Automated Generation of Event-Oriented Exploits in Android Hybrid Apps

Guangliang Yang, Jeff Huang, and Guofei Gu
Texas A&M University
{ygl, jeffhuang, guofei}@tamu.edu

***Abstract*—Recently more and more Android apps integrate the embedded browser, known as "WebView", to render web pages and run JavaScript code without leaving these apps. WebView provides a powerful feature that allows event handlers defined in the *native* context (i.e., Java in Android) to handle *web* events that occur in WebView. However, as shown in prior work, this feature suffers from remote attacks, which we generalize as Event-Oriented Exploit (EOE) in this paper, such that adversaries may remotely access local critical functionalities through event handlers in WebView without any permission or authentication.**

In this paper, we propose a novel approach, EOEDroid, which can automatically vet event handlers in a given hybrid app using selective symbolic execution and static analysis. If a vulnerability is found, EOEDroid also automatically generates exploit code to help developers and analysts verify the vulnerability. To support exploit code generation, we also systematically study web events, event handlers and their trigger constraints.

We evaluated our approach on 3,652 most popular apps. The result showed that our approach found 97 total vulnerabilities in 58 apps, including 2 cross-frame DOM manipulation, 53 phishing, 30 sensitive information leakage, 1 local resources access, and 11 Intent abuse vulnerabilities. We also found a potential backdoor in a high profile app that could be used to steal users' sensitive information, such as IMEI. Even though developers attempted to close it, EOEDroid found that adversaries were still able to exploit it by triggering two events together and feeding event handlers with well designed input.

I. Introduction

More and more Android apps leverage the power of the embedded browser, known as "WebView", to render web pages and run JavaScript code. In contrast to regular web browsers (such as desktop browsers), WebView is more powerful by providing a unique feature that allows event handlers defined in the *native* context (i.e., Java in Android) to handle *web* events that occur in WebView.

This powerful feature of WebView significantly enriches the functionalities of Android apps. However, as shown in prior work [19], [26], such a feature also introduces potential security flaws. More specially, it opens a bridge that links web code to native code, but the bridge is not protected in WebView.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23236
www.ndss-symposium.org

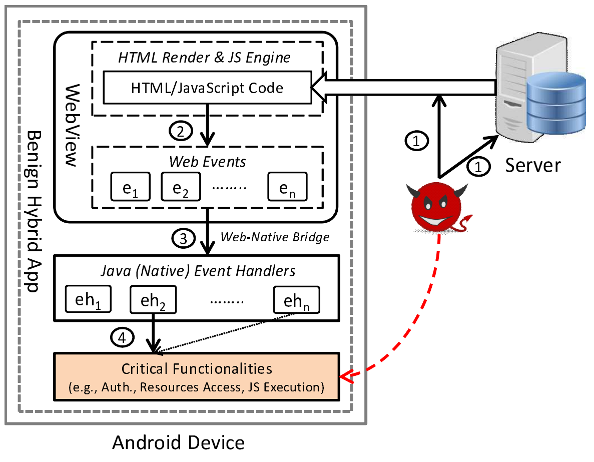


Figure 1: Attack Model

However, up to now it still remains unclear how adversaries involve the event handler feature in their attack vectors in practice. A possible attack scenario is that an adversary may trigger an event handler with appropriate input to leverage its internal *critical functionalities*. More details are shown in Figure 1. First, the adversary injects malicious HTML/JavaScript code into WebView through web or network attacks (Step 1). Then, the malicious code is executed and triggers a web event (Step 2). After that, the corresponding event handler in the native code is called (Step 3). Finally, the event handler is guided by the injected input to execute its internal critical functionalities (Step 4).

The above possibility is confirmed by our small-scale empirical study of 100 popular hybrid apps collected from Google Play. We found that an event handler in an old but still popular advertisement (ad) library, *"millennialmedia"* (version 5), contains rich and powerful functionalities, such as reading Android ID, recording audio and opening the camera. However, the access control on that event handler is weak. The internal critical functionalities can be utilized by triggering the associated web event and feeding it with appropriate input that follows the format "`mmsdk://c1.c2?args=...&callback=...`", where *c1* and *c2* are the native functions to be accessed, *args* are the function's parameters and *callback* is a JavaScript function name to receive the execution result of the native function.

In addition to the above scenario, another potential attack scenario is that a path to a critical functionality inside an event handler may be executed only under a specific program state, but such state may not be simply reached by only feeding that event handler with arbitrary input. Instead, similar to *return oriented programming* based attacks [32], it is possible for adversaries to play web events as *"gadgets"* and change an app's state. Assume the target program state is S_t. It may be reached through the transitions [$S_1 \rightarrow S_2 \rightarrow$ ... $\rightarrow S_t$], which could be achieved by triggering the sequence of web events [$E_1 \rightarrow E_2 \rightarrow$ ... $\rightarrow E_t$]. Hence, by following the above web event chain, adversaries can still change the program state to S_t and execute the target critical functionality.

For convenience, in this paper, we generalize all above attacks as *Event-Oriented Exploit* (EOE). Due to EOE's powerful capabilities to access critical functionalities through event handlers, serious consequences may be caused, such as local resource access, users' private data leakage and web cross-frame DOM manipulation.

Compared with existing attacks on Android (such as Trojan Attack [7]), EOE has multiple advantages. First, EOE does not require any extra permissions. The malicious web code injected by adversaries fully inherits the target apps' permissions. Second, EOE does not require malicious payloads. Instead, the functionalities contained in event handlers are utilized.

Furthermore, compared with existing attacks on WebView (such as sidewinder targeted attack [38], fracking attack [19], and code injection attack [24]), EOE is more practical and feasible. Existing attacks usually require JavaScript and JavaScript-bridge to be enabled, but EOE has no such requirements (Section V-A). Even only through HTML code and special HTTP(s) responses, adversaries can still trigger and leverage many event handlers, including the popular event handlers *shouldOverrideUrlLoading()*, which handles the URL navigation event.

The impact of EOE to smartphone security is serious considering the pervasive deployment of hybrid apps today. However, exiting techniques face significant challenges in detecting and verifying apps against EOE. Static analysis suffers from high false positives due to the lack of real data and context. In addition, the limitation of static analysis for handling Java reflection is exacerbated when the reflection operation is combined with *array-indexing type implicit flows*, which occur frequently when parsing the gadgets' inputs. Dynamic analysis may have low false positives, but is prone to low code coverage. Moreover, generating the required sequence of gadgets to reveal an EOE vulnerability is inherently challenging.

Our Approach. In this paper, we present a systematic study of EOE in Android hybrid apps together with a novel technique, `EOEDroid`, which can automatically analyze event handlers, detect exploitable critical functionalities, and further generate exploit code. EOEDroid can be applied to help developers detect and verify the EOE security issues before publishing their apps. The basic idea behind EOEDroid is that a critical functionality f can be leveraged by adversaries if there is a program state s that makes f's corresponding path p to be feasible. Since state s can be influenced or determined by all conditional statements $[c_0, c_1, ..., c_n]$ along p, if adversaries can affect the path selection of each c_i, the program may be executed along p. To cover each c_i, adversaries have two ways: (1) feeding the event handler with appropriate input, and (2) changing execution orders of event handlers. Our goal is to explore these two ways to cover every c_i to reach f. For convenience, we refer to the second case as *event handler dependency*, which is defined as follows. If operands of a condition c_0 in the path p_0 of an event handler eh_0 can be influenced by the path p_1 of another event handler eh_1, we say eh_0 depends on eh_1 on c_0 (i.e., $\langle eh_1, p_1 \rangle \xrightarrow{c_0} \langle eh_0, p_0 \rangle$). This means that if adversaries first guide the app to execute p_1, the program state related to c_0 may be influenced, and then, the expected branch behind c_0 may be taken.

The design of EOEDroid is depicted in Figure 2. Given a target app, EOEDroid first employs selective symbolic execution to analyze all its event handlers, *actively* explore all interesting paths and identify critical functionalities. The path constraints of each interesting path are collected for further analysis. A significant difference with existing symbolic execution based techniques is that EOEDroid carefully handles all conditional statements, including those whose associated operands are not symbolic (i.e., concrete or constant). This is because those conditional statements can provide hints to generate gadgets' execution orders.

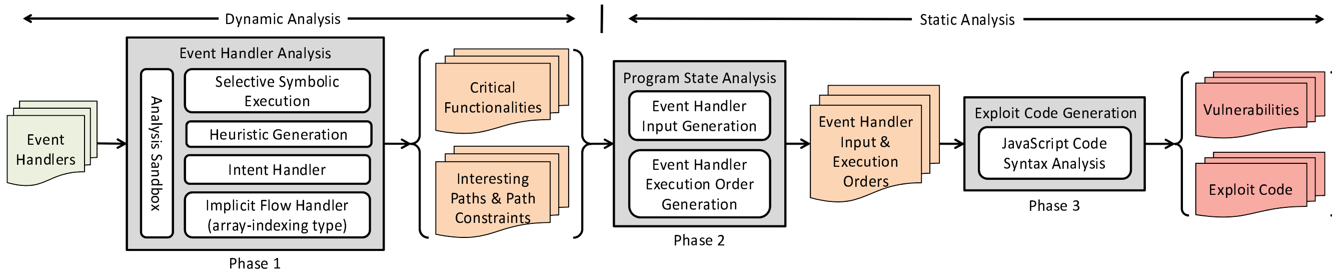
To mitigate the notorious "path explosion" problem in symbolic execution, we use several heuristics (e.g., scanning "interesting" APIs and instructions to discover interesting paths in Section V-B1). While these heuristics might cause over-approximation and/or inaccuracy to our analysis, they help us make a good tradeoff between performance and accuracy. In addition, we propose new solutions to address the analysis challenges raised by array-indexing type implicit flows as well as Android features and specifications such as unsupported *fork()* [1] and inter-component communication (e.g., Android Intent).

Based on the results of selective symbolic execution, EOEDroid then applies static analysis to discover program states that can lead to the execution of a critical functionality, and generates input and execution order of event handlers to reach the program state. The input of an event handler can be generated by solving its path constraints, and the execution order of event handlers can be constructed by solving the *event handler dependency* problem on those conditional statements whose operands are not symbolic.

Finally, EOEDroid generates exploit code by converting event handlers' input and execution orders to gadgets' (i.e., web events). If JavaScript code is required as gadgets' input, EOEDroid is also aware of its syntax and generates the required code.

