ωTest: WebView-Oriented Testing for Android Applications

Jiajun Hu
jhuao@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Lili Wei
lili.wei@mcgill.ca
McGill University
Montreal, Canada

Yepang Liu*
liuyp1@sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Shing-Chi Cheung†
scc@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

ABSTRACT

WebView is a UI widget that helps integrate web applications into the native context of Android apps. It provides powerful mechanisms for bi-directional interactions between the native-end (Java) and the web-end (JavaScript) of an Android app. However, these interaction mechanisms are complicated and have induced various types of bugs. To mitigate the problem, various techniques have been proposed to detect WebView-induced bugs via dynamic analysis, which heavily relies on executing tests to explore WebView behaviors. Unfortunately, these techniques either require manual effort or adopt random test generation approaches, which are not able to effectively explore diverse WebView behaviors. In this paper, we study the problem of test generation for WebViews in Android apps. Effective test generation for WebViews requires identifying the essential program properties to be covered by the generated tests. To this end, we propose *WebView-specific properties* to characterize WebView behaviors, and devise a cross-language dynamic analysis method to identify these properties. We develop ωTest, a test generation technique that searches for event sequences covering the identified WebView-specific properties. An evaluation on 74 real-world open-/closed-source Android apps shows that ωTest can cover diverse WebView behaviors and detect WebView-induced bugs effectively. ωTest detected 36 previously-unknown bugs. From the 22 bugs that we have reported to the app developers, 13 bugs were confirmed, 9 of which were fixed.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android, WebView, Test Generation, Coverage Criteria

*Yepang Liu is affiliated with both the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonoumous Systems, Southern University of Science and Technology.
†Corresponding Author

ISSTA '23, July 17–21, 2023, Seattle, WA, United States
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States*, https://doi.org/10.1145/3597926.3598112.

ACM Reference Format:
Jiajun Hu, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2023. ωTest: WebView-Oriented Testing for Android Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598112

1 INTRODUCTION

WebViews are user interface (UI) widgets of the Android framework to display web pages in Android apps [20]. They are instances of the `android.webkit.WebView` class. WebViews not only help integrate web applications developed in HTML/JavaScript into Android apps but also provide mechanisms to support interactions between the web-end (JavaScript) and the native-end (Java) of an Android app. WebViews are widely used in real-world Android apps. An earlier study [47], which analyzed over one million apps from Google Play store, showed that 85% apps use WebViews in some fashion. A recent study [64] that randomly crawled 6000 popular apps from two leading app stores demonstrated similar results (i.e, 90.6% of them use WebViews). A dataset containing 6400 most-popular Google Play apps [14] recently crawled by us also showed that 83.74% apps use WebViews in their code. Furthermore, a new trending ecosystem called app-in-app (e.g., WeChat Mini-Programs [21]) in which a host-app uses WebViews to embed sub-apps is adopted by many high-profile apps [41, 57, 64, 66]. In reality, millions of active users are interacting with WebViews everyday [2].

Despite the popularity, WebView programming is error-prone due to its complicated cross-language interaction mechanisms (e.g., bridge communication [25, 39] and WebView callbacks [61]). Existing studies have investigated various types of bugs induced by the misuses of WebViews (e.g., security issues [27, 33, 39, 40, 42, 49, 50, 54, 61–63, 65]). They also proposed different bug detection techniques based on static/dynamic analysis. A fundamental limitation of static analysis techniques [33, 39, 49, 50, 62, 65] is their inability of analyzing dynamically loaded web data. For example, some JavaScript code may not be available for analysis until they are loaded at runtime. In comparison, dynamic analysis techniques [27, 35, 40, 54, 61, 63] exercise WebViews through testing, and therefore do not suffer from such a limitation. Nonetheless, the effectiveness of dynamic analysis techniques heavily relies on the quality of the generated tests. For example, manifesting a WebView-induced bug may require the tests to trigger some specific events on the buggy WebViews. Yet, most of the existing dynamic analysis techniques either adopt manual [27] or random approaches for

test generation [35, 61, 63]. Their generated tests cannot effectively explore diverse WebView behaviors to expose hidden issues.

Conventional general-purpose test generation techniques for Android apps are also ineffective for WebView testing. A typical category of these techniques is model-based graphical user interface (GUI) testing [26, 31, 32, 34, 43, 48, 51, 59], whose objective is to generate tests to visit more GUI states of an Android app. The GUI state of an app, which is modeled by the hierarchy of the rendered UI elements, is an abstraction of app behaviors. Intuitively, visiting more GUI states that "look different" means that more app behaviors are explored. However, GUI states may not be a good abstraction of WebView behaviors. Loading a new page in a WebView does not necessarily mean that a new WebView behavior is explored. For example, opening two different websites in a WebView-wrapped browser app may exercise the same page-loading process of WebViews. Another typical category of conventional techniques is to guide test generation by the coverage of some program properties [45, 46, 51, 56] that reflect the test objectives. Program statements and branches are two commonly adopted properties. For example, Sapienz [46], an Android test generation technique, regards the maximization of statement coverage as a main objective in test generation. However, the program properties adopted by existing work are not suitable for testing WebViews since none of them are specifically designed to model WebView behaviors. For example, not all statements in an app are used for implementing WebView functions. Leveraging these properties cannot guide the generation of tests to systematically examine WebView behaviors. Therefore, an effective test generation technique targeting WebViews in Android apps is needed.

To effectively test WebViews, it is desirable to define specific test objectives and propose properties accordingly to guide test generation. A straightforward solution is to take WebView API[1] call sites as the properties for tests to cover. However, such a design of properties is inadequate: it only captures the interaction sites between the web-end and native-end but ignores the involved data exchanges. As we will illustrate in Section 2, effective WebView testing should also examine how the data sent from the web-end are manipulated at the native-end and vice versa. Based on this observation, we propose a novel design of *WebView-specific properties* that considers both WebView API call sites and the data exchanges involved in the web-native interactions. We also devise a cross-language dynamic analysis method to identify these properties. We further propose a WebView-oriented test generation technique ωTest. ωTest is powered by a novel fitness function, which awards those tests that can cover diverse WebView-specific properties.

We evaluated ωTest on 44 open-source and 30 closed-source Android apps and compared it with 5 baseline methods in terms of property coverage and the number of detected WebView-induced bugs. Our evaluation results show that ωTest achieved the highest coverage and detected the most number of bugs. ωTest detected 36 previously-unknown bugs. From the 22 bugs that we have reported to the corresponding app developers, 13 bugs were confirmed and 9 of them have been fixed. To summarize, this paper makes the following major contributions:

- **WebView-specific properties:** We propose the first design of WebView-specific properties to formally characterize WebView behaviors in Android apps.
- **ωTest:** We propose a test generation technique ωTest, which leverages the proposed properties to generate tests that are able to explore diverse WebView behaviors.
- **Implementation and evaluation:** We implemented ωTest and evaluated its performance on real-world Android apps. It detected 36 previously-unknown bugs in various apps. We make the tool and experiment dataset available to facilitate future research [23].

2 MOTIVATION AND PROPERTY DESIGN

Program properties (e.g., program statements) are entities that characterize the program behaviors of interest. The coverage of these properties provides an effective measure of test adequacy for the interested behaviors. We consider both WebView APIs call sites and the data exchanges in the web-native interactions as *WebView-specific properties*. We demonstrate the need of doing so using two web-native interaction scenarios in Wikipedia [22].

2.1 Two Interaction Scenarios

Figure 1 gives the code of the two interaction scenarios. Boxes and arrows marked in blue and yellow represent Scenario #1 and #2, respectively. Boxes marked in grey represent the functions shared by both scenarios. Both of the scenarios (1) are initiated at the native-end, (2) invoke a JavaScript function to pass data to the web-end via the WebView API `loadUrl()` (line 14), and (3) send the results generated at the web-end back to the native-end via the WebView callback `onJsPrompt()` (line 37). The two scenarios share the same interfaces that handle cross-language data transmissions (grey boxes). The handling is different according to the interaction *type*. Scenario #2 contains a reported bug [1].

Scenario #1 depicts the process of loading an article section. Its interaction type is "*section*". Figure 1 shows a section introducing cranberry. In this scenario, the native-end constructs a JSON-format `msg` containing a requested `url` (line 3) and an `offset` (line 4). It calls `loadUrl()` to pass `msg` and the interaction `type` (`"section"` specified at line 5) as arguments to the web-end through a dynamically constructed JavaScript code (line 14). The web-end retrieves the data in `handleMessage()` and dispatches `msg` to the handler for type `"section"` (line 17). The handler (lines 19–25) then fetches the resources through an HTTP request based on the information in `msg` (lines 20–21). If an error occurs, the error status will be sent back to the native-end (line 23). The error message is sent by prompting a dialog (line 35) and thereby triggering the native-end WebView callback `onJsPrompt()` (lines 37–42). The native-end receives the error as the callback parameter (line 37) and handles it at line 46.

