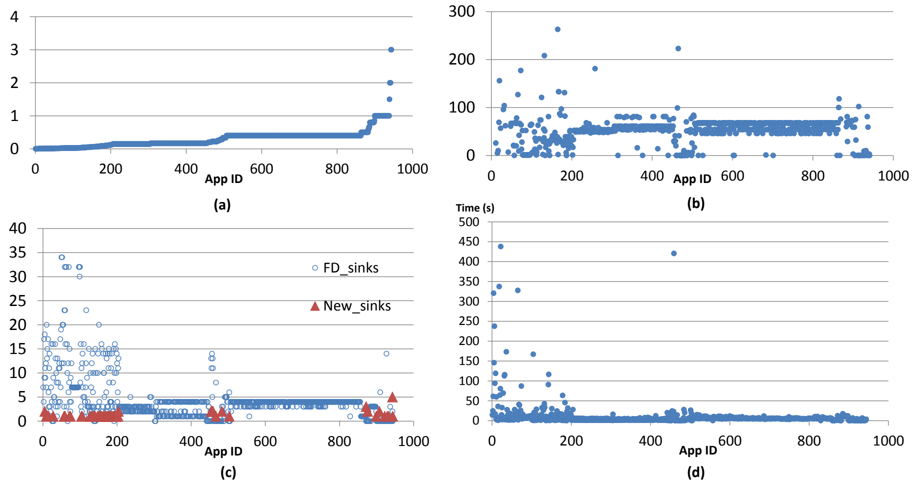


Fig. 3: (a) Ratio of number of paths generated by W2AIScanner and vanilla FlowDroid; (b) Number of missing edges in the initial CFG which were found and added by W2AIScanner; (c) FD_sinks are number of FlowDroid false positive sinks and new_sinks are number of new true positive sinks found by W2AIScanner; (d) Total execution time for static analysis in seconds. Apps are sorted based on the ratio in figure (a). All apps have at least one potential vulnerable sink.

0.2). Fig. 3-(c) also shows that FlowDroid has many false positive sinks for the same apps. This shows that our system can successfully reduce the number of generated paths for these apps by rejecting the false positive sinks. The high number of initial flows for these apps also results in more runtime execution in Fig. 3-(d).

We successfully generate accurate intent hyperlinks that allow us to find 0-day vulnerabilities. The intent hyperlink parameters generated for 236 of applications in our dataset follow complex patterns. For example, Letv is an Android app which only processes an intent hyperlink if it has a query parameter with `from` as the key and `baidu` as value. Another example is Kobobooks which requires that action parameter of the intent hyperlink that invokes the app be not equal to `android.intent.action.VIEW`. Thus, symbolic execution and validation is the key for finding confirmed paths. An alternative approach to symbolic execution is fuzzing but we believe that any fuzzing without some symbolic reasoning is unlikely to give good results.

5 Related Work

We discuss Android related work from two angles, attacks on apps and analysis of apps. Privilege escalation attacks have been shown in Android [8,10,13,24,31,32,36,39,40]. These works all assume that the malicious apps are present on the victim's Android device, while our W2AI attacks work without any installation of malware.

Recently, WebView and hybrid apps have been shown to be vulnerable to new classes of attacks [9, 14, 17, 26]. Luo et al. observe that malicious JavaScript code can access sensitive resources [26]. Georgiev et al. carry out an analysis on hybrid apps, and demonstrate vulnerabilities that affect bridge mechanisms [14]. Jin et al. introduce code injection attacks on HTML5-based mobile apps via internal and external channels [17]. These attacks require the user to visit the malicious page *directly* in the WebView of the hybrid apps. In contrast, our W2AI attacks utilize intent hyperlinks to convey the payload simply by clicking a link in the default browser, which is more probable.