Along with this, we conduct a systematic study of events, event handlers, and their triggering code and constraints in WebView. We find that 37 web events are exposed to adversaries, and the constraints on triggering events and event handlers are mainly caused by the status of JavaScript and the level of the web frame the malicious code is injected into. We also find that five event handlers have extra trigger constraints caused by predetermined execution orders of event handlers, and we identify 29 channels that can pass data from web code to native code.

Evaluation. We have implemented EOEDroid based on the Android framework and the Dalvik virtual machine (DVM), and evaluated it with 3,652 most popular apps collected from Google Play. EOEDroid found 97 total vulnerabilities in 58



Dynamic Analysis
Static Analysis
Event Handlers
Event Handler Analysis
Analysis Sandbox
Selective Symbolic Execution
Heuristic Generation
Intent Handler
Implicit Flow Handler (array-indexing type)
Phase 1
Critical Functionalities
Interesting Paths & Path Constraints
Program State Analysis
Event Handler Input Generation
Event Handler Execution Order Generation
Phase 2
Event Handler Input & Execution Orders
Exploit Code Generation
JavaScript Code Syntax Analysis
Phase 3
Vulnerabilities
Exploit Code

Figure 2: The Design of EOEDroid.

apps, including 2 cross-frame DOM manipulation, 53 phishing, 30 sensitive information leakage, 1 local resources access, and 11 Intent abuse vulnerabilities. We also found a potential backdoor in a high-profile app that may be used by adversaries to steal users' sensitive information, such as IMEI. Even though the developers of the app attempted to close the backdoor, EOEDroid found that adversaries were still able to exploit it by triggering two events together and feeding event handlers with appropriate inputs. We show more details in our case study in Section VI-C2 to illustrate this vulnerability.

We have reported all our findings to app developers, and are working with them to fix the vulnerabilities.

To sum up, we make the following contributions:

- We present a systematic study of Event-Oriented Exploits (EOE) in Android hybrid apps and a novel technique EOEDroid to automatically generate exploits that reveal security vulnerabilities.
- We thoroughly study events and event handlers as well as their triggering constraints in WebView.
- We evaluate EOEDroid using 3,652 hybrid apps. EOEDroid identified 97 vulnerabilities in 58 apps that can cause critical attacks.

II. Background: Android Apps, WebView, And Event Handlers

Android apps are typically written in Java and compiled to Dalvik bytecode [5]. At runtime, bytecodes are interpreted and executed by Dalvik virtual machine (DVM) [4]. Generally, an app consists of four components: activity (i.e., the user interface), background service, content providers (i.e., database), and Android native event receivers. Intent can be used in interactions among components and apps.

WebView is a small UI component in Android, which can be integrated into apps to display web pages. WebView can also execute JavaScript code if developers enable JavaScript in WebView's settings, which is disabled by default. WebView settings can also be used to disable local files, database, and GPS location access.

The event handler feature makes WebView more powerful. Usually, the function prototypes of event handlers are pre-defined by the Android system in the *native* (i.e., Java in Android) language. Hence, to implement an event handler, developers need to override the corresponding Java function, and then register the implementation in WebView. When the corresponding event is triggered in the *web* context, the event handler implemented by developers is called to handle it. For instance, developers can override the event handler *shouldOverrideUrlLoading()* to handle the URL navigation event. If an event handler is not implemented by developers, the default implementation in the Android system will be called.

WebView manages event handlers by either itself or *event handler classes*. An event handler class is a collection of event handlers. There are mainly two types of event handler classes. One is WebViewClient, which manages the event handlers that are relevant to URL navigation. The other is WebChromeClient, which manages the event handlers that are relevant to UI display, such as handling the alert dialog opened by JavaScript alert().

Through the API *loadUrl()*, WebView renders content in its UI component. The parameter format supported by *loadUrl()* is diverse. It can be a URL, a local HTML file, or JavaScript code. If the parameter is JavaScript code, 1) it must start with the special string *"javascript:"*, and 2) it is executed in the ***main*** web frame. For instance, the following code will popup an alert window to show current cookie in the main frame:

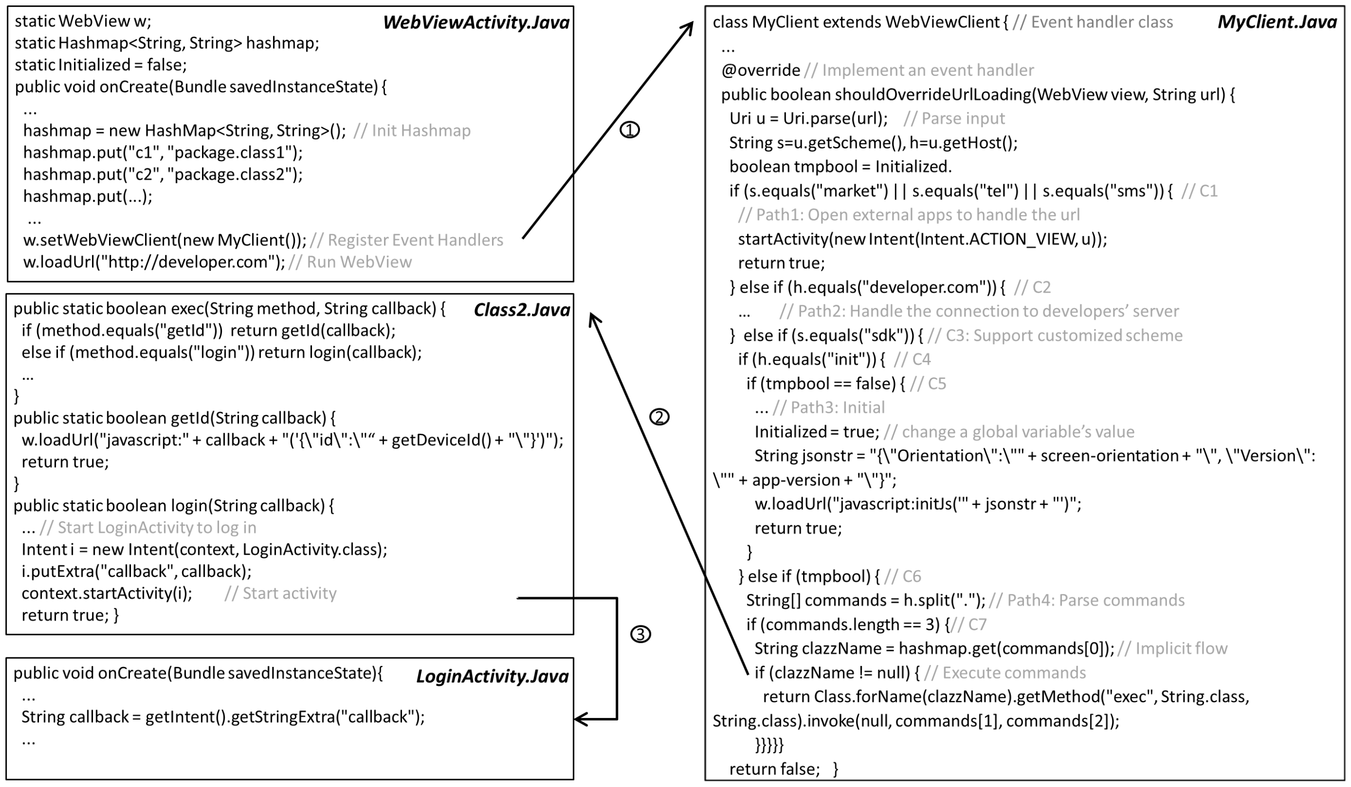
```
WebView.loadUrl("javascript:alert(document.cookie);").
```

III. Problem Statement

A. Motivating Example

To illustrate event-oriented exploits, we walk through a real-world vulnerable app with relevant code shown in Figure 3. In the activity *"WebViewActivity"*, the app initializes a webview component by a class *"MyClient"*, which implements an event handler *"shouldOverrideUrlLoading()"*. In the event handler, the input *url* is firstly parsed by a class *"URI"*, which is commonly used to analyze URI's syntax and extract useful information, such as URI's scheme and host. Then, the *url*'s content is analyzed, which determines the event handler's behaviors. If the *url*'s scheme is *"market"*, *"tel"*, or *"sms"*, the corresponding external apps (such as Google Play, default phone call app, or default text message app) will be opened to handle the input (`Path1`). If the *url*'s host is *"developer.com"* (which means WebView connects to a remote server), the event handler may approve the connection (`Path2`).

Meanwhile, the event handler implements supports for the customized scheme *"sdk"*. If the *url*'s host h is *"init"*, WebView executes JavaScript code to perform initialization (`Path3`). If h's format is "$c_0.c_1.c_2$", the app calls the Java method whose class name is determined by c_0, method name c_1, and execution result is transferred to the JavaScript method c_3 (`Path4`). Note that resolving the Java method relies on the content of the variable *"hashmap"*, which converts c_0 (i.e., *commands[0]*) to the real class name (i.e., *className*). Such an operation introduces an **implicit flow** from c_0 to *className*. *"Class2"* is one of classes whose methods can be invoked by the event



WebViewActivity.Java
```
static WebView w;
static Hashmap<String, String> hashmap;
static Initialized = false;
public void onCreate(Bundle savedInstanceState) {
 ...
 hashmap = new HashMap<String, String>(); // Init Hashmap
 hashmap.put("c1", "package.class1");
 hashmap.put("c2", "package.class2");
 hashmap.put(...);
  ...
 w.setWebViewClient(new MyClient()); // Register Event Handlers
 w.loadUrl("http://developer.com"); // Run WebView
```

Class2.Java
```
public static boolean exec(String method, String callback) {
 if (method.equals("getId"))  return getId(callback);
 else if (method.equals("login")) return login(callback);
 ...
}
public static boolean getId(String callback) {
 w.loadUrl("javascript:" + callback + "('{\"id\":\"“ + getDeviceId() + "\"}')");
 return true;
}
public static boolean login(String callback) {
 ... // Start LoginActivity to log in
 Intent i = new Intent(context, LoginActivity.class);
 i.putExtra("callback", callback);
 context.startActivity(i);      // Start activity
 return true; }
```

LoginActivity.Java
```
public void onCreate(Bundle savedInstanceState){
 ...
 String callback = getIntent().getStringExtra("callback");
 ...
```