Scenario #2 depicts the process of handling a click on a link. Its interaction type is "*onClick*". This scenario is initiated at the native-end (lines 7–12) by packing the link `url` and the `id` of the DOM element that consumes the click event into `msg` (lines 9–10). It calls `loadUrl()` (line 14) and passes the `msg` to the handler (line 26) for `"onClick"` at the web-end. If the DOM element contains an image (line 28), a message of type `"imageClicked"` is sent back to the

[1] WebView APIs include APIs of classes under packages `android.webkit` and `androidx.webkit`, and bridge methods [7].

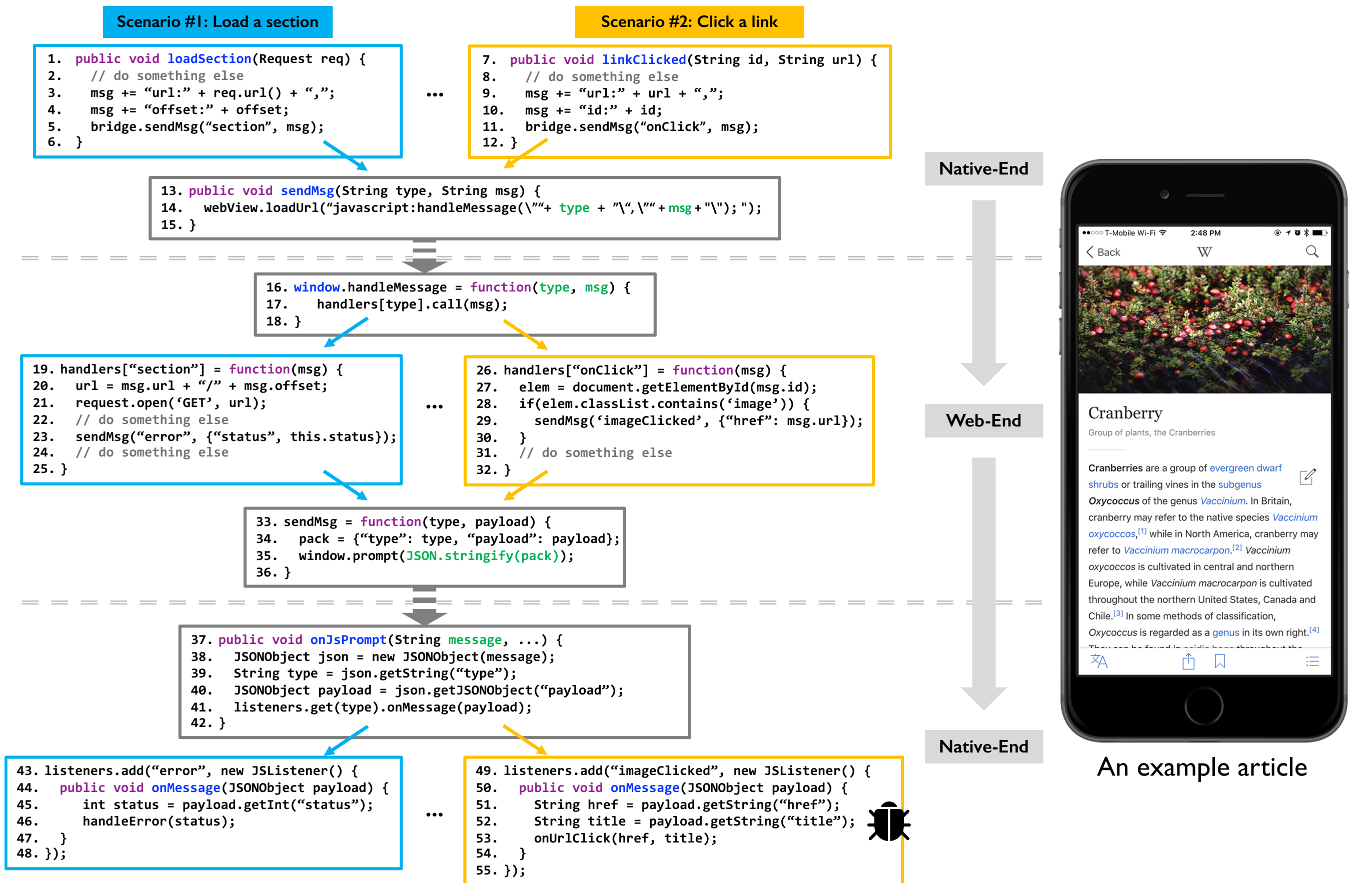

Figure 1: Two web-native interaction scenarios in the Wikipedia Android app (the code is simplified and adapted to ease understanding).

corresponding event handler at the native-end (line 49). This scenario contains a bug that results in a crash when a `JSONException` is thrown at line 52 [1]. The exception occurs because the Java code is looking for a `"title"` attribute from a JSON-typed object `payload` (line 52) prepared by the JavaScript code at line 29, where the `"title"` attribute is not set.

2.2 Limitation of WebView API Call Sites

We can see from the example that it is inadequate to characterize different web-native interaction scenarios by considering only WebView API call sites (`loadUrl()` at line 14 and `onJsPrompt()` at line 37) as properties for test coverage. While the two scenarios explore different WebView behaviors, a test enacting either one of them can cover all these call sites. A test generation technique solely driven by WebView API call sites may consider the test exploring Scenario #2 redundant after generating the test for Scenario #1, thus missing the bug residing in Scenario #2. This motivates us to propose new properties that can better characterize WebView behaviors.

2.3 New Design of WebView-Specific Properties

From the example, we can make a key observation. Besides the call sites of WebView APIs, it is necessary to consider the data exchanges across the language boundaries to characterize WebView behaviors. In other words, ***program variables that carry the transmitted data between the web-end and the native-end should be captured by WebView-specific properties.*** We call such variables ωVars. A higher coverage of these properties is likely to explore more WebView behaviors. For example, a test that covers ωVars `req.url()`, `offset`, and `status` exercises Scenario #1, while a test that covers ωVars `href`, `id`, and `title` exercises Scenario #2. With ωVars, two scenarios can be distinguished from each other. However, it is challenging to identify ωVars. Unlike conventional program properties such as program statements or branches, ωVars cannot be simply identified from program structures. Their identification involves two challenges:

Challenge 1: JavaScript code can be dynamically constructed at runtime. If it cannot be precisely determined, we may miss ωVars at the web-end. For example, line 14 in Figure 1 builds a string of JavaScript code. When a section is loaded, the string can be *javascript:handleMessage("section", {url: "...", offset: ...});*. The `url` and `offset` field of the object expression (the second argument of `handleMessage()`) should be identified as ωVars since their values are set by the Java code. Without recognizing the JavaScript code,

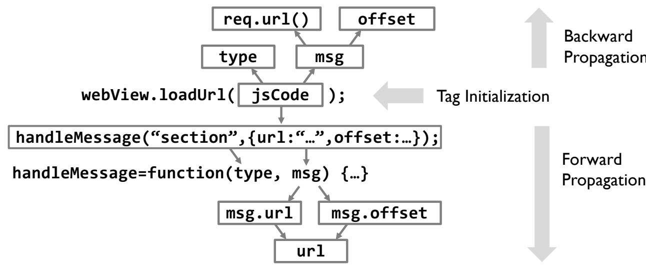

Figure 2: An overview of ωVar identification.

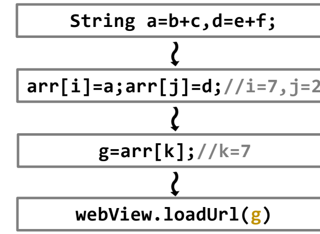

Figure 3: Motivation of variable tracking

these two ωVars can be missed, further affecting the identification of more ωVars that depend on them (e.g., `url` at line 20).

Challenge 2: Not all variables are ωVars. It is difficult to statically identify ωVars precisely due to language or Android framework features (e.g., dynamic data structures, intent, etc.) that involve dynamic decisions. For example, static analysis often adopts an over-approximation strategy when analyzing collections. It may consider all the elements in an array to be ωVars even though only some of them carry transmitted data.

If these ωVars cannot be precisely identified, the generated tests are ineffective. To address the challenges, we devise a set of *dynamic* ωVar tagging rules to identify WebView-specific properties *on-the-fly* during testing. We present the method in the next section.