Attacks have also been found through scheme mechanisms [18, 33, 34]. Wang et al. [34] reveal confused deputy attacks on Android and iOS applications which abuse channels provided by the OS. One of these channels is the scheme mechanism through which an attacker can invoke apps on the phone by crafting intent hyperlinks and publishing on web. They study the problem where the user surfs through the web in customized WebViews of benign applications and launch confused deputy attacks abusing the benign app's "origin". They present a CSRF attack on the Dropbox SDK in iPhone [34] launched through an intent hyperlink. However, our attacks differ because our attack model is more general – the user clicks on an intent hyperlink in the default browser which does not need to be started from the benign app and can leverage safer channels like default browsers. More importantly, we investigate which vulnerabilities can be exploited once the attacker can manage to start an application via an intent hyperlink. We present a detection and validation method which we show is able to scale for automatically detecting and generate exploits for vulnerabilities in real apps.

Another approach is static analysis of Android apps [5,25,35,41]. CHEX [25] finds component hijacking vulnerabilities in Android by approximating app execution as a sequential permutation of "splits". We try to reduce the over-approximation and show that precise detection is feasible. Additionally, our handling of threads is more precise than CHEX as our analysis is object-sensitive. FlowDroid [5] is a state-of-the-art information flow analyser tailored for Android applications which we leverage upon and improve in the context of W2AI. AppSealer can automatically generate patches for Android apps with component hijacking vulnerabilities [38]. This work can potentially be used as a solution for injection attacks like W2AI attacks.

There are also dynamic analysis approaches [11,27,37]. TaintDroid uses taint analysis to track the flow of privacy sensitive data through third-party apps [11]. However, it requires a proper set of inputs to begin with. Our analysis generates the requisite inputs for W2AI attacks. Symbolic execution has been used to generate test inputs for Android apps. Cadar et. al. [7] generate event sequences based on concolic testing but does not address data inputs. Mirzaei et al. [28] perform symbolic execution for event sequences and data inputs by making an abstraction for modelling framework. However, their approach may not scale. Our refinement based approach is designed to reduce the state explosion problems in symbolic execution.

6 Conclusion

We present a comprehensive study on an underexplored class of W2AI attacks in Android. These attacks can be significant threats as they open a web-to-app attack channel without needing malware and can perform privileged operations. Our work is also novel in that unlike most analysis papers which are about finding potential vulnerabilities, we show that it is possible to automatically both detect and confirm vulnerabilities with an attack intent hyperlink (0-day web input) at scale on real apps.

References

1. Intents and intent filters. http://developer.android.com/guide/components/intents-filters.html
2. Java Debug Wire Protocol. http://developer.android.com/tools/debugging/index.html
3. Tencent Android SDK. http://wiki.open.qq.com/wiki/mobile/Android_SDK%E4%BD%BF%E7%94%A8%E8%AF%B4%E6%98%8E
4. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: CSF (2010)
5. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI (2014)
6. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In: SOAP (2012)
7. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Automated concolic testing of smartphone apps. In: FSE (2012)
8. Chen, Q.A., Qian, Z., Mao, Z.M.: Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In: USENIX Security (2014)
9. Chin, E., Wagner, D.: Bifocals: Analyzing WebView vulnerabilities in android applications. In: WISA (2014)
10. Davi, L., Dmitrienko, A., Sadeghi, A., Winandy, M.: Privilege escalation attacks on Android. In: ISC (2010)
11. Enck, W., Gilbert, P., Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX Security (2010)
12. Felt, A.P., Wagner, D.: Phishing on mobile devices. In: W2SP (2011)
13. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security (2011)
14. Georgiev, M., Jana, S., Shmatikov, V.: Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks . In: NDSS (2014)
15. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: PLDI (2005)
16. Grace, M.C., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: NDSS (2012)
17. Jin, X., Hu, X., Ying, K., Du, W., Yin, H., Peri, G.N.: Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In: CCS (2014)
18. Kaplan, D.: (cve-2014-3500/1/2) Apache Cordova for Android - multiple vulnerabilities. http://seclists.org/fulldisclosure/2014/Aug/21
19. King, J.C.: Symbolic execution and program testing. In: Commun. ACM (1976)
20. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. (1990)
21. Lam, P., Bodden, E., Hendren, L., Darmstadt, T.U.: The Soot framework for Java program analysis: a retrospective. In: CETUS (2011)
22. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using SPARK. In: CC (2003)
23. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV (2014)
24. Lin, C.C., Li, H., Zhou, X., Wang, X.: Screenmilker: How to milk your Android screen for secrets. In: NDSS (2014)
25. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: Statically vetting android apps for component hijacking vulnerabilities. In: CCS (2012)
26. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on WebView in the Android system. In: ACSAC (2011)

27. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: an input generation system for Android apps. In: FSE (2013)
28. Mirzaei, N., Malek, S., Păsăreanu, C.S., Esfahani, N., Mahmood, R.: Testing Android apps through symbolic execution. In: SIGSOFT Softw. Eng. Notes (2012)
29. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing Android sources and sinks. In: NDSS (2014)
30. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL (1995)
31. Schlegel, R., Zhang, K., Zhou, X.y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: NDSS (2011)
32. Schrittwieser, S., Frühwirt, P., Kieseberg, P., Leithner, M., Mulazzani, M., Huber, M., Weippl, E.R.: Guess who's texting you? evaluating the security of smartphone messaging applications. In: NDSS (2012)
33. Terada, T.: Whitepaper attacking android browsers via intent scheme urls. http://www.mbsd.jp/Whitepaper/IntentScheme.pdf
34. Wang, R., Xing, L., Wang, X., Chen, S.: Unauthorized origin crossing on mobile platforms: threats and mitigation. In: CCS (2013)
35. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In: CCS (2014)
36. Xing, L., Pan, X., Wang, R., Yuan, K., Wang, X.: Upgrading your Android, elevating my malware: Privilege escalation through mobile os updating. In: Security and Privacy (2014)
37. Yan, L.K., Yin, H.: DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: USENIX Security (2012)
38. Zhang, M., Yin, H.: AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In: NDSS (2014)
39. Zhou, X., Lee, Y., Zhang, N., Naveed, M., Wang, X.: The peril of fragmentation: Security hazards in android device driver customizations. In: Security and Privacy (2014)
40. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy (2012)
41. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in Android applications. In: NDSS (2013)
42. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In: NDSS (2012)

A Appendix

Case Studies from Table 2

In what follows, we detail the reached sinks which are exploitable and the damage caused by the vulnerabilities for each representative app in Table 2.

Abusing JavaScript-to-Native Bridge. WorkNet provides job information in Korea. This app enables settings for JavaScript and JavaScript-to-native interfaces in its configuration file (`config.xml`). We found vulnerabilities which exploit the `WebView.loadUrl` sink. This app enables the following settings:

`setJavaScriptEnabled`, `setGeolocationEnabled`, `setAllowFileAccess`, `setAllowFileAccessFromFileURLs`

Hence, the web attacker can mount all the attacks in the WebView interfaces abuse category on WorkNet. As explained in Sec. 2, its WebView loads arbitrary URLs which

Table 2: Representative vulnerable apps for each W2AI vulnerability category

ID	App	Version	Downloads
1	WorkNet (kr.go.keis.worknet)	3.1.0	1 - 5 M
2	Wikipedia (org.wikipedia)	1.3.4	10 - 50 M
3	WeCal Calendar (im.ecloud.ecalendar)	3.0.8	1 - 5 M
4	IPharmacy (com.sigmaphone.topmedfree)	1.0.92	1 - 5 M
5	i2X RDP (com.tux.client)	11.0.1899	1 - 5 M
6	Moneycontrol (com.divum.MoneyControl)	2.0	1 - 5 M
7	Caller ID (com.callapp.contacts)	1.56	1 - 5 M
8	Sina Weibo (com.sina.weibo)	4.3.0	5 - 10 M