MyClient.Java
```
class MyClient extends WebViewClient { // Event handler class
 ...
 @override // Implement an event handler
 public boolean shouldOverrideUrlLoading(WebView view, String url) {
  Uri u = Uri.parse(url);   // Parse input
  String s=u.getScheme(), h=u.getHost();
  boolean tmpbool = Initialized.
  if (s.equals("market") || s.equals("tel") || s.equals("sms")) {  // C1
   // Path1: Open external apps to handle the url
   startActivity(new Intent(Intent.ACTION_VIEW, u));
   return true;
  } else if (h.equals("developer.com")) {  // C2
   ...     // Path2: Handle the connection to developers' server
  }  else if (s.equals("sdk")) { // C3: Support customized scheme
   if (h.equals("init")) {  // C4
    if (tmpbool == false) { // C5
     ... // Path3: Initial
     Initialized = true; // change a global variable's value
     String jsonstr = "{\"Orientation\":\"" + screen-orientation + "\", \"Version\":
\"" + app-version + "\"}";
     w.loadUrl("javascript:initJs('" + jsonstr + "')";
     return true;
    }
   } else if (tmpbool) { // C6
    String[] commands = h.split("."); // Path4: Parse commands
    if (commands.length == 3) {// C7
     String clazzName = hashmap.get(commands[0]); // Implicit flow
     if (clazzName != null) { // Execute commands
      return Class.forName(clazzName).getMethod("exec", String.class,
String.class).invoke(null, commands[1], commands[2]);
     }}}}}
  return false;   }
```

Figure 3: Vulnerable Code From A Real-World App.

handler. In its method *getId()*, the device ID is transferred to the web space. In its method *login()*, the activity *"LoginActivity"* is started through an Intent message to ask users to login. In the Intent message, part of the *url*'s content (i.e., c_2) is contained and passed to the message receiver.

Note that in the example app there is a critical function: *getDeviceId()* in the method *getId()* of *Class2*. Adversaries cannot directly utilize this functionality, because the operand *tmpbool* is false (i.e., the conditional statement C_6). However, by manipulating gadgets, adversaries may change the program state (such as *tmpbool*'s value) and drive the app to call *getId()*. In *getId()* a JavaScript function is also required as part of the event handler's input to receive the device ID.

Although it appears simple to manually analyze this example code, real-world apps are much more complex. Our goal is to develop a technique that can automatically detect such vulnerabilities and construct exploit code.

B. Threat Model

We assume that WebView is enabled in apps, but JavaScript is not required to be enabled, since HTML code can also trigger event handlers. We assume that adversaries can inject malicious HTML/JavaScript code into WebView. As Figure 1 shows, we consider the following two different attack scenarios:

- *Web Attack*: In this scenario, we assume adversaries control several malicious domains and servers, but they are not able to control or monitor the network traffic between apps and other domains.
  The web content loaded from first-parties is trustable. However, the content may further contain subframes (e.g., iframe) to load extra web content from third-parties, which may be malicious.
  Generally, all web frames loaded in WebView are well isolated and protected by same origin policy (SOP) [8].
- *Network Attack*: Adversaries can hijack unsafe network traffic (such as HTTP) through man-in-the-middle attacks. Compared with desktop programs, mobile apps are more likely to suffer from this type of attacks, considering that many unsafe WI-FI hotspots are used [23].

Note that we do not assume any other abilities of the adversaries. They may not access the users' device, install any certificate or malware, or change apps' internal data. The target app itself as well as all the apps pre-installed on the users' devices may be benign.

C. Security Issues

Similar to other attacks on WebView [19], the security issues caused by event handlers are rooted in the inconsistency of security models between web and native context. In hybrid apps, the SOP security model for the web context is circumscribed to prevent event handlers from being triggered by malicious web code, because the handlers do not have any way of identifying the origin of an event (so they have no way to distinguish between trusted and untrusted origins). SOP is also ineffective to protect the local resources (such as camera), which are located in the native context. The permission based sandbox model for the native context can protect local resources. However, it is ineffective to prevent the access to critical functionalities from web code, since the origin information of the access is lost.

D. Problem Definition

We state that an exploit is successful if it successfully triggers a critical functionality through event handlers defined in the app. A successful exploit must satisfy the constraints in triggering target events and event handlers: it must guide the target app to reach the target state by manipulating the input and execution orders of gadgets, and it must bypass all security checks which are usually located before the critical functionality.

The event-oriented exploit generation problem can be formally defined as follows. Given an app, discover a program state s that leads the app to execute a critical functionality. Such a state should be reached through a sequence of executions of gadgets ((W_0, E_0, I_0, J_0), (W_1, E_1, I_1, J_1), $\ldots$, (W_n, E_n, I_n, J_0)), where W_i is the HTML/JavaScript code that triggers the event E_i and passes the input I_i to E_i. I_i may also include pre-defined JavaScript code J_i.

E. Critical Functionalities

We define critical functionalities as sensitive APIs in the Android framework. In this paper, we mainly consider the following four types of APIs. Nevertheless, EOEDroid is extensible and user customized APIs can be added easily.

URL Loading API (e.g., *WebView.loadUrl(p)*). If malicious HTML/JavaScript code in ***subframes*** leverages the API through EOE, the content of the *main* frame or the whole WebView may be changed (Section II). Depending on the value of the API parameter p, the following two consequences may be caused.

- *Cross-Frame DOM Manipulation*: If the web code in subframes influences p's value and makes p be starting with "javascript:", the JavaScript code contained in p may be executed in the main frame. Hence, through EOE, the web code in subframes obtains the capability to bypass SOP and inject malicious code to the main frame.
- *Phishing*: If the web code in subframes can determines p's value through EOE, it may change p's value to the url of a fake web page. Then, WebView is redirected to show the fake web page. Considering that WebView usually does not have an address bar to indicate the url it is loading, such attacks on WebView are much more stealthy than on regular web browsers.

Compared with other attack channels (such as MITM attacks) which may also be utilized to perform above attacks, EOE over *loadUrl()* is more powerful. Considering the situation that WebView loads a webpage from developers' web site using HTTPS, and one of its nested subframes uses HTTP. Due to boundaries between frames, existing attacks may only be able to control the content of the subframe, but not the main frame. However, EOE does not have this limitation. By means of *loadUrl()*, adversaries can directly change the content of the main frame.

Source and Sink APIs. This type of API invocations may result in users' privacy leakage. We mainly consider two scenarios: (1) there are paths from source to sink in event handlers. (2) source is passed to the web space, and then sent out through HTML/JavaScript code.

We consider the Android ID, device ID, phone number, and serial number, and GPS location information as *source*, and connecting network and sending text message as *sink*.

APIs Accessing Local Resources. This type of APIs may be leveraged by adversaries to access local resources, such as local files, and hardware resource (e.g., camera). Serious consequences may be caused when these APIs are combined with other sensitive APIs. For instance, adversaries may remotely take a picture and also save it to the local storage using camera APIs. Then, adversaries may obtain the picture in the web context through file reading API and further send the picture out through native sink APIs or HTML/JavaScript code.

APIs Sending Intent messages. As demonstrated by Wang et al. [36], the Intent messages that are sent out through WebView may have serious consequences. We consider the following type of APIs as sensitive: the API parameter is totally controlled by adversaries, which means the destination of the Intent message to be sent is totally determined by adversaries. For other Intent-sending APIs, we treat them as regular inter-component communications.

IV. System Overview

In this section, we provide an overview of EOEDroid and illustrate it with the motivating example described in the previous section. The technical details of EOEDroid are presented in Section V.

We use the following basic concepts and notations:

- *A Symbolic Conditional Statement*: a conditional statement whose operands are symbolic.
- *Path Constraints*: all constraints that must be satisfied when guiding an app to execute a path. Different from prior work, EOEDroid involves both symbolic and non-symbolic conditional statements in path constraints.
- *Input Constraints*: A subset of path constraints but are only related to event handlers' input.

We assume that s is the target program state that leads to the execution of a critical functionality; f is the target critical functionality; p_0 is the path containing f; eh_0 is the event handler containing p_0.

A. Overview

EOEDroid consists of three modules: *event handler analysis*, *program state analysis*, and *exploit code generation*, as shown in Figure 2. In the first module, *selective symbolic execution* is used to explore paths in the event handlers and collect path constraints. To apply the technique for Android hybrid apps, technical challenges (Section V-B) are addressed by four sub-modules: *analysis sandbox*, *heuristic generation*, *Intent handler*, and *array-indexing type implicit flow handler*. More specifically, given an app, *"selective symbolic execution"* is called to repeatedly test each event handler until all the inside *interesting* paths are traversed. The interesting paths are discovered by the sub-module *"heuristic-generation"*. Note that when a branch is flagged as interesting, no matter whether the conditional statement is symbolic or not, EOEDroid *forcely* traverses this path. Meanwhile, the corresponding path constraint is constructed and saved.

For each round of test, the sub-module *"analysis sandbox"* is applied to guard the analysis environment from pollution and keep each round of test independent.

In the second phase, the module *"program state analysis"* runs to discover state s and learn how to reach s by manipulating event handlers' input and execution order, which are handled by the sub-modules *"event handler input generation"* and *"event handler execution order generation"* respectively. For event handlers' input, it is generated by applying an SMT solver in the associated input constraints collected in the first phase. For event handlers' execution order, it is generated by solving the event handler dependency problem (as described in Section I).

For each path p that contains critical functionalities, EOEDroid repeatedly resolves all event handler dependences for p with four steps: (1) it analyzes p's path constraints to identify all non-symbolic conditional statements; (2) it confirms the expected value v for each conditional statement; (3) starting from each conditional statement c, it performs backward program analysis to determine the variables O that can influence c's operands, and further computes the required value for each variable in O; and (4) it analyzes all paths in all event handlers that contain the instructions changing the variables in O to their corresponding expected values.

In the third phase, the module *exploit code generation* generates exploit code for each exploitable critical functionality. First, the event handlers' execution order generated in the second phase is converted to the web event order, and the event handlers' input is converted to the corresponding web events'. Second, if JavaScript code is required as the event handler's input (such as the callback function in our motivating example), the syntax of the associated JavaScript code is parsed and analyzed to generated required JavaScript code.