3 DYNAMICALLY IDENTIFYING WEBVIEW-SPECIFIC PROPERTIES

This section presents a dynamic analysis method to identify WebView-specific properties in an Android app. Figure 2 gives an overview of the method. The idea is to first initialize a set of ωVars that are directly used by WebView APIs (***Tag Initialization***). Then, other variables depending on existing ωVars and vice versa are iteratively identified and added to the set of ωVars using ***Forward*** and ***Backward Propagation***, respectively. Figure 2 abstracts the dependencies between variables in Scenario #1 in Figure 1. In this example, when `loadUrl()` is executed by a test, its argument `jsCode` is the first variable to be tagged as an ωVar. Then the variables that `jsCode` depends on are iteratively tagged as ωVars by traversing the variable dependencies built during test execution according to a set of *backward propagation* rules. For example, since `jsCode` is built from `type` and `msg`, both of them are tagged as ωVars. Variable `msg` further depends on the return value of `req.url()`[2] and `offset`, the corresponding variables are also tagged as ωVars. Next, `loadUrl()` will execute the JavaScript code. We analyze the code and identify the JavaScript variables whose values are defined by Java code. In this example, the string argument `"section"`, the `url` and `offset` field of the object expression (the second argument) of `handleMessage()` are tagged as ωVars. We further use a set of *forward propagation* rules to tag new ωVars depending on existing ωVars at the web-end. In Figure 2, `url` is tagged since it depends on two existing ωVars, `msg.url` and `msg.offset`.

Table 1 shows the tagging rules to identify ωVars. We use a function $\omega(x)$ to indicate if a variable x is an ωVar : if x is an ωVar, the value of $\omega(x)$ is 1; otherwise 0. The code that implements the tagging logic is injected into an app using program instrumentation. Therefore, it is independent from any test generation approaches. In the following, we present the three steps to identify ωVars: *Tag Initialization, Backward Propagation, and Forward Propagation.*

3.1 Tag Initialization

In the first step, we identify an initial set of ωVars based on *tag initialization rules*, which consider the following five circumstances according to the semantics of WebView APIs [7].

(1) *Parameters and return variables of WebView callback methods are tagged as ωVars.* These native-end parameters carry the information of events occurring at the web-end. For example, the parameter `message` of the callback method `onJsPrompt()` at line 37 in Figure 1 carries the prompted message sent by `window.prompt()` at line 35. The callback return also affects WebView behaviors. For instance, WebViews determine whether to continue loading a URL according to the return of `shouldOverrideUrlLoading()` when a user clicks a URL.

(2) *Parameters and return variables of bridge methods [39] are tagged as ωVars.* Bridge methods are Java methods that can be directly invoked by JavaScript [7]. The parameters of bridge methods are passed from the web-end to the native-end. The result computed at the native-end is passed back to the JavaScript code by the return values of bridge methods.

(3) *Arguments passed to WebView APIs and variables saving return values of WebView API invocations are tagged as ωVars.* These APIs are invoked to control the behaviors of WebViews (e.g., the display of a web page) through their arguments or extract information from the web-end (e.g., the current URL) through their return values.

(4) *Arguments passed to bridge method invocations and the variables saving the return values of bridge method invocations are tagged as ωVars.* Different from rule (2), this rule captures the variables involved with bridge methods at the web-end.

(5) *The literals in the JavaScript code that is dynamically constructed and loaded by WebView APIs that can execute JavaScript (e.g.,* `loadUrl()`*) are tagged as ωVars.* The reason for tagging only the literals is that once a string representing JavaScript code is formatted, only the values of literals are given by the Java code. For example, the constructed JavaScript code at line 14 of Figure 1 can be *javascript:handleMessage("section", {url: "https://...", offset: 3})* when sections are loaded. The literals *"section"*, *"https://..."*, and *3* are tagged as ωVars as their values are given by three Java variables, respectively (i.e., `type`, `req.url()`, `offset`). We implement this by intercepting the constructed JavaScript code at runtime. We

[2] We instrument an app to inject the propagation logic at instruction level. There will be a temporary variable to hold the return of req.url() and this variable will be tagged.

Table 1: ωVar Tagging Rules ([] means optional)

	No.	Operation Semantics	Propagation/Tagging Rules	Applied Language
Tag Initialization	1	$webview_callback([y_1, y_2, ..., y_n])\{...[return\ r;]\}$	$\omega(y_1)=1, \omega(y_2)=1, ..., \omega(y_n)=1, \omega(r)=1$	Java
	2	$bridge_mtd([y_1, y_2, ..., y_n])\{...[return\ r;]\}$	$\omega(y_1)=1, \omega(y_2)=1, ..., \omega(y_n)=1, \omega(r)=1$	Java
	3	$[x=]webview_api_Ivk([y_1, y_2, ..., y_n])$	$\omega(y_1)=1, \omega(y_2)=1, ..., \omega(y_n)=1, \omega(x)=1$	Java
	4	$[x=]bridge_ivk([y_1, y_2, ..., y_n])$	$\omega(y_1)=1, \omega(y_2)=1, ..., \omega(y_n)=1, \omega(x)=1$	JavaScript
	5	$execJS(jsCode)$	$\{\omega(x)=1 \mid x \in jsCode \wedge isLiteral(x) == true\}$	JavaScript
Backward Propagation	6	$x = op\ y$	$\omega(x) \rightarrow \omega(y)$	Java & JavaScript
	7	$x = y\ op\ z$	$\omega(x) \rightarrow \omega(y), \omega(x) \rightarrow \omega(z)$	Java & JavaScript
	8	$x = lib(y_1, y_2, ..., y_n)$[1]	$\omega(x) \rightarrow \omega(y_1), \omega(x) \rightarrow \omega(y_2), ..., \omega(x) \rightarrow \omega(y_n)$	Java & JavaScript
	9	$x.y = z$	$\omega(x) \rightarrow \omega(z)$	Java & JavaScript
	10	$arr[i] = x$	$\omega(arr) \rightarrow \omega(x)$	Java & JavaScript
Forward Propagation	11	$x = op\ y$	$\omega(y) \rightarrow \omega(x)$	Java & JavaScript
	12	$x = y\ op\ z$	$\omega(y) \mid \omega(z) \rightarrow \omega(x)$	Java & JavaScript
	13	$x = lib(y_1, y_2, ..., y_n)$[1]	$\omega(y_1) \mid \omega(y_2) \mid ... \mid \omega(y_n) \rightarrow \omega(x)$	Java & JavaScript
	14	$x = y.z$	$\omega(y.z) \rightarrow \omega(x)$	Java & JavaScript
	15	$x = arr[i]$	$\omega(arr[i]) \rightarrow \omega(x)$	Java & JavaScript

[1]lib stands for library methods, which include Android system APIs and APIs in third party libraries. We rely on existing tools [24, 44] to identify them if an app's source code is not available (i.e., closed-source apps).

instrument the JavaScript code to inject the tagging logic into it before the code is loaded by WebView APIs.

3.2 Backward Propagation

Backward propagation is used to iteratively identify new ωVars that existing ωVars depend on. Backward propagation based on static analysis may misclassify many variables as ωVars. Figure 3 depicts one scenario. At the beginning, two string variables `a` and `d` are put into `arr[i]` (i=7) and `arr[j]` (j=2), respectively. Then at another place `arr[k]` (k=7) is assigned to `g`, which will be tagged as an ωVar when it is used by `loadUrl()`. In this scenario, it is difficult to statically infer that only `arr[7]` is an ωVar. An over-approximation strategy based on static analysis can incorrectly classify all the elements in the array as ωVars and subsequently tag properties irrelevant to WebView behavior. In this case, both `a` and `d` will be tagged as ωVars, which will further lead to the tagging of `b`, `c` (which `a` depends on) and `e`, `f` (which `d` depends on). However, only `a` is used by `loadUrl()`. As such, we address the problem using a dynamic ***variable tracking*** strategy.

3.2.1 Variable Tracking. We adapt the idea of Phosphor [28, 29] and BridgeTaint [27] to track variables in Java and JavaScript respectively at runtime. The idea is to leverage a unique *ID* assigned to each variable at its initialization to precisely identify created variables at runtime. During an app's execution, variable dependencies specified by the backward propagation rules are memorized using the ID assigned to each variable. When a WebView API is invoked, we will use the recorded dependencies to propagate. Take Figure 3 as an example. Suppose that the IDs of `a`, `b`, `c` are 3, 1, 2, respectively. We can construct a data dependency that is $1, 2 \Rightarrow 3$ when executing `a=b+c;`. When `loadUrl()` is executed, we will look at the ID of `g`, which is 3 because `g` is a reference of `a`, and tag it as an ωVar. Then the backward propagation will be triggered to search for IDs that 3 depends on, which are 1 and 2, and tag the corresponding variables as ωVars (i.e., `b` and `c`). The over-approximation problem can be much alleviated this way.