ID: Category ID
App: Representative App (Package Name)
Version: App's Version
Download: # of Downloads (Million)

Table 3: Sinks and policies/settings for representative apps from Table 2

Category	Sub-category	Representative Sinks	Policies/Settings	ID
Abusing WebView Interfaces	Abusing JavaScript-to--Native Bridge	`WebView.loadUrl`	JavaScript-to-native interfaces, `setJavaScriptEnabled`	1
	Abusing HTML5 APIs	`WebView.loadUrl`	`setGeolocationEnabled`, `setJavaScriptEnabled`	2
	Local File Inclusion	`WebView.loadUrl`	`setAllowFileAccess`, `setJavaScriptEnabled`, `setAllowFileAccessFromFileURLs`	3
	Phishing	`WebView.loadUrl`	`setJavaScriptEnabled`	4
Abusing Native App Interfaces	App Database Pollution	`SQLiteDatabase.insert`	-	5
	Persistent Storage Pollution	`SharedPreferences.Editor.putString`	-	6
	Open Re-delegation	`Class.forName`	-	7
	App-Specific Logic Flaws	`TextView.setText`	-	8

exposes the Java native methods to the Javascript code. Once the user clicks the malicious link, WorkNet loads the URL from the intent hyperlink's parameters in the WebView. Therefore, the malicious page running in the WebView can invoke 21 JavaScript-to-native interfaces to access private user data (e.g., contacts) and perform privileged operations (e.g., modifying local files).

Abusing HTML5 APIs. Wikipedia is the free encyclopedia containing more than 32 M articles in 280 languages. It contains 2 paths that reaches the `WebView.loadUrl` sink and enables JavaScript and `geolocation` settings. The combination of this sink and setting enables the malicious site running in the WebView to access the GPS sensors and send out the user's current location to the attacker to track the user at any time.

Local File Inclusion. WeCal Calendar is a calendar assistant, which synchronizes with the Google calendar, takes notes, sets alarm and so on. W2AIScanner detects that the app has flows that reach the `WebView.loadUrl` sink and enables settings for JavaScript and the file's access. The settings are: `setAllowFileAccess`, `setAllowFileAccessFromFile-URLs`. After validation, we find that with loading the local HTML file (whose URL from the URI intent) in the WebView, the file can utilize `XMLHttpRequest` to read the local files (e.g., `/etc/hosts`) and leak the content to the attacker.

Phishing. IPharmacy provides medical products. W2AIScanner detects that the `Webvi-ew.loadUrl` sink in this app is reachable and exploitable. Therefore, this app can be exploited to load a phishing page whose URL is derived from the intent hyperlink from the web attacker in the customized WebView.

App Database Pollution. 2X RDP Client is a popular remote desktop app. The exploitable sink reported by W2AIScanner is `SQLiteDatabase.insert`, which adds items to `farms` table. The web attacker can set sensitive attributes, e.g., credentials, in the URI intent to pollute the app's database.

Persistent Storage Pollution. MoneyControl is a popular business and marketing app. W2AIScanner detects paths that inject data to the `SharedPreferences.Editor.putString` sink. Exploiting this vulnerability, the web attacker can make permanent changes to the storage.

Open Re-delegation. Caller ID - Call Blocker is a caller-ID app in Google Play that identifies unknown callers. The reached sink for this app is `Class.forName`. The attacker can set a private activity's name in the URI intent's parameters, this app will be launched and invoke the designated activity when the user clicks the malicious link.

App-Specific Logic Flaws. Sina Weibo is a microblogging client for Android phones. A W2AI vulnerability in this application allows the attacker to show arbitrary title messages to the victim user. The vulnerable sink in this application is `TextView.setText`. The attacker can launch an injection attack by putting an arbitrary title as query paramater in the malicious intent hyperlink.