B. Analyzing the Example

Now we illustrate how EOEDroid works for our motivating example. When the event handler *shouldOverrideUrlLoading()* is triggered, EOEDroid is started. First of all, EOEDroid symbolizes the event handler's second parameter as *'InputUrl'*, since its value can be controlled by adversaries. Then, EOEDroid analyzes each instruction. As the class *Uri* is frequently used, we model it by symbolizing its instance u as *'Uri.<init>(Input Url)'*. The input's scheme and host are also symbolized, whose symbolic expressions are *'Uri.<init>(InputUrl).getScheme()'* and *'Uri.<init>(InputUrl).getHost()'*, respectively.

When the conditional statement $C1$ is analyzed, *"heuristic generation"* is started to discover which branches are interesting. In this case, both branches have interesting instructions. So both of them are sequentially traversed. In the true branch, when an Intent message is sent to another app or component, the module *"Intent handler"* (Section V-B3) is set up to fill the symbolic information gap between the sender and receiver.

Similarly, the conditional statements $C2$, $C3$ and $C4$ are processed. In $C4$'s true branch, EOEDroid encounters a special conditional statement that is non-symbolic (i.e., $C5$). As its true branch is interesting, EOEDroid forcely executes it and also collects necessary information, such as the executed path information, the instruction's position (such as <*MyClient.java*, *C5*>), the condition expression (i.e., *tmpbool == 0*), the operand variable (i.e., *tmpbool*), current value of the variable (i.e., 0) and its selected branch (i.e., 1). Note that in this path the external field variable *Initialized* is written. To ensure each round of test is independent, such interaction between the event handler and the external variable is handled by the sub-module *"analysis sandbox"*.

The conditional statement $C6$ is then reached. In the true branch, the host name is split to an array, whose symbolic expression is *Uri.<init>(InputUrl).getHost().split(".")*. Then, an implicit flow is faced, which is caused by the Hashmap accessing operation. To handle it, the sub-module *"implicit flow handler"* is started to try all possibilities in the Hashmap instance. Therefore, a critical functionality is found in *getId()* in *Class2*, which can be leveraged by adversaries to perform cross-frame DOM manipulation and steal the device ID information. The main associated path constraints are shown in Listing 1.

```
(1) Uri.<init>(InputUrl).getScheme().equals("market") == 0
(2) Uri.<init>(InputUrl).getScheme().equals("tel") == 0
(3) Uri.<init>(InputUrl).getScheme().equals("sms") == 0
(4) Uri.<init>(InputUrl).getHost().equals("developer.com")
     == 0
(5) Uri.<init>(InputUrl).getScheme().equals("sdk") ≠ 0
(6) tmpbool ≠ 0
(7) Uri.<init>(InputUrl).getHost().split(".").length == 3
(8) Uri.<init>(InputUrl).getHost().split(".")[0].equals("c2
     ") ≠ 0 // generated by implicit flow handler
(9) Uri.<init>(InputUrl).getHost().split(".")[1].equals("
     getId") ≠ 0
```

Listing 1: Path Constraints In Executing *getId()*

In the second phase, the module *"program state analysis"* analyzes the path constraints (Listing 1) to change the program state. First, the sub-module *"event handler input generation"* checks if the constraints can be satisfied by feeding the event handler with appropriate input. In this case, all constraints except (6) can be satisfied. Second, the sub-module *"event handler execution order generation"* runs to check how to influence the program state to satisfy the constraint (6). Starting from the conditional statement *C6*, EOEDroid backward tracks the operand *tmpbool* along the executed path, and confirms the variable (i.e., *Initialized*) can influence its value. Next, EOEDroid goes through all paths identified in the first phase to check whether there is a path that contains an instruction changing *Initialized*'s value. It finds that `Path3` contains the expected operation. Hence, there is an event handler dependency on *C6*: <*shouldOverrideUrlLoading(), Path3*> $\xrightarrow{C6}$ <*shouldOverrideUrlLoading(), Path4*>.

In the third phase, the module *"exploit code generation"* generates the exploit code for the critical functionality in *getId()*. To drive the app to execute the critical functionality, event handlers should be executed as follows:

```
(1) shouldOverrideUrlLoading(webview, "sdk://init")
(2) shouldOverrideUrlLoading(webview, "sdk://c2.getId.?")
```

Then, the above event handler execution order is converted to the web event order, and further transformed to the following HTML/JavaScript code (based on our event handler study presented in Section V-A):

```
<iframe src="sdk://init"/>
<iframe src="sdk://c2.getId.?"/>
```

The above code can change the program state and reach the sensitive API *loadUrl()*. However, part of the event handler's input is still missing, which is a JavaScript callback function used to receive the sensitive information (i.e., device ID). To address this problem, the sub-module *"JavaScript code syntax analysis"* runs to analyze the syntax of the parameter of *loadUrl()*, and generate required JavaScript code. Finally, the following exploit code is generated, which can help developers test and verify the EOE problem.

```
<script>
function steal_device_id(id) {
  document.write("<" + "img src='" + "http://attacker.com/"
          + id + "' />")
}
</script>
<iframe src="sdk://init"/>
<iframe src="sdk://c2.getId.steal_device_id"/>
```

Listing 2: Exploit Code

V. Technical Approaches

In this section, we first present our study of events and event handlers in WebView to understand their constraints for triggering event-oriented exploits. We then present technical details about the design and implementation of selective symbolic execution, program state analysis, and exploit code generation.

A. Understanding Event Handler Triggering Constraints

The official Android documentation of events and event handlers is obscure and incomplete. We hence conduct a systematic study based on both reading documents about WebView on the web and analyzing real-world hybrid apps. The main study result is shown in Table I. We find that 37 events are available for adversaries in WebView. The triggering code for each event is shown in the fifth column. Note that the DOM element '*<iframe src=...>*' can directly trigger two events, whose corresponding event handlers are *shouldOverrideUrlLoading()* and *shouldInterceptRequest()*, respectively. It depends on the attribute src's content s. If s's scheme is not 'HTTP' and 'HTTPS', but customized, the former one is triggered. Otherwise, the latter one is triggered.

As reported in the third column, using HTML code, adversaries can trigger 15 event handlers, including popular event handlers *shouldOverrideUrlLoading()* and *shouldInterceptRequest()*. Note that, four of them require supports from the web server side to get appropriate HTTP response code. For instance, *onReceivedLoginRequest(webview, realm, account, args)* can be triggered by the combination of the HTML code *"<iframe src="http://attacker.com/login">"* and the HTTP response header *"x-auto-login:realm=x&account=y&args=z"*, which is from the malicious server "attacker.com". x, y, and z are passed to *onReceivedLoginRequest()* as function parameters. As the above example shows, adversaries can pass data from the web context to the native context. In our study, we find that the parameters of 29 event handlers can be influenced by adversaries. More details are shown in the first column (i.e., the parameters between parentheses).

As reported in the fourth column, triggering event handlers are influenced by the level of web frames where the events occur. We find that events which occur in the main frame could trigger all event handlers, whereas the capability of events in subframes is limited. More specifically, three event handlers cannot be triggered by events that occur in the main frame. Let E_i ($i \geq 0$) be the events that occur in the *ith* level web frame and can be handled by event handlers, E denotes all events available in the whole WebView space and E_0 denotes the events available in the main frame, they have the following relationship: $E_0 = E$ while $E_i \sqsubset E$ ($i > 0$).

Event Triggering Constraints. The constraints for triggering events are mainly caused by the status of JavaScript. As reported in Table I, almost 60% of the event handlers require JavaScript enabled to trigger.

Event Handler Triggering Constraints. The constraints for triggering event handlers are mainly from two aspects:

I: *The frame level*. Triggering three of the event handlers require their corresponding events to occur in the main frame. Adversaries must inject malicious code into the main frame, which is usually well protected. Also, it is easy for users to realize the injected web code, because it may reload web pages.

II: *Predetermined execution orders*. Several event handlers' execution orders are predetermined in WebView, which also imposes constraints on triggering the event handlers. To understand these predetermined execution orders, we create an experimental app which registers all event handlers, and profile them when they are invoked. Then, the app loads fuzzing HTML/JavaScript code. We also apply static analysis to track the return values of all event handlers. If an event handler's return value appears in a conditional statement, and later another event handler is called, a predetermined event order may exist. Finally, we confirm five predetermined execution orders:

1. *shouldInterceptRequest()* $\rightarrow$ *onLoadResource()*: The latter event handler is called only when the former event handler returns null.

2. *shouldOverrideKeyEvent()* $\rightarrow$ *onUnhandledKeyEvent()*: The latter event handler is only called when the former event handler returns false.

3. *onPageStarted()* $\rightarrow$ ... $\rightarrow$ *onPageFinished()*: When WebView starts loading a web page, *onPageStarted()* is called. When WebView finishes loading the page, *onPageFinished()* is called. During the process, other event handlers may be called as well, such as *onReceivedError()* and *shouldInterceptRequest()*.

4. *onPageStarted()* can be called multiple times before *onPageFinished()* is called. This happens when there are URL redirections in the web server side (i.e., 3xx HTTP response code). The number of times that $onPageStarted()$ is called depends on the URL redirection number. Moreover, generally, *onPageFinished()* is only called once, no matter how many URL redirections there are. But if the last HTTP response code is 4xx, WebView may be redirected to show a page-not-found HTML, and then, *onPageFinished()* is called again.

5. *onGeolocationPermissionsShowPrompt()* $\rightarrow$ *onGeolocationPermissionsHidePrompt()* and *onShowCustomView()* $\rightarrow$ *onHideCustomView()*: When location permission is requested, or Full Screen is entered, these events are called sequentially.

Adversaries' Capability: Playing Gadgets. Adversaries can change program states by manipulating gadgets' input and execution orders. More specifically, adversaries can pass data to web events, and then the data are passed to the corresponding event handlers as their function parameters. Adversaries can also trigger events and event handlers in arbitrary orders, even though there are constraints on triggering events and event handlers.

Gadgets' Input. Adversaries may be able to control event handlers' parameters. For example, *shouldInterceptRequest()*'s parameter (i.e., request) can be set as *''https://attacker.com/img''*, if adversaries use the HTML code "*<iframe src="https://attacker.com/img"></iframe>*" to trigger the event handler.