3.2.2 Backward Propagation Rules. The backward propagation rules are presented in the *Backward Propagation* section of Table 1. The five rules identify new ωVars that provide values to existing ωVars in assignment expressions.

3.3 Forward Propagation

The five forward propagation rules identify new ωVars that depend on existing ωVars in assignment expressions. Unlike backward propagation which relies on dependencies memorized during app execution, forward propagation is immediately built whenever a program statement that matches one of the rules is executed. For example, when $x = y\ op\ z$ is executed and one of y or z is an ωVar, x will be directly tagged as a new ωVar according to rule No.12. We also make forward propagation field-sensitive, i.e., when a newly created ωVar is a reference to an object/array/collection, its fields/elements will be recursively tagged as ωVars.

3.4 WebView-Specific Properties

Finally, we summarize the set of identified WebView-specific properties. At runtime, all variables including ωVars are uniquely represented by their assigned IDs. It is a key step to alleviate the over-classification problem of ωVars. However, the assigned IDs cannot be directly treated as the properties that tests should cover. It is because IDs are assigned in a non-deterministic way at runtime (i.e., it is increased by 1 whenever a new variable is initialized) so that even two identical tests may generate two different sets of IDs. Therefore, tests cannot be measured by the covered ID set. To generate a deterministic property set, we propose to take the *def* locations of ωVars ($Def_{\omega\text{Var}}$) found by backward propagation and the *use* locations of ωVars ($Use_{\omega\text{Var}}$) found by forward propagation as the property set. The *def* location of an ωVar is the place in a program where the ID of that ωVar is assigned or the place where a variable dependency on that ωVar is built. The *use* location of an ωVar is the place in a program where that ωVar is used (e.g., it is used as an argument of a method invocation). These locations are also uniquely indexed. To summarize, the WebView-specific property set that a test covers (P) is the union of the covered WebView API call sites (ωAPI), $Def_{\omega\text{Var}}$, and $Use_{\omega\text{Var}}$:

$$P = \omega API \cup Def_{\omega\text{Var}} \cup Use_{\omega\text{Var}} \quad (1)$$

3.5 Implementation

We implement the tagging/propagation logic by app instrumentation. We use Soot [18] and Esprima [9] to instrument all the Java and JavaScript statements that match the operation semantics in Table 1, respectively. We only instrument the JavaScript code located inside the asset folder of an app (which can be accessed by decoding an apk file via Apktool [5]) and the JavaScript code dynamically loaded by WebView APIs (e.g., the one loaded by `loadUrl()`). Other JavaScipt code (e.g., code in online websites) are not instrumented as they usually do not interact with the native-end of an app. The properties covered in the JavaScript code are sent to the native-end via bridge communication [7]. Together with the properties covered in the Java code, they are stored in the memory and will be saved to a file in the hard disk of an Android emulator or a real device every 500 ms. The file can be read by any test generation tools to retrieve coverage information. To also measure traditional coverage such as statement/method coverage, we also implement the logic of JaCoCo [13] into our instrumentation tool. The tool supports statement/method coverage on both Java and JavaScript code. To ease the variable tracking, we require the instrumented app to run on a customized Android OS. The tool and the OS are publicly available online [23].

4 TEST GENERATION

This section presents how we utilize the identified WebView-specific properties to guide test generation for WebViews. We implement the test generation procedure as a tool called ωTest, which aims to generate tests that maximize the coverage of WebView-specific properties of an Android app.

4.1 Overview

ωTest is a search-based test generation technique that generates a sequence of events to optimize a fitness function designed to meet the objective above. Intuitively, it keeps appending events (e.g., click a widget) to exercise app UI components [3] if new properties are frequently discovered. It leaves a UI component if existing properties are repeatedly covered or no properties are found after enough events are tried.

Algorithm 1 depicts our proposed test generation procedure. It takes an app under test (AUT) that has been instrumented according to Section 3 and a time budget as inputs, and outputs a set of covered WebView-specific properties (P) and a set of discovered WebView-induced bugs (ωBugs). ωTest iteratively generates events to explore the AUT within the given time budget (lines 4–16). It triggers an event in each iteration and monitors the manifested ωBugs (lines 5–6) as well as the covered properties (lines 7–8). ωTest decides whether to continually append new events (lines 10–14) or to leave the current activity A (line 16) by utilizing a fitness function defined in $continue()$ (line 9). When the foreground activity has a good fitness value, ωTest will similarly compute a fitness value for the current fragments-state A_F in A (an activity can render multiple fragments on the screen, so the set of fragments currently displayed forms a fragments-state). When the fitness is good or there is zero or one fragments-state (i.e., $|A| \leqslant 1$, which means there is no more fragments-state to switch) in A, ωTest will

[3]UI components includes activities [3] and fragments [11].

Algorithm 1: Test Generation Procedure (ωTest)

```
Input: An app under test (AUT) and a time budget Budget
Output: A set of covered WebView-specific properties P and a
        set of WebView-induced bugs ωBugs
1  P = ∅ ;                                   ▷ empty property set
2  ωBugs = ∅ ;                               ▷ empty bug set
3  Launch AUT ;                              ▷ initial event
4  while time ⩽ Budget do
5      if bug = foundBug() then
6          ωBugs = ωBugs ∪ bug ;             ▷ find a bug
7      if P_current != ∅ then
8          P = P ∪ P_current ;               ▷ merge covered properties
9      if continue(A) then                   ▷ A: the foreground activity
10         if continue(A_F) || |A| ⩽ 1 then
               ▷ A_F: current fragments-state in A
               ▷ |A|: number of fragments-states in A
11             e = selectAnEvent(UI);
12             executeEvent(e);
13         else
14             switchFragmentsState();
15     else
16         pressBack();
```

randomly pick an available event (e.g., pressing a button, scrolling a list, inputting text, rotating screens, etc.) according to the current UI hierarchy (line 11) and execute it (line 12). Otherwise, an event that can switch to another fragments-state is triggered (line 14). ωTest reports ωBugs according to pre-defined test oracles (lines 5–6) [35]. In the following, we explain how ωTest (1) decides whether to continue exploration from the current activity and fragments-state, and (2) identifies bugs with pre-defined test oracles.

4.2 Continual Appending of Events

ωTest determines whether to continue exploration according to a fitness value calculated to evaluate how good a state S is. The state S can be either an activity A or a fragments-state (A_F) in A. In this section, we will (1) define the fitness function, and (2) explain how $continue()$ makes decision based on the fitness value.

4.2.1 Fitness function. ωTest aims to cover as many unique properties as possible while minimizing the number of times that a property is covered more than once in order to diversify the properties covered during test generation. Test resources could be wasted if the generated tests keep visiting properties that have already been covered many times. In addition, if a state's function is adequately explored by enough events but no WebView-specific properties are found, fewer test resources should be spent on that state. Considering these factors and given a state S, the fitness function $f(S)$ is defined as:

$$f(S) = w \times f1(P_S, t_S) + (1 - w) \times f2(N_S, c_S) \tag{2}$$

where P_S is the set of properties covered when exploring S, t_S is the number of times that new properties are found in S, N_S is the number of events spent on S, c_S is the number of times that

code coverage increases in S, and w is a weight that balances two sub-fitness functions $f1()$ and $f2()$.

$f1(P_S, t_S)$ is inversely proportional to the frequency of the properties in P_S. Suppose P_S is represented as $\{p_1, p_2, ..., p_m\}$ and the number of times that the properties in P_S are covered is represented as $\{n_1, n_2, ..., n_m\}$. With this representation, $f1(P_S, t_S)$ is defined as:

$$f1(P_S, t_S) = \frac{\sum_{i=1}^{m} max(0, 1 - (\frac{n_i/t_S}{\alpha})^{\beta})}{|P_S|} \quad (3)$$

$f1()$ ranges between 0 and 1 and a higher value indicates a better fitness. The value of $f1()$ is high when new properties are frequently covered because in this case, the number of unique properties ($|P_S|$) and the number of times that property coverage increases (t_S) will be higher, and the frequency of each covered properties (n_i) will be relatively lower. The value of $f1()$ decreases when the covered properties are repeatedly covered (thus leading to higher n_is). Two parameters α and β control the decrease rate of $f1()$, which are set as e and 2, respectively.