Event Handlers and Main Parameters	Handled Events	JS?	E$_0$?	Example Trigger Code (HTML/JavaScript/HTTP)
onFormResubmission	Resubmitting a form	✓		[HTML] <form ...> [JS] form.resubmit()
onPageCommitVisible(url)				[HTML] <body bgcolor="#0f0" ..> <img bgcolor="#0f0" ..
doUpdateVisitedHistory(url, isReloaded)	Updating history	✓	✓	[JS] document.location="url"
onPageStarted(url, icon)	Starting to load a page	✓	✓	[JS] document.location="url", reload()
onPageFinished(url)	finishing loading a page	✓	✓	Trigger Constraint
onReceivedError(errorcode, description, url)	Failing to load a page			[HTML] <iframe src="http://invalid.url" ...
onReceivedSslError(error)	SSL error			[HTML] <iframe src="https://invalid.url"...
onReceivedClientCertRequest(request)	Client cert request			[HTTP] Send client cert request
onReceivedHttpAuthRequest (host, realm)	Authentication request			[HTTP] Send authorization header
onReceivedHttpError(request, response)	HTTP error			[HTTP] Send 404 header
onReceivedLoginRequest(realm, account, arg)	Login request			[HTTP] Send x-auto-login header
onScaleChanged(old_scale, new_scale)	Updating scale	✓		[JS] document.body.style.zoom=...
shouldOverrideKeyEvent(keyevent)	Pressing key	✓		[JS] dispatch key-press event
onUnhandledKeyEvent(keyevent)	Facing an unhandled key	✓		Trigger Constraint
shouldInterceptRequest(request)	Resources loading			[HTML] <img src="... >, <iframe src="http://... >
onLoadResource(url)	Loading a resource			Trigger Constraint
shouldOverrideUrlLoading(url [or request])	URL navigation			[HTML] <iframe src="customizedScheme://...>
onCreateWindow	Creating a window	✓		[JS] window.open()
onCloseWindow	Closing a window	✓		[JS] window.close()
onConsoleMessage(message)	Printing messages	✓		[JS] console.log()
onGeolocationPermissionsShowPrompt (origin)	GPS request	✓		[JS] navigator.geolocation.getCurrentPosition()
onGeolocationPermissionsHidePrompt		✓		Trigger Constraint
onShowCustomView	Entering full screen	✓		[HTML] <video ... controls>[JS] webkitRequestFullScreen()
onHideCustomView	Quitting full screen	✓		Trigger Constraint
onJsBeforeUnload(url, message, result)	Leaving a webpage	✓		[JS] dispatch onbeforeunload event
onJsAlert(url, message, result)	Popuping an alert box	✓		[JS] alert()
onJsConfirm(url, message, result)	Popuping a confirm box	✓		[JS] confirm()
onJsPrompt(url, message, defaultValue, result)	Popuping a prompt box	✓		[JS] prompt()
onPermissionRequest(request)	Permission request	✓		[JS] navigator.getUserMedia()
onPermissionRequestCanceled(request)	Request is cancelled	✓		
onRequestFocus	Requesting focus	✓		[HTML] <input type="text" id="name" … [JS] focus();
onShowFileChooser	Browsing file system	✓		[HTML] <input type="file" … [JS] dispatch a click event
onProgressChanged(progress)	Page loading status			
onReceivedIcon(icon)	Receiving a icon			
onReceivedTitle(title)	Receiving a title			
onReceivedTouchIcon(url, precomposed)	Receiving an apple touch icon			
onDownloadStart(url, userAgent, contentDisposition, mimetype, contentLength)	Downloading a file			[HTML] <iframe src="http://url.apk" ...

Table I: The Systematic Study Result. The third column 'JS?' means: 'Is JavaScript required to trigger the event?', and the forth column 'E$_0$?' means: 'Does the event handler only deal with events from E_0?'. In answers, we use ✓and blank to indicate 'Yes' and 'No', respectively.

Gadgets' Execution Orders. Consider two event handlers eh_1 and eh_2, there are two cases to analyze: (1) If eh_1 and eh_2 do not have any relationship, adversaries can call them in any order (i.e., $eh_1 \rightarrow e_2$ and $eh_2 \rightarrow eh_1$); (2) If eh_1 must be executed before eh_2, their relationship should be $\xrightarrow{t} eh_1 \xrightarrow{c} eh_2$, where t is the trigger code to call eh_1 and c is the pre-condition that must be satisfied to trigger eh_2. By repeating t and make c be satisfied, we may get the event handler sequence $(eh_1 eh_2 eh_1 eh_2)$, which includes expected sequences (both $eh_1 \rightarrow eh_2$ and $eh_2 \rightarrow eh_1$).

B. Selective Symbolic Execution

To apply symbolic execution in event handlers, we address four challenges (with details in following subsections):

- *Path explosion*: To address this notorious problem, EOEDroid uses static analysis to provide heuristic information for path selection. However, as discussed in Section I, static analysis may introduce false negatives to the heuristic information. To avoid it, we conservatively and safely apply static analysis on only a certain number of instructions that do not cause false negatives (Section V-B1).
- *Unsupported fork()*: In existing dynamic symbolic execution based approaches, *fork()* is frequently used to help systems traverse branches and keep the analysis environment clean. However, in Android, *fork()* is not supported. Instead, EOEDroid needs to sequentially traverse branches. However, different with desktop software, it is expensive to save and restore states of Android apps. To fix the problem, we propose an *analysis sandbox* to handle the interaction with the external environment (Section V-B2).
- *Android Intent*: Intent is frequently used in event handlers, such as triggering a GUI event to open a GUI activity. However, it introduces semantic gap between Intent senders and receivers. Figure 4 shows an example that an intent message is delivered between two apps. The Intent message escapes from the Java

context (i.e., DVM), enters the C/C++ context (i.e., Linux kernel), and finally returns to the Java context. This way raises challenges to track the Intent message in the Java context. When the receiver obtains the message, the associated symbolic information may be lost. To address the problem, we fill the gap between senders and receivers by synchronizing the symbol information in both sides (Section V-B3).

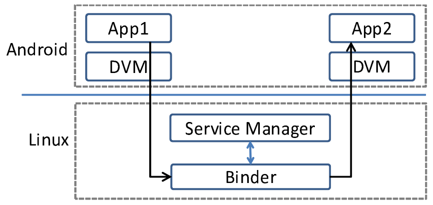


Figure 4: Intent In Inter-Apps Communications

- *Array-indexing class implicit flows*: In array-indexing type operations, if the index is symbolic, it is challenging to determine which element should be returned. The problem is known as "implicit flow". Similar problems also exist in other data structures such as Hashmap, Android Bundle, and Android share preference. In real world, this type of operations and data structures was frequently used in popular apps and ad libs, such as Google Ads.
  To further demonstrate the problem, we use Hashmap as the example. As Figure 5 shows, in Java, Hashmap is implemented based on a bucket array with linked lists that are used to handle hashing collisions. Assume that the instruction *v = M.get(k)* is being executed, where M is the Hashmap object, k is the key and it is symbolized as 'key'. In the function *Hashmap.get()*, the bucket index is firstly determined, which is k's hashcode. Hence, the index is a symbolic expression built on key. Then, an *array-indexing operation* is done to obtain the associated linked-list. Since the index is symbolic, the operation introduces an implicit flow.
  To mitigate the problem, we instrument k to brute-forcely try all possibilities of keys (Section V-B4).

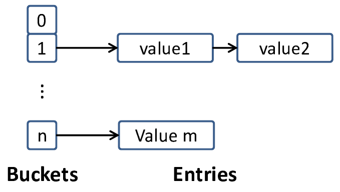


Figure 5: The Internal Structure Of HashMap

We implement selective symbolic execution by instrumenting the Android framework and Dalvik virtual machine (DVM). In Android frameworks, event handler functions and sensitive APIs (Section III-E) are handled. In DVM, the mapping between variables and their corresponding symbolic expressions are managed through a global symbolic table. To support string operations, which are frequently faced in event handlers, the associated string APIs are modeled, including compare, append, replace, search, substring and split, and we use Z3-Str [42] to resolve string based path constraints.

1) Heuristic Generation: The heuristic information used in path selection includes the indication of whether a branch is interesting. To determine it, EOEDroid uses static analysis to scan a certain number (such as 100) of instructions in advance to check if a critical functionality is contained.

Due to the imprecision of static analysis, false negatives may be introduced (Section I). The determination result may be further influenced. To eliminate the concern, we also flag the following types of operations as *interesting*.

- *Field variables reading and writing*: This affects points-to and alias relationship.
- *Virtual function invocation*: Resolving this kind of invocations requires points-to information.
- *Java Reflection*: Due to the lack of real data, it is challenging for static analysis to solve this kind of problems.
- *Return Instruction*: In event handlers, the returned values of some event handlers (Section V-A) are meaningful, such as *shouldOverrideUrlLoading()* and *shouldInterceptRequest()*. Take the former event handler as the example: If the event handler returns true, it means the app being analyzed handles the input. Otherwise, the Android system processes the input.

2) Analysis Sandbox: To keep the analysis environment clean, EOEDroid creates a sandbox environment to replace the real environment. All interactions with the external real environment is redirected to the sandbox environment. Based on the access direction, the interactions can be divided into two categories: writing and reading. For the writing operation, EOEDroid updates variables' values in the sandbox instead of the real environment. For the reading operation, if the destination variable is written earlier, the corresponding value in the sandbox is retrieved and returned; otherwise, the value in the real environment is returned.

In this paper, we consider the interactions include accessing file system, global variables, and field variables whose scopes are bigger than the event handler function being analyzed. To implement them, necessary APIs and instructions are hooked and handled. For reading and writing files, the corresponding POSIX APIs (in libcore\io\Posix.java) are handled. However, it is challenging to maintain a file's status, especially when the file is partially modified. To mitigate the problem, a backup file is created, and then all reading and writing operations are redirected to the backup file. For reading and writing global and field variables, the associated instructions (i.e., iget/iput, aget/aput, and sget/sput) [5] are handled. In practice, it is challenging to determine the scope of a field variable. To simplify the problem, all changes on the field variable are recorded. Please note that in the beginning of each round of test, all data and files saved in the sandbox are cleaned.