To effectively cover diverse WebView-specific properties in a limited time budget, we use $f2()$ to limit the number of events spent on a state to search for uncovered properties. We rely on the code coverage information to compute $f2()$. Intuitively, if no more code coverage increase is observed after enough events are spent on a state, that state should not waste any more test resources afterwards. With this design, $f2(N_S, c_S)$ is defined as:

$$f2(N_S, c_S) = max(0, 1 - (\frac{N_S/(c_S)^{r(c_S)}}{\epsilon})^{\theta}) \quad (4)$$

Similarly, $f2()$ also ranges between 0 and 1. A higher value of $f2()$ indicates better fitness. When the number of times that code coverage increase observed in S is high (i.e., a higher c_S), more events N_S can be allocated to S before $f2()$ returns a relatively small value. $r(c_S)$ is a function that returns a value slightly smaller than 1. It is defined as $r(c_S) = 1 - 0.001 * c_S$ to prevent the test from being stuck in a state whose code coverage is increased too many times. Two parameters ϵ and θ also control the decrease rate of $f2()$, which are set to 8 and 2, respectively.

When no properties are covered in a state, we only use $f2()$ to guide testing. When WebView-specific properties are covered in a state S, we make the fitness of S (i.e., $f(S)$) biased to $f1()$ by setting the weight w in Equation 2 as 0.7.

4.2.2 Continue condition. The function *continue*() (line 9 & line 10) makes decision based on the fitness value. It returns true if the following condition is satisfied:

$$Rand(0, 1) < min(0.9, f(S)) \quad (5)$$

where $Rand(0, 1)$ returns a random number between 0 (inclusive) and 1 (exclusive). Intuitively, $f(S)$ can be treated as the probability that Condition 5 is satisfied. A higher $f(S)$ will make *continue*() more likely to return true. Therefore, ωTest will have a higher probability to continue exploration from a state with good fitness.

4.3 Oracle

Test oracles (line 5) are designed to detect WebView-induced bugs based on *WebView-induced crashes* and the *lifecycle misalignment criteria* proposed by ωDroid [35]. We identify a crash as WebView-induced if the crash results from an event executed on an element in the websites loaded by WebViews. The lifecycle misalignment criteria were designed to detect UI inconsistencies of a WebView before and after executing certain lifecycle events that make an activity restart (e.g., rotate screen). The assumption is that the displayed web page should be consistent before and after an activity restarts. For instance, the entered information of a form on a web page should not be lost after a device orientation change. Otherwise, the end-user has to re-enter the information in the form.

4.4 Implementation

ωTest is built on top of Appium [6], a test automation framework for mobile apps. ωTest obtains the UI hierarchy using the UiAutomator2 driver [19] facilitated by Appium. ωTest retrieves the covered properties by reading the files that record coverage information dumped by the instrumented AUT (Section 3.5) through Android's `adb` debugging tool [4]. Following previous work [34, 48, 56], we set a 200 ms delay between events. To switch to other fragment-states (line 14 in Algorithm 1), ωTest remembers the events that can lead to the switching of fragment-states during testing. One of them (including not-yet-triggered events, events that can open menus/navigation-drawers, etc.) will be picked if ωTest decides to switch again. ωTest is available online [23].

5 EVALUATION

In this section, we apply ωTest to test real-world Android apps. We evaluate it by studying three research questions:

- **RQ1:** Can ωTest effectively explore WebView behaviors? Compared with baseline methods, can ωTest achieve higher WebView-specific property coverage?
- **RQ2:** Can ωTest effectively detect WebView-induced bugs?
- **RQ3:** Compared with other coverage criteria, what is the bug-exposing capability of the WebView-specific property coverage criteria? Is covering more WebView-specific properties helpful to detect more WebView-induced bugs?

5.1 Evaluation Subjects

Our evaluation subjects contain 44 open-source Android apps and 30 closed-source Android apps. They are listed on our website [23].

Collecting open-source apps: We collected open-source apps from F-Droid [10], a popular open-source app hosting site, and the apps used in ωDroid [35]. We filtered the apps following the selection criteria below. An app was excluded if it satisfies one of the following conditions: (1) the app does not use WebViews in its application code; (2) the app does not have any commits within the past three months at the time when we conducted experiments; (3) the app's main functions cannot be reached in a fully automated way (e.g., requires login, remote devices or resources); (4) the app cannot be installed on Android emulators; (5) the app only uses WebViews to display simple pages such as About/License/Advertisement; (6) the app is a toy app (e.g., proof-of-concept apps) or a duplicate of the others. These criteria enable us to select well-maintained apps with non-trivial use of WebViews. Following the criteria and excluding the apps that cannot run after instrumentation, 44 apps were selected as our experiment subjects. Among them, 32 apps can

be found on Google Play. These apps are large-scale (*avg* over 100 Kloc), well-maintained (*avg* over 3k revisions), highly rated (*avg* 4/5), and diverse (covering 13 categories).

Collecting closed-source apps: We collected closed-source apps using a Google Play crawler Raccoon [16]. We downloaded the most-popular apps according to the rankings given by AppBrain [12, 14]. We followed criteria (1), (3), (4), (5) adopted in the previous paragraph when collecting the subjects. We stopped collecting until we successfully instrumented 30 apps. These 30 apps cover 15 categories and have over 2.84 billion downloads in total.

5.2 Experiment Setup

5.2.1 Baselines. To study the RQs, we selected baseline methods whose tools are publicly available and can run on Android 10. We compared ωTest with the following four baselines, including one state-of-the-art WebView test generation technique and three state-of-the-art general-purpose Android test generation techniques.

- **ωDroid:** ωDroid [35] is a Monkey-based random test generation technique that uses specially designed oracles to detect WebView-induced bugs. The oracles are also adopted by ωTest (Section 4.3).
- **Q-Testing:** Q-Testing [48] is a reinforcement learning-based general-purpose Android test generation technique. During app exploration, Q-Testing is more likely to pick an event that is expected to obtain higher accumulative rewards (i.e., discover new UI states). Whether two UI states are similar or different is determined by a trained neural network.
- **ComboDroid:** ComboDroid [56] is a general-purpose Android test generation technique. Its core idea is to generate a long event sequence (a combo) by combining multiple short event sequences (use cases). ComboDroid tends to combine two use cases where the latter one uses the data written by the former one so as to maximize data-flow diversity. ComboDroid provides a fully-automated variant and a semi-automated variant, and we use the former one. We set the modeling time, a parameter required by ComboDroid, to 30 minutes by following their recommendations.
- **Fastbot2:** Fastbot2 [43] is a general-purpose model-based test generation product from ByteDance [8]. It builds a probabilistic activity-event transition model and uses reinforcement learning to assist event selection. Fastbot2 uses APE [34] and has been deployed at ByteDance for two years.

5.2.2 Ablation Study. To evaluate the necessity of including ωVars in guiding test generation, we additionally make one more baseline.

- **ωTest-API:** ωTest-API differs from ωTest in that it only considers WebView API call sites in the fitness function. This baseline evaluates whether using WebView API call sites alone is adequate to guide WebView testing.

5.2.3 Coverage Calculation. To measure the effectiveness of exploring WebView behaviors, we measure the property coverage achieved by ωTest and the baselines. In traditional test generation studies, a coverage percentage (the covered properties over the total number of available properties) is usually used. However, unlike program statements/methods, the complete set of properties is difficult to obtain in our problem since it requires cross-language data flow analysis that is both sound and complete. For fair comparisons, we define the total property set P_{all} for an app as the union of the properties covered by ωTest and the baselines when finishing testing. Then the coverage of a method for an app is defined as:

$$Cov_{method} = \frac{|P_{method}|}{|P_{all}|} \tag{6}$$

5.2.4 Experiment Environment. We ran experiments on Android emulators running Android 10. We choose Android 10 to balance the OS popularity (it is the second most popular Android OS version when we were conducting experiments) and the number of available baselines (e.g., ComboDroid can support up to Android 10). We followed recent works [34, 35, 48, 51, 53] and allocated one hour to test each app. Since the executions of ωTest and all the baselines are subject to some randomness, we repeated the experiments five times to mitigate randomness in the results. Under these settings, the complete property set of an app was further enlarged to the union of P_{all}s over the five rounds. The experiments on closed-source apps were conducted on a machine running CentOS Stream 8, powered by AMD Ryzen Threadripper PRO 3995WX 64-Cores and 512GB memory. The experiments on open-source apps were conducted on a machine running CentOS Stream 8, powered by AMD Ryzen Threadripper 3970X 32-Core Processor and 256GB memory. We ran 16 emulators in parallel on each machine.