3) Intent Handler: To fill the symbolic information gap between Intent message senders and receivers, it is critical to restore symbolic information of the message in the receiver side. For this purpose, when the Intent message is sent, EOEDroid temporally pauses the program by hooking the associated APIs (such as *startActivity(Intent)*), makes snapshot on the Intent object and its corresponding symbolic data, and also saves it. Then, when the receiver accepts and reads the message using associated APIs (such as *getIntent()*), the snapshot is read, and then the symbolic information is linked with the Intent object. Considering the sender and receiver may be not in the same app, such a snapshot is dumped to a public folder, which is allowed to be accessed by any app.

As variables' absolute memory addresses are used to save their symbolic information in the snapshot, in the receiver side the restored symbolic information cannot be directly applied in the received Intent message, whose memory addresses are totally different from the sent message. To correct the differences, when the snapshot is made in the sender side, memory addresses are changed to relative addresses, based on the starting address of the sent message. Then, when the snapshot is read in the receiver side, memory addresses are changed back to the absolute addresses, based on the starting address of the received message.

Furthermore, to distinguish different Intent messages, each message is assigned a unique ID, which is also used as the corresponding snapshot's name. To support it, a new integer field "*IntentId*" is added into the Intent Java class. Each time an Intent message is created, the field is automatically added by one.

4) Array-Indexing Type Implicit Flow: To mitigate the problem caused by this type of implicit flows, we brute-forcely convert the associated operation into multiple conditional statements. Array and other data structures are handled respectively as follows.

- *Array*: Assume the content of an array A is $[e_0, e_1, e_2, ..., e_n]$, and in the operation *r = A[i]*, i is symbolic. The operation can be converted to the following structure :

```
if (0 == i) r = e_0;
else if (1 == i) r = e_1;
...;
else if (n == i) r = e_n;
```

  Next, EOEDroid can handle the operation as regular conditional statements.

- *Hashmap, Android Bundle, and Android Share Preference*: Similar to array-indexing operations, hashmap type accessing can also be transformed to conditional statements. Assume that the following instruction is faced: *r = hashmap.get(k)*. The keys of hashmap is $[k_0, k_1, k_2, ..., k_n]$. Hence, by instrumenting k's real value in memory, the operation can also be converted to regular conditional statements.

```
if (k.equals(k_0)) k = k_0;
else if (k.equals(k_1)) k = k_1;
...;
else if (k.equals(k_n)) k = k_n;
r = hashmap.get(k);
```

  To support the above operations, all keys in the hashmap object must be retrieved. However, it is challenging to do that in the low level layer (e.g., DVM). To fix the problem, the HashMap class is instrumented by adding a string array to record all keys. Thus, in the DVM, all keys can be retrieved by restoring the values of the added string array.

C. Program State Analysis

To discover how to reach the program state that leads to the execution of a critical functionality, we deal with the input and execution order of event handlers respectively.

1) Event Handler Input Generation: Given an arbitrary interesting path, its input can be generated by handling its associated path constraints that are collected in the first phase. First, input constraints are extracted from the whole path constraints by filtering out the constraints of non-symbolic conditional statements. Second, the input can be generated by resolving the input constraints using an SMT solver (e.g., Z3-Str).

2) Event Handler Execution Order Generation: Given a path that contains a critical functionality, the execution order of event handlers can be obtained by addressing the event handler dependency problem. The algorithm is shown in Algorithm 1. In the algorithm, three critical functions are required as input.

- *NS(eh, p, insn)*: Non-symbolic conditional statements can be extracted by going backward through p starting from $insn$ and checking the operands of all faced conditional statements.
- *get_origin_variables(eh, p, insn, v)*: We define the origin variables as following. If in p, v' can influence v's value, v' is an origin variable of v. Hence, to locate all v', we go backward through p starting from $insn$, and apply backward data flow tracking on v. If a variable is found in the backward data flow and located in the external environment, the variable may be one of v's origin variables.
- *get_origin_values(eh, p, O, insn, value)*: To compute the expected values of origin variables, we re-run symbolic execution on p to construct v's symbolic expression relying on origin variables. To this end, all origin variables in the set O are symbolized. Then, p is executed and analyzed by feeding eh with appropriate input. Next, when conditional statements are faced, the path constraint is constructed and saved. After that, when the instruction $insn$ is faced, the analysis is finished. Finally, the values of origin variables can be generated by resolving the collected path constraints.

D. Exploit Code Generation

Algorithm 2 shows our algorithm to generate the exploit code. Two main functions (*get_web_trigger_code()* and *get_js( )*) are required. The former function is implemented based on our study result (Table I), and the latter function is provided by the sub-module *"JavaScript code syntax analysis"*.

1) JavaScript Code Syntax Analysis: . It is challenging to generate required JavaScript code as part of an event handler's input. Because the JavaScript code is executed by associated WebView APIs (such as *loadUrl()*), the values of these APIs' parameters provide hints. Suppose the JavaScript code extracted from input is I, and the JavaScript code that already exists in associated WebView APIs (such as hard code format) is J. $I + J$ have complete semantics.

To mitigate the problem, we assume that I is atomic, i.e., it is a leaf element in the AST (Abstract Syntax Tree) of $I + J$. We can hence generate I based on its position in the AST. More specifically, when a WebView API that can execute JavaScript code (such as *WebView.loadUrl()*) is executed, its parameter's symbolic expression is dumped. Then, by replacing I with a specific concrete string (such as a randomized string), the concrete string of the parameter (i.e., $I + J$) is generated. Next,

Algorithm 1 Event Order Generation

```
Input:
 1: EH : all event handlers;
 2: P(eh) : return all paths in the event handler $eh$;
 3: $NS(eh, p, insn)$ : return all non-symbolic conditional statements before
    the instruction $insn$ in the path $p$ of the event handler $eh$;
 4: get_origin_variables(eh, p, insn, v) : return the variable $v$'s origin variables
    that influence $v$'s value;
 5: get_origin_values(eh, p, insn, v, value, O) : return the required values for
    all origin variables that can assign $value$ to $v$.
Output: the event order $R$
 1: function GENERATE_EVENT_HANDLER_ORDER(eh, p, expect_insn)
 2:     for $ns$ in NS(eh, p, expect_insn) do
 3:         c ← $ns$'s condition expression
 4:         v ← $c$'s value ▷ Depending on which branch is taken, v is true or
    false.
 5:         r ← resolve_event_handler_dependency(eh, p, ns, c, v)
 6:         if FAILURE == r then
 7:             return FAILURE
 8:         end if
 9:     end for
10:     return SUCCESS
11: end function
12:
13: function RESOLVE_EVENT_HANDLER_DEPENDENCY(eh, p, insn, variable,
    value)
14:     O ← get_origin_variables(eh, p, insn, variable)
15:     if O == $\phi$ then
16:         R ← {} return FAILURE
17:     end if
18:     for o in O do
19:         if o ∈ eh's parameters then
20:             R.add(<eh, p>)
21:         end if
22:     end for
23:     for ($o_i$, $v_i$) in get_origin_values(eh, p, insn, value, O) do ▷
    Rerun symbolic execution on the path $p$ to compute each origin variable's
    expected value
24:         for $eh'$ in E do
25:             for $p'$ in P($eh'$) do
26:                 $insn'$ ← the instruction writing $o_i$
27:                 r ← resolve_event_handler_dependency($eh'$, $p'$, $insn'$, $o_i$,
    $v_i$)
28:                 if FAILURE == r then
29:                     R ← {} return FAILURE
30:                 end if
31:             end for
32:         end for
33:     end for
34:     return SUCCESS
35: end function
```

Algorithm 2 Exploit Code Generation

```
Input:
 1: $EO$ : the event handler execution order, which is the set of the pair <$eh$,
    $p$>;
 2: get_input(eh, p) : return $eh$'s input that can guide the app to execute $p$;
 3: get_web_trigger_code(eh, parameter): return web code that can trigger $eh$
    and pass parameter to $eh$
 4: get_js(eh, p) : return required JavaScript code
 5:
Output: the exploit code X
 1: function GENERATE_EXPLOIT_CODE(eh, p)
 2:     for <$eh_i$, $p_i$> in EO do
 3:         X += gen_js($eh_i, p_i$)
 4:         input ← get_input($eh_i$, ($p_i$))
 5:         X += gen_event_trigger_code($eh_i, input$)
 6:     end for
 7: end function
```

by applying a JavaScript interpreter engine (such as Mozilla Rhino 1.6) in $I + J$, AST is generated. After that, I's semantics can be understood by checking AST's semantics and locating I in AST. Finally, concrete JavaScript code of I can be generated.

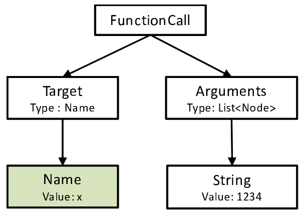


Figure 6: AST of $I + J$

We use the code in Figure 3 to illustrate how this submodule works. In the event handler *shouldOverrideUrlLoading()*, I is passed to *getId()* and executed to receive sensitive information. To automatically generate concrete JavaScript code of I, *loadUrl()*'s parameter is firstly dumped. Suppose the device ID is "1234". The parameter's symbolic expression is then ''javascript: + Uri.<init>(InputUrl).getHost().split(" .")[2] + ("1234")''. By replacing the symbolic data with a concrete string (such as "x"), a concrete example code of $I + J$ may be ''javascript:x(1234)''. Next, AST (Figure 6) can be generated by applying Rhino in the JavaScript code "x(1234)". By locating x in AST, we can find that x is a function name, and the function has only one string parameter. Hence, a JavaScript function (such as *steal_device_id()* in Listing 2) that satisfies the requirement can be defined in advance, and then the function name is passed to the event handler *shouldOverrideUrlLoading()*.

VI. Evaluation

To evaluate EOEDroid, we implemented it on Android 4.3, and deployed it in a Nexus 10 smartphone. Given apps, we started the random UI exploration tool Android Monkey [9] to trigger as many WebView components as possible.

Note that it is challenging to automatically trigger a UI component. To mitigate the problem, We run Monkey to simulate users' behaviors. Furthermore, we also use Monkey as the first-layer filter. The intuition is that if WebView is an important part of the app, it will be likely triggered in this way. Thus we reduce our workload by only considering the apps whose WebView components are successfully triggered in our dataset (Section VI-A).

Once a WebView complement is triggered, complete fuzzing code is injected to trigger all event handlers. More specifically, when WebView is going to connect to a web server, we start a crawler to check whether an HTTP link is involved in the connection. We limit the crawling depth in three levels. If there is an HTTP link, man-in-the-middle attacks is performed (Section III-B). The proxy tool "mitmproxy" [6] is used to inject web event trigger (fuzzing) code, which is generated based on the study result (Section V-A). Hence, once the injected code is loaded and executed in WebView, all event handlers are triggered, and then, EOEDroid is started to analyze them.

A. Dataset

In our evaluation, we collected apps as our evaluation dataset from two different app groups based on whether the WebView component could be triggered at run time. Both these two groups were collected from the Android official store Google Play. The first app group consists of 13,000 popular apps that we crawled from 26 categories, and extracted 500 most popular free apps for each category. The other app group contains 220 browser apps, which were collected by searching the key word 'web browser' in Google Play.

Finally, 3,652 apps were totally collected as our dataset, with 3,552 apps from the first app group and 212 apps from the second app group.

B. Findings

Our experiment casts light on the usage of event handlers in real-world hybrid apps. It also reveals interesting facts about EOE in hybrid apps.

1) Usage Of Event Handlers: Figure 7 shows the distribution of the usage of top 20 event handlers. *shouldOverrideUrlLoading()* and *onPageFinished()* are the two most frequently used event handlers.

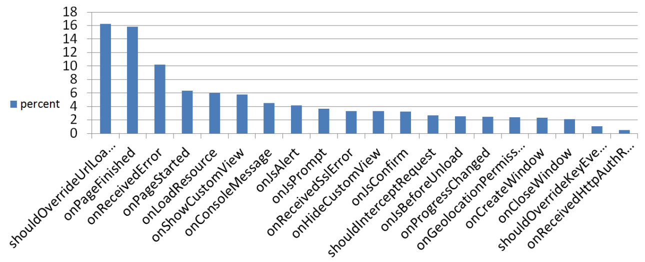


Figure 7: Usage Of Event Handlers

We also found most hybrid apps define their own event handlers. In our dataset (Section VI-A), 3,440 of 3,652 (94.2%) hybrid apps implemented their event handlers. It is clear that event handlers are in widespread use in real-world apps. Next we discuss the typical scenarios in which event handlers are used in apps.

Access Control. Event handlers can be applied to perform access control on the communication to be accessed, and the content to be loaded in WebView. For instance, *shouldInterceptRequest()* can check the content requested by web code. If the content is not expected, the event handler can directly return *null* to reject the access.

Customized URL Scheme. Event handlers can be used to support customized URLs. For instance, the link "tel:xx" and "smsto:xx" can be supported to make a phone call and send a text message.

Event Driven Authentication. Using customized URL schemes, event handlers can also be applied to perform authentication. Consider that *shouldOverrideUrlLoading()* supports a customized URL scheme "sdk". When the URL "sdk://auth_request" is received, the event handler redirects WebView to the authentication web site, while specifying the redirection URL as "sdk://auth_success". Hence, when the URL "sdk://auth_success" is received by the event handler, the event handler can learn the authentication is successfully done.

2) EOE In Event Handlers: By applying EOEDroid on the 3,652 hybrid apps, we successfully identified 97 vulnerabilities in 58 hybrid apps, as briefly shown in Table II.

Vulnerability Type	Number
Cross-Frame DOM Manipulation	2
Phishing	53
Sensitive Information Leakage	30
Local Resource Access	1
Intent Abuse	11

Table II: Vulnerabilities Found By EOEDroid

Distribution of vulnerable Event Handlers. We found that most vulnerabilities (96/97) existed in the event handler *shouldOverrideUrlLoading()*. The remaining two vulnerabilities were found in *onCreateWindow()* and *onReceivedHttpAuthRequest()*.

Phishing. We found the usage of the API *loadUrl()* to load new content in WebView likely introduced this type of vulnerabilities. It is mainly because developers wrongly assume the code loaded in WebView is trustable, and do not set up security checks before the sensitive API is called. In some apps, even though security checks were provided, these checks were incompetent to protect the critical functionalities and could be evaded. Take the following code as the example. Adversaries could still hit the sensitive API by feeding the input *'http://attacker.com/malicious/code?from=developer.com'*.

```
public boolean shouldOverrideUrlLoading(WebView view,
    String url) {
    else if (url.contains("developer.com")) {
        view.loadUrl(url);
        return true;
```

Cross-Frame DOM Manipulation. As shown in Table II, different from phishing, there were only a few cross-frame DOM manipulation vulnerabilities, even though $loadUrl()$'s parameter was totally controlled by adversaries. This is because that it is challenging to transfer the prefix string "javascript:" from the web code to the native code. Typically, in the web context, the prefix string "javascript:" is directly handled by JavaScript engine, rather than triggering any web events. However, using tricks it is still possible to deliver the prefix string. EOEDroid successfully discovered two vulnerable event handlers that could be leveraged to pass JavaScript code to the native context and execute the code. More details are discussed in our case studies in Section VI-C1.

Sensitive Information Leakage. In this category, EOEDroid successfully caught 26 vulnerable event handlers that could be utilized to steal Android ID. The further study showed that all of them were caused by an ad lib. The remaining 4 vulnerabilities were found in high profile browser apps. The first vulnerable event handler (from "com.webroot.xxx") could be leveraged to leak the phone number to a public log file, which could be accessed by any app. The second vulnerable event handler (from "com.kiddoware.xxx") could be triggered to leak IMEI. The third event handler (from "reactivephone.xxx") could be exploited to steal GPS location information using the input in a specific format. More specifically, if the URL to be accessed contained the string "latitude,longitude", the real GPS location data were retrieved to replace the string.

The last vulnerable event handler (from "com.mx.xxx") was interesting, which contained a potential backdoor that could be used to steal sensitive information, such as IMEI. Although developers had attempted to close the backdoor, EOEDroid found that it was still possible for adversaries to leverage the backdoor by changing the program state through the manipulation of execution orders of gadgets. More details are shown in our case study in Section VI-C2.

Local Resource Access. One vulnerable app was found that it could allow adversaries to access local database. Even though this app checked the origin information of web code that was going to access the database, it could still be bypassed by containing the developer website name.

Intent Abuse. One of the vulnerabilities was found in the event handler of the Korean Air app, which was allowed to send arbitrary intent message. Furthermore, the event handler also suffered from phishing attacks and cross-frame DOM manipulation.

Other ten vulnerabilities were found in browser apps. It was mainly because browser apps aimed to support the popular scheme “intent://”. However, these apps did not check the origin information, and specify the action or destination class, which might cause serious problems, as demonstrated by Wang et al. [36].

C. Case Studies

App	Input Format
com.exsoul.xxx	“exsoul://id=[0-9]{8}&url=”
com.fevdev.xxx	“intent://...fallback_url=”

Table III: The Input Format Of The Two Vulnerable Apps Shown In Case 1

1) Case 1: Cross-Frame DOM Manipulation: This section presents two vulnerable apps that suffer from cross-frame manipulation attacks. To transfer the prefix string “javascript:”, the input is crafted following the input format shown in Table III. When the input is received and parsed by the event handler *shouldOverrideUrlLoading()*, the content l of “url” and “fallback_url”is extracted and then fed into a sensitive API *loadUrl()*. Hence, if l is in the format “javascript:...”, the JavaScript code can be then executed in the main frame.

2) Case 2 : Leveraging A Closed Backdoor: This high profile app has been downloaded more than 10 million times. The Listing 3 shows a code snippet of the vulnerable event handler. In this app, the variable *flag* (Line 1) is initially false. When the event handler *shouldOverrideUrlLoading()* is triggered, several conditional statements are determined relying on the flag (Line 12) and the URL. In Line 24, the URL is saved to a local variable, and then “%IMEI%” is replaced with real IMEI.

```
1  flag = false;
2
3  public void onPageFinished(WebView view, String url) {
4    ...
5    flag = true;
6    ...
7  }
8
9  public boolean shouldOverrideUrlLoading(WebView view,
        String url) {
10   ...
11   url = url.toLowerCase();
12   if (!flag)
13     ...
14   else {
15     if (url.startsWith("http://") || url.startsWith("https
           ://"))  ...
16     else if (url.startsWith("file://")||url.startsWith("
           content://")) ...
17     else if (url.startsWith("mx")) ...
18     else {
19       if (url.contains("app_name")) {
20         ...
21         String tmpstr = url;
22         // read imei from shared preference
23         String i = PreferenceManager.
                getDefaultSharedPreferences(this).getString("
                imei", "");
24         tmpstr = tmpstr.replaceAll("%IMEI%", i)
25         ...
26         // send a Intent message containing tmpstr
27         Intent intent = new ...;
28         intent.setData(Uri.parse(tmpstr));
29         startActivity(intent)
30   ...
```

Listing 3: Code snippet extracted from the example in case

By applying EOEDroid on this app, the vulnerable event handler’s path constraints are collected, which are shown as follows.

```
(1) InputUrl.startsWith("http://") == 0
(2) InputUrl.startsWith("https://") == 0
(3) InputUrl.startsWith("file://") == 0
(4) InputUrl.startsWith("content://") == 0
(5) InputUrl.startsWith("mx") == 0
(6) InputUrl.contains("app_name") == 1
(7) flag == 1
(8) InputUrl.contains("%IMEI%") == 1
```

All constraints can be satisfied except (7). By addressing the event handler dependency problem on (7), the event handler execution order is generated : $onPageFinished() \rightarrow shouldOverrideUrlLoading()$.

However, due to the trigger constraint (Section V-A), we found *onPageFinished()* was executed after *shouldOverrideUrlLoading()*. Hence, to generate the required execution order, the web page should be refreshed as follows.

```
(1) <script> window.location.reload(true); </script>
```

Then, the web code that can guide *shouldOverrideUrlLoading()* to execute the sensitive API *getDeviceId()* is shown as follows, if assuming FTP is supported by users’ phone.

```
(2) <iframe src="ftp://attacker.com/app_name?imei=%imei%"/>
```

D. Performance and Accuracy

The performance and accuracy of EOEDroid may be impacted by our symbolic execution implementation, where several heuristics are leveraged to mitigate the path explosion problem. Admitting that these heuristics may cause over approximation and/or inaccuracy to our analysis, they help us make a good tradeoff between performance and accuracy. In this section, we presented more evaluation details, and showed that our current system performance and accuracy were acceptable.