5.3 Results for RQ1 & RQ2

In this section, we present the results of WebView-specific property coverage and the detected WebView-induced bugs achieved by each methods. When reporting the coverage results, we will exclude an app for a baseline if it cannot successfully test the app. ComboDroid requires to instrument an app before testing. It fails to instrument 7 open-source apps and 9 closed-source apps in our dataset. Q-Testing cannot successfully test 4 open-source apps and 3 closed-source apps because of multiple engineering issues (e.g., UiAutomator error, app launching failure, etc.). ωDroid also fails to run on 1 open-source app and 1 closed-source app. We excluded those apps when reporting the coverage results of ComboDroid, Q-Testing, and ωDroid. When reporting the results on the detected bugs, ComboDroid, Q-Testing, and Fastbot2 are not included because they are not equipped with the oracles to detect WebView-induced bugs, and therefore no such bugs can be detected.

Figure 4 and 5 illustrate the WebView-specific property coverage distributions and its average progressive improvements of different methods using box plots and line charts. In figure 4, each app's coverage achieved by a method is averaged over five rounds. In figure 5, the properties covered by a method on each app is accumulated from five rounds of experiments. The data behind these figures can be found on our website [23]. From the figures, we can see that ωTest and ωTest-API can significantly outperform the baseline methods on both open-/closed-source apps. In particular, ωTest can achieve 16%-36% higher coverage on average on open-source apps and 9%-33% higher coverage on average on closed-source apps according to Figure 4. ωTest also outperforms ωTest-API, which eliminates ωVars and simply considers WebView API call sites in the fitness function. Compared with ωTest-API, ωTest can increase the coverage by 4%(on average)/6%(on median) and 3%(on average)/8%(on median) on open-source apps and closed-source apps, respectively, according to Figure 4. Such results indicate that

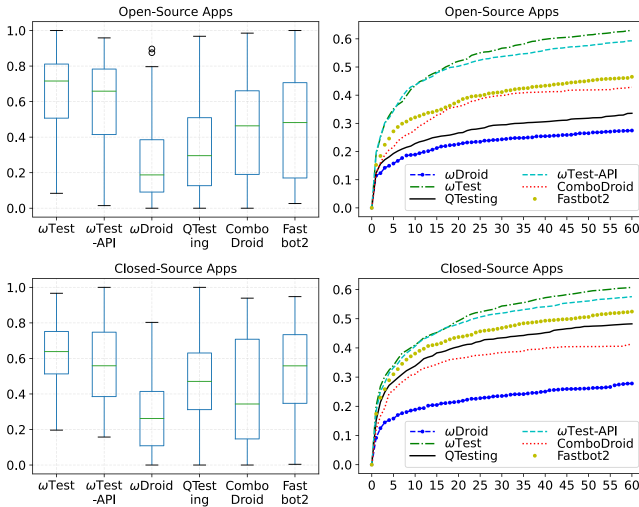

Figure 4: WebView-specific property coverage distributions and its average progressive improvements over 60 mins (The coverage of each app is averaged over 5 experiment rounds)

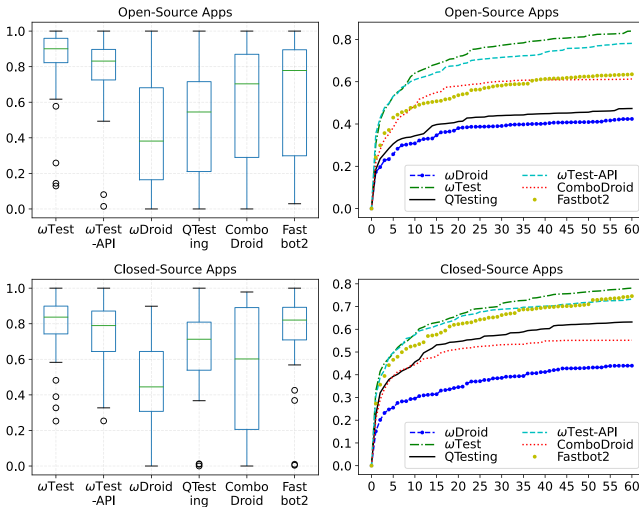

Figure 5: WebView-specific property coverage distributions and its average progressive improvements over 60 mins (The coverage of each app is accumulated from 5 experiment rounds)

using WebView API call sites alone is inadequate to achieve higher property coverage.

Table 2 shows the number of bugs detected by ωTest, ωTest-API, and ωDroid over the five rounds of experiments (ComboDroid, Q-Testing, and Fastbot2 do not have the oracles to detected WebView-induced bugs). In total, these 3 methods detected 27 bugs in 19 open-source apps and 13 bugs in 8 closed-source apps in these 5 runs. On average, ωTest detected more number of bugs than ωTest-API and ωDroid. ωTest failed to find three bugs that were detected by ωDroid in open-source apps because specific event types are not supported by ωTest currently. ωTest failed to find one bug

Table 2: Number of bugs detected by ωTest, ωTest-API, and ωDroid in 5 rounds of experiments

App Type	Method	R1	R2	R3	R4	R5	Avg	Total
Open-Source	ωTest	16	13	12	12	17	14	24
	ωTest-API	11	11	8	10	12	10.4	22
	ωDroid	8	7	7	5	7	6.8	11
Closed-Source	ωTest	8	10	8	6	8	8	12
	ωTest-API	8	5	3	6	5	5.4	11
	ωDroid	3	2	3	2	4	2.8	7

that was detected by ωDroid and ωTest-API in a closed-source app because we found the fitness value drops quickly on the activity that contains the buggy WebView in that app. Therefore ωTest decides to spend fewer events on that activity, leaving the bug undetected. We reported all the bugs detected in open-source apps to their corresponding GitHub issue trackers and provide the issue links on our website [23]. To comply with each app's contributing guide and license, we thoroughly tested each app and submitted a bug report in a proper format if the bug can be consistently reproduced. If a GitHub repository is archived or the detected bug has already been fixed in newer versions of the app, we provide the reproducing steps on our website [23]. We also provide the reproducing steps for bugs detected in closed-source apps on our website [23]. Among the 22 submitted bug reports whose bugs were detected by ωTest, 13 bugs have been confirmed and 9 of them have been fixed.

During bug reproduction, we observed that covering ωVars is helpful in driving ωTest to explore diverse WebView behaviors, thus discovering more hidden bugs in different usage scenarios of WebViews. For example, ωTest consistently detected bugs in Notepad [15], which is a notes edit app that has over 10 million downloads. The app uses a WebView to display Frequently Ask Questions (FAQs). ωTest discovered that a user's reading progress would get lost after device rotation because the WebView refreshes the page after rotation. In another scenario, Notepad uses a WebView to display Privacy Policies (PPs). Exploring FAQs and PPs will cover the same set of WebView API call sites but different sets of ωVars. ωTest-API , which is solely driven by WebView API call sites, may consider these scenarios being adequately explored after a few events, thus missing the bug hidden in these scenarios.

In summary, ωTest can effectively explore WebView behaviors of Android apps. It can achieve higher WebView-specific property coverage and detect more number of WebView-induced bugs.

5.4 Results for RQ3

To demonstrate the bug-exposing capability of the WebView-specific property coverage criteria, we compare it with WebView API call site coverage criteria and code coverage criteria. Figure 6-7 show the coverage distributions and the progressive improvements of WebView API call cite coverage and code coverage, respectively. In addition to the number of bugs detected at the end of testing shown in Table 2, we plot the progressive improvements of the number of bugs detected by each method in Figure 8. The time that a bug is found by a method is the first time when it is detected. The time is averaged by a number between 1 to 5, depending on how many times that the bug is detected in the five rounds of experiments.

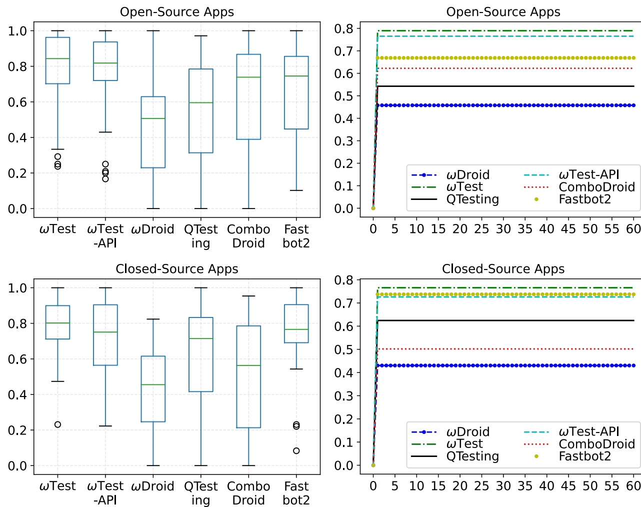

Figure 6: WebView API call site coverage distributions and its average progressive improvements over 60 mins (The coverage of each app is averaged over 5 rounds of experiments)