For each app, the average successful analysis time of EOEDroid is around 4.2 minutes, including 3.4 minutes for the event handler analysis. Considering our tool is designed to analyze apps offline, the overhead is acceptable.

We use false positives (FP) and false negatives (FN) to measure EOEDroid’s accuracy. We define a FP as that a non-vulnerable event handler is flagged as vulnerable, and a FN as that a vulnerable event handler is identified as non-vulnerable.

False Positives. We manually analyzed all vulnerable event handlers by running the exploit code generated by EOEDroid. Finally, we found that all vulnerabilities were successfully triggered, which indicated EOEDroid’s FP rate was low.

False Negatives. To confirm false negatives, we randomly selected 200 apps from the hybrid apps that were flagged as non-vulnerable by EOEDroid. By carefully manually checking their event handlers, we found all apps were non-vulnerable except two apps. Our further study on these two apps showed that the main reason was that the SMT solver failed to resolve some path constraints that contained multiple regular expressions and string split operations. This still represents a low FN rate for EOEDroid.

VII. EOE Countermeasure Discussion

The key to counter EOE is that apps should only allow trustable web code to access critical functionalities in event handlers. To achieve this, apps should first fully use HTTPS in all communications, which will effectively reduce the attack surface. Second, when a critical functionality is called through an event handler, the frame level and origin information of web code should be carefully checked.

The newest version of Android provides a new setting that only allows web code downloaded over HTTPS to access *shouldOverrideUrlLoading()*, and also includes more information in the event handler's parameters, such as the frame level and origin information of web code. Hence, we strongly recommend developers port their apps to the new version, and leverage these security information in their development.

VIII. Related Work

Attacks on WebView. Recently, security issues caused by event handlers have received significant attention from researchers. Luo et al. [26] discussed that event handlers may be used by malware to hijack and sniff web events. However, compared with EOE, this type of attacks is more difficult to launch, because adversaries have to control the native code in user devices, such as registering their own native event handlers in WebView. Chen et al. [13] and Mutchler et al. [27] discovered the event handler feature may cause sensitive data leakage (such as the authentication URL) in Oauth. Georgiev et al. [19] and Tuncay et al. [35] discussed the possibilities that adversaries may leverage the event handler feature to access native code. In contrast, we systematically study all types of feasible web event oriented attacks, including the attacks that are carried out by leveraging both one single web event and stitching multiple web events together to influence the program state.

Compared with existing attacks on WebView, EOE is more feasible and practical. Chin et al. [14] analyzed WebView vulnerabilities that result in excess authorization and file-based cross-zone scripting attacks. Wu et al. [40] discussed file leakage problems caused by file:// and content:// schemes in webview. However, these two kinds of attacks are limited in the Android new versions, which provide better protections on directly accessing local files.

Bhavani et al. [10] also studied the possibility of cross-site scripting attacks in WebView. Neugschwandtner et al. [28] described data leakage scenarios and presented several real-world case studies of JavaScript injection attacks through WebView. Jin et al. [24] systematically investigated the JavaScript code injection consequences on hybrid apps and showed the pervasiveness of data leakage due to classic web attack vectors that are possible through WebView. Wei et al. [38] introduced attack scenarios where attackers could exploit existing vulnerabilities (such as CVE-2012-6636 [2] and CVE-2013-4710 [3]) to invoke arbitrary Java functions in WebView. Rastogi et al. [30] demonstrated the hidden attacks based on app-web bridges. However, all above attacks require JavaScript and JavaScript-Bridge to be enabled, whereas EOE does not have such requirement.

Wang et al. [36] systematically studied the Intent abuse problem and demonstrated the serious consequences. However, this attack requires the pre-installation of a WebView-enabled malware in user devices, which is not required in EOE.

Yang et al. [41] and Hassanshahi et al. [22] studied app-web bridge based attacks, and proposed detection solutions to vet hybrid apps. However, they either did not support the event handler feature, or focus on the attacks launched from a special URL navigation event (i.e., "intent://..."). In contrast, EOEDroid is generic.

Defense On WebView. Several defense approaches, such as NoFrak [19], MobileIFC [33], and Draco [35], are proposed to extend SOP to local resources, or provide access control on event handlers in the native layer. However, there are difficulties in applying existing approaches to prevent the EOE attacks. First, Draco requires the root permission to replace WebView's internal native library, and MobileIFC and NoFrak also require the recompilation of hybrid apps with their own customized hybrid frameworks. Second, they are implemented by instrumenting WebView or third-party hybrid frameworks. Hence, they may have to keep doing extra more work in porting their systems into newest versions. Third, the defense level totally depends on how well the security policies are written by developers. Finally, they performed access control based on the web frame's origin information. Hence, it is challenging for them to limit the access from embedded inline JavaScript code.

Other defense approaches, such as WIREframe [17] and HybridGuard [29], provided policy enforcement in WebView to protect app-web bridges. However, both of them only focused on JavaScript code and yet ignored HTML code. Hence, they can still be evaded by EOE, since EOE can be launched purely in HTML code.

Symbolic Execution. In past years, symbolic execution has made big progress. Several static approaches (such as Intellidroid [39] and TriggerScope [18]) were proposed to vet Android apps using symbolic execution. However, these static approaches may have both higher false positives and negatives in the context faced in this paper. First, static analysis has to address points-to and alias problems. Second, due to the lack of real data, it is challenging to resolve Java Reflection and Intent. Finally, it is difficult to address the array indexing type implicit flows. In real world, this type of implicit flows is frequently used in popular apps and ad libs, such as Google Ads.

Many dynamic approaches were also implemented based on symbolic execution. For example, DART [20] and CUTE [31] applied concolic execution to automatically test software. EXE [12] and KLEE [11] used symbolic execution to find bugs. IntScope [37] employed symbolic execution to detect integer overflow problems. SAGE [21] was designed for Windows to apply symbolic execution to vet the operating system. S2E [15] proposed the selective symbolic execution to improve the performance. Driller [34] used selective symbolic execution to guide fuzzing, and the result showed the combination was very effective. Existing dynamic approaches may have low false positives. However, it is challenging for them to generate the event sequences required for triggering a found vulnerability.

Several symbolic execution based approaches were also designed to handle implicit flows. For instance, DTA++ [25] used symbolic execution to solve control flow problem (i.e., implicit flows), while Spandex [16] implemented symbolic execution in Android to vet apps about password usage. However, these two systems fall short of handling Android specifications (such as Android Intent) and array indexing type implicit flows.

IX. System Limitations and Future Work

EOEDroid is not perfect. First, currently we simply use Monkey to trigger WebView. Exploring all possible UI components is a difficult issue, though orthogonal to this research. Second, in EOEDroid, we do not solve all implicit flow problems, instead only focus on array-indexing type operations, which are frequently used in event handlers. Finally, we do not handle all native code in Android, instead only model important native code such as *system.arraycopy()*. In future work, we plan to explore solutions in these directions to improve EOEDroid.

X. Conclusion

In this paper, we thoroughly studied all web events, native event handlers and their triggering constraints. Based on our findings, we present EOEDroid, a novel system that can automatically detect and verify EOE vulnerabilities by generating exploit code. We evaluated EOEDroid using a large number of apps and found several critical vulnerabilities.

Acknowledgment

We thank all anonymous reviewers and our shepherd, Adam Doupé, for their insightful comments and suggestions. This material is based upon work supported in part by the the National Science Foundation (NSF) under Grant no. 1314823 and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Calling fork() from jni code. https://groups.google.com/forum/#\protect\kern-.1667em\relaxtopic/android-platform/80jr-_A-9bU.

[2] Cve-2012-6636. https://cxsecurity.com/cveshow/CVE-2012-6636.

[3] Cve-2013-4710. https://cxsecurity.com/cveshow/CVE-2013-4710.

[4] Dalvik. https://en.wikipedia.org/wiki/Dalvik_(software).

[5] Dalvik opcode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[6] An interactive tls-capable intercepting http proxy for penetration testers and software developers. *https://github.com/mitmproxy/mitmproxy*.

[7] Mcafee mobile threat report. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf.

[8] Same origin policy. https://en.wikipedia.org/wiki/Same-origin_policy.

[9] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html.

[10] A. B. Bhavani. Cross-site Scripting Attacks on Android WebView. *IJCSN International Journal of Computer Science and Network*, 2(2):1–5, 2013.

[11] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *CCS'06*.

[13] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. Oauth demystified for mobile application developers. CCS'14, 2014.

[14] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *WISA'13*. Jeju Island, Korea.

[15] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS'11*.

[16] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *USENIX Security'14*.

[17] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha. Secure integration of web content and applications on commodity mobile operating systems. ASIA CCS'17, New York, NY, USA.

[18] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Apps. In *IEEE S&P'16*, San Jose, CA.

[19] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS'14*, San Diego, USA.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI'05*.

[21] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[22] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang. Web-to-application injection attacks on android: Characterization and detection. In *ESORICS*, volume 9327, pages 577–598. Springer, 2015.

[23] InfoSecurity. Public wifi hotspots ripe for mitm attacks. https://www.infosecurity-magazine.com/news/public-wifi-hotspots-ripe-for-mitm-attacks/.

[24] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *CCS'14*.

[25] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *NDSS'11*. San Diego, California, USA.

[26] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC'11*.

[27] P. Mutchler, A. DoupÃ, J. Mitchell, C. Kruegel, G. Vigna, A. Doup, J. Mitchell, C. Kruegel, and G. Vigna. A Large-Scale Study of Mobile Web App Security. *MoST'15*.

[28] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: Webview exploitation. In *LEET'13*.

[29] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. MoST'17.

[30] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces. *NDSS'16*.

[31] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *ACM SIGSOFT'05*, Lisbon, Portugal.

[32] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07*.

[33] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *RAID'13*.

[34] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS'16*.

[35] G. S. Tuncay, S. Demetriou, and C. A. Gunter. Draco: A system for uniform and fine-grained access control for web code on android. In *CCS'16*.

[36] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS'13*.

[37] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS'09*, San Diego, CA.

[38] T. Wei, Y. Zhang, H. Xue, M. Zheng, C. Ren, and D. Song. Sidewinder targeted attack against android in the golden age of ad libraries. In *Black Hat'14*.

[39] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS'16*.

[40] D. Wu and R. K. C. Chang. Indirect File Leaks in Mobile Applications. In *MoST'15*.

[41] G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In *RAID'17*.

[42] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE'13*.