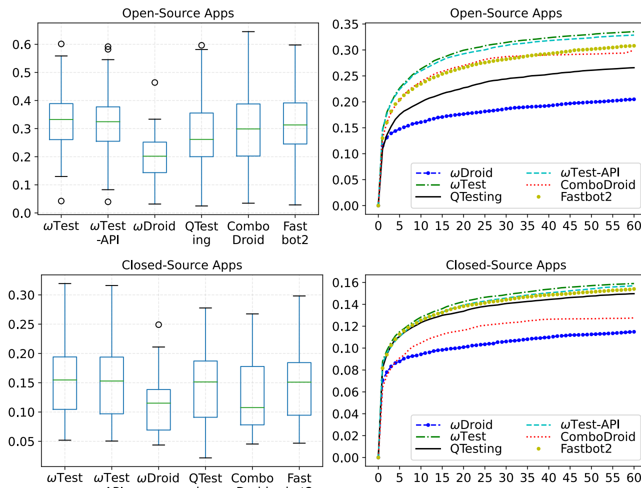

Figure 7: Code coverage distributions and its average progressive improvements over 60 mins (The coverage of each app is averaged over 5 rounds of experiments)

Figures 6 and 8 suggest that the coverage criterion based on WebView API call sites has a weak correlation with the number of detected bugs. The coverage quickly converges at the initial stage of the testing (before 5 minutes). However, many bugs are detected after 5 minutes. In addition, Figure 7 shows the code coverage achieved by ωTest and ωTest-API are similar on both open-source apps (avg 33.5% vs 32.9%) and closed-source apps (avg 15.9% vs 15.7%). However, Table 2 shows that ωTest detects far more bugs than ωTest-API, which sugguests that code coverage criterion also has a weak correlation with the number of detected bugs.

To quantitatively measure the correlations, we follow existing work [36, 38, 60] to compute Kendall correlations [37] between

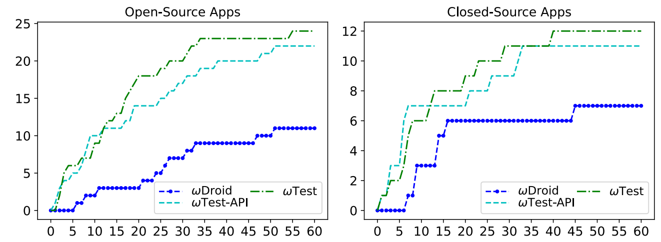

Figure 8: Progressive improvements of the detected bugs over 60 mins

the coverage computed base on a criterion and the number of detected bugs. Kendall correlation measures the correlations between two sets of data. A value between 0 to 1 indicates they are positively correlated (0-0.4 means a low correlation, 0.4-0.7 means a moderate correlation, 0.7-1 means a strong correlation). We compute the correlations between the average coverage on the buggy apps and the number of dectected bugs achieved by each method in each round at 5,10,15,...,60 minutes. The results for WebView-specific property coverage criteria, WebView API call site coverage criteria, and code coverage criteria on open-source apps are 0.7, 0.49, and 0.67, respectively. The results for WebView-specific property coverage criteria and WebView API call site coverage criteria on closed-source apps are 0.59 and 0.53, respectively. The result on code coverage is not reliable becasue closed-source apps are highly obfuscated. The code coverage are severely affected by third-party/system libraries, which cannot be effectively distinguished from application code. The results suggest that WebView-specific property coverage criteria has a moderate to strong correlations with the number of detected bugs.

In summary, the WebView-specific property coverage criteria have a stronger bug-exposing capability than the WebView API call site coverage criterion and the code coverage criterion. Covering more WebView-specific properties is helpful in detecting more WebView-induced bugs.

5.5 Limitations

We observed a limitation of ωTest when analyzing the experiment results. We found ωTest is not effective in reaching difficult-to-reach WebViews in an Android app. Some WebViews are deeply hidden in an app (e.g., a long activity stack is observed when the WebView is reached) so that ωTest may decide to leave a component too early, but actually more events deserve to be tried. Furthermore, we also observed that some WebViews cannot be reached until specific conditions are satisfied. For example, a weather app called RadarWeather [17] uses a WebView to display a weather map in an activity. The map is only available until users enters a city name in another activity. The fitness function adopted by ωTest is not able to model such information. In future work, we plan to extend ωTest with static analysis, which aims to identify the activities/fragments that are helpful to reach WebViews.

5.6 Threats to Validity

Our property coverage (Formula 6) may not reflect the "real" coverage since P_{all} can be different from the complete set of WebView-specific properties (there can be properties that were not discovered in our experiments). Obtaining the complete set in our problem is difficult because it requires cross-language data flow analysis that is both sound and complete. It also requires a sound and complete string analysis to predict the possible JavaScript code that is dynamically constructed at runtime. We mitigated this problem by approximating the complete set with the union of the properties identified by the six methods over five runs. Although our coverage results can be different from that computed using the complete property set, it provides a reliable evaluation of the relative performance achieved by different methods. This meets our evaluation goals and it is adopted by existing work [55, 60] in which the complete set is hard to obtain.

The conclusions drawn from our evaluation are affected by the representativeness of the selected subjects. To mitigate the threat, we selected 44 real-world open-source Android apps that are large-scale, well-maintained, and diverse in categories. We also included 30 closed-source apps that are selected from the most popular Android apps on the Google Play store. They have more than 2.8 billion downloads in total.

Randomness can affect our evaluation results as the algorithms of ωTest and all the baselines involve randomness. To mitigate the issue, we follow existing works [32, 34, 51] to repeat our experiments five times.

5.7 Discussion

The WebView-specific properties are computed via a set of propagation rules based on explicit data flows. Like existing work [28, 29], we choose not to propagate ωVars through implicit operations (e.g., control flows) to reduce noise. Except ωVars, coverage could be measured based on other types of properties such as call chains that involve WebView API calls, which seems sufficient for revealing the bugs. However, we choose not to use such call chains because (1) determining calls relevant to WebViews is difficult. Appending all calls around WebView API calls can include irrelevant methods. In comparison, the analysis defined in Section 3 can effectively determine the part of an app that is relevant to WebViews. (2) Determining the length of call chains is hard. Shorter chains may have a weak bug detection capability. For example, the covered calls for cs-bug1 and cs-bug2 [23] in Notepad [15] are the same if the length of the chain is smaller than 100 (50 calls before and after WebView API calls). Longer chains may increase the bug-detection capability, but can include many methods irrelevant to WebViews, complicating the analysis and increasing test cost.

6 RELATED WORK

6.1 WebView Study

WebView has attracted immense interest from research communities over the past ten years mainly because of the new security threats it brings to Android apps. Many attack models and mitigation solutions were proposed. For example, Bai et al. proposed BridgeTaint [27], a dynamic taint tracking technique targeting the WebView's bridge communication [7] to detect privacy leaks and cross-language code injection attacks. More recently, Yang et al. proposed EOEDroid [61], OSV-Hunter [63], and DCV-Hunter [62] that detect three kinds of new vulnerabilities resulting from WebView's event handlers, postMessage mechanism, and iframes/popups, respectively. Hu et al. proposed ωDroid [35] to detect WebView-induced lifecycle misalignment bugs. Our work complements existing work because we focus on effective test generation to examine WebView behaviors in an Android app.

6.2 Android Testing

A large number of test generation techniques have been proposed to test Android apps [30, 52, 58]. They can be classified into two major categories according to their test objectives. One is model-based test generation [26, 31, 32, 34, 43, 48, 51] whose objective is to discover diverse GUI states of an Android app. Intuitively, more discovered GUI states that "look different" means more app behaviors are explored. Another major category looks for program properties (e.g., program statements and branches) in an app's program structure and takes them as the test objectives [45, 46, 51, 56]. Although these existing program properties may be suitable for general-purpose testing, none of them are designed for testing WebViews in Android apps. In our paper, we design a novel coverage criterion based on WebView-specific properties to guide the test generation to effectively examine WebView behaviors in Android apps.

7 CONCLUSION

In this paper, we proposed a novel design of WebView-specific properties that can abstract WebView behaviors in Android apps. The property can be utilized to guide test generation to explore diverse WebView behaviors. Based on this idea, we devised ωTest, a test generation technique that maximizes the number of covered WebView-specific properties. Our evaluation results show that ωTest can effectively generate tests exercising diverse WebView behaviors and detect WebView-induced bugs. ωTest now only adopts crashing and lifecycle misalignment oracles. In the future, we plan to extend the oracles and leverage ωTest to detect more types of WebView-induced bugs. We will also study how to effectively reach difficult-to-reach WebViews in our future work.

8 DATA AVAILABILITY

We make all our data publicly available at https://richardhooooo.github.io/wTest/. The website includes (1) open-source and closed-source apps used in experiments, (2) coverage results achieved by ωTest and the baselines, (2) the links to the submitted bug reports and reproduction steps if the report is not able to be submitted, (3) the customized Android OS, (4) the tool ωTest and its guidance.

ACKNOWLEDGMENTS

The authors thank ISSTA 2023 reviewers. This work is supported by National Natural Science Foundation of China (Grant No. 61932021), Hong Kong Research Grant Council/General Research Fund (Grant No. 16211919), Hong Kong Research Grant Council/Research Impact Fund (Grant No. R503418), NSERC Discovery Grant RGPIN-2022-03744 DGECR-2022-00378, and Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562).

REFERENCES

[1] 2016. *Wikipedia Bug T149692*. https://phabricator.wikimedia.org/T149692
[2] 2022. *WeChat Mini-Programs Statistics*. https://www.statista.com/statistics/1155778/china-number-of-wechat-mini-program-daily-active-users/
[3] 2023. *Activity*. https://developer.android.com/reference/android/app/Activity
[4] 2023. *Android Debug Bridge*. https://developer.android.com/studio/command-line/adb
[5] 2023. *Apktool*. https://ibotpeaches.github.io/Apktool/
[6] 2023. *Appium*. http://appium.io
[7] 2023. *Building Web Apps in WebView*. https://developer.android.com/guide/webapps/webview
[8] 2023. *ByteDance*. https://www.bytedance.com/en/
[9] 2023. *Esprima*. https://esprima.org
[10] 2023. *F-Droid*. https://f-droid.org
[11] 2023. *Fragment*. https://developer.android.com/guide/fragments
[12] 2023. *Google Play Ranking*. https://www.appbrain.com/stats/google-play-rankings
[13] 2023. *JaCoCo*. https://www.jacoco.org/jacoco/trunk/doc/flow.html
[14] 2023. *Most Popular Android Apps*. https://www.appbrain.com/apps/popular/
[15] 2023. *Notepad Free*. https://play.google.com/store/apps/details?id=com.atomczak.notepat
[16] 2023. *Raccoon*. https://raccoon.onyxbits.de
[17] 2023. *RadarWeather*. https://f-droid.org/en/packages/org.woheller69.weather/
[18] 2023. *Soot*. https://github.com/Sable/soot
[19] 2023. *UiAutomator2 Driver*. https://github.com/appium/appium-uiautomator2-driver
[20] 2023. *WebView*. https://developer.android.com/reference/android/webkit/WebView
[21] 2023. *WeChat Mini-Programs*. https://developers.weixin.qq.com/miniprogram/en/dev/framework/MINA.html
[22] 2023. *Wikipedia Android App*. https://github.com/wikimedia/apps-android-wikipedia
[23] 2023. *wTest*. https://richardhooooo.github.io/wTest/
[24] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. 356–367. https://doi.org/10.1145/2976749.2978333
[25] Sora Bae, Sungho Lee, and Sukyoung Ryu. 2019. Towards understanding and reasoning about android interoperations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*. IEEE, 223–233. https://doi.org/10.1109/ICSE.2019.00038
[26] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 238–249. https://doi.org/10.1145/2970276.2970313
[27] Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. 2019. BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *IEEE Transactions on Information Forensics and Security* 14 (2019), 677–692. Issue 3. https://doi.org/10.1109/TIFS.2018.2855650
[28] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating dynamic data flow in commodity jvms. *Proceedings of the 2014 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2014)* 49, 10 (2014), 83–101. https://doi.org/10.1145/2714064.2660212
[29] Jonathan Bell and Gail Kaiser. 2015. Dynamic taint tracking for java with phosphor. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. 409–413. https://doi.org/10.1145/2771783.2784768
[30] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. IEEE, 429–440. https://doi.org/10.1109/ASE.2015.89
[31] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. 296–306. https://doi.org/10.1145/3293882.3330569
[32] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. 481–492. https://doi.org/10.1145/3377811.3380402
[33] Mohamed A El-Zawawy, Eleonora Losiouk, and Mauro Conti. 2021. Vulnerabilities in Android webview objects: Still not the end! *Computers & Security* 109 (2021). Issue C. https://doi.org/10.1016/j.cose.2021.102395
[34] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*. IEEE, 269–280. https://doi.org/10.1109/ICSE.2019.00042
[35] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A tale of two cities: How WebView induces bugs to Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, 702–713. https://doi.org/10.1145/3238147.3238180
[36] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 435–445. https://doi.org/10.1145/2568225.2568271
[37] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93. https://doi.org/10.2307/2332226
[38] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. 2017. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability* 66, 4 (2017), 1213–1228. https://doi.org/10.1109/TR.2017.2727062
[39] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. IEEE, 250–261. https://doi.org/10.1145/2970276.2970368
[40] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*. ACM, 829–844. https://doi.org/10.1145/3133956.3134021
[41] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang Wang. 2020. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS 2020)*. 569–585. https://doi.org/10.1145/3372297.3417255
[42] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011)*. ACM, 343–352. https://doi.org/10.1145/2076732.2076781
[43] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. https://doi.org/10.1145/3551349.3559505
[44] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE 2016)*. 653–656. https://doi.org/10.1145/2889160.2889178
[45] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 599–609. https://doi.org/10.1145/2635868.2635896
[46] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 94–105. https://doi.org/10.1145/2931037.2931054
[47] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST 2015)*.
[48] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. 153–164. https://doi.org/10.1145/3395363.3397354
[49] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. 2018. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2018)*. Springer, 25–46. https://doi.org/10.1007/978-3-030-00470-5_2
[50] Wei Song, Qingqing Huang, and Jeff Huang. 2020. Understanding JavaScript Vulnerabilities in Large Real-World Android Applications. *IEEE Transactions on Dependable and Secure Computing* 17 (2020), 1063–1078. Issue 5. https://doi.org/10.1109/TDSC.2018.2845851
[51] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*. ACM, 245–256. https://doi.org/10.1145/3106237.3106298
[52] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2021)*. 119–130. https://doi.org/10.1145/3468264.3468620
[53] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. 204–215. https://doi.org/10.1145/3460319.3464806

[54] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-force: understanding WebView malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, (ASE 2018)*. ACM, 714–725. https://doi.org/10.1145/3238147.3238221

[55] Valerio Terragni and Shing-Chi Cheung. 2016. Coverage-driven Test Code Generation for Concurrent Classes. In *38th ACM/IEEE International Conference on Software Engineering (ICSE 2016)*. 1121–1132. https://doi.org/10.1145/2884781.2884876

[56] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE 2020)*. 469–480. https://doi.org/10.1145/3377811.3380382

[57] Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, and Tao Huang. 2022. Characterizing and Detecting Bugs in WeChat Mini-Programs. In *44th International Conference on Software Engineering (ICSE 2022)*. 363–375. https://doi.org/10.1145/3510003.3510114

[58] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of android test generation tools in industrial cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*. IEEE, 738–748. https://doi.org/10.1145/3238147.3240465

[59] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: Identifying and Avoiding UI Exploration Tarpits. In *Proceedings of the 29th Joint Meeting on Foundations of Software Engineering (FSE 2021)*. https://doi.org/10.1145/3468264.3468554

[60] Zan Wang, Yingquan Zhao, Shuang Liu, Jun Sun, Xiang Chen, and Huarui Lin. 2019. Map-coverage: A novel coverage criterion for testing thread-safe classes. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE, 722–734. https://doi.org/10.1109/ASE.2019.00073

[61] Guangliang Yang and Jeff Huang. 2018. Automated generation of event-oriented exploits in android hybrid apps. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2018)*. https://doi.org/10.14722/ndss.2018.23236

[62] Guangliang Yang, Jeff Huang, and Guofei Gu. 2019. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 977–994. https://www.usenix.org/conference/usenixsecurity19/presentation/yang-guangliang

[63] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. 2018. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *2018 IEEE Symposium on Security and Privacy (SP 2018)*. IEEE, 742–755. https://doi.org/10.1109/SP.2018.00043

[64] Lei Zhang, Zhibo Zhang, Ancong Liu, Yinzhi Cao, Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang. 2022. Identity Confusion in WebView-based Mobile App-in-app Ecosystems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1597–1613. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-lei

[65] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, XiaoFeng Wang, Long Lu, and Haixin Duan. 2018. An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1183–1198. https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-xiaohan

[66] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. 2021. A Measurement Study of Wechat Mini-Apps. In *Proceedings of the 2021 ACM on Measurement and Modeling of Computer Systems*, Vol. 5. 1–25. Issue 2. https://doi.org/10.1145/3460081

Received 2023-02-16; accepted 2023-05-03