Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

TC 11 Briefing Papers

Vulnerabilities in Android webview objects: Still not the end!

Mohamed A. El-Zawawy[a,*], *Eleonora Losiouk*[b], *Mauro Conti*[b]

[a] *Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt*
[b] *Department of Mathematics, University of Padua, Italy*

ARTICLE INFO

Article history:
Received 26 January 2021
Revised 27 May 2021
Accepted 4 July 2021
Available online 10 July 2021

Keywords:
Android security
Taint analysis
Hybrid applications
Web view
Java script interfaces
Web view client,

ABSTRACT

WebView objects allow Android apps to render web content in the app context. More specifically, in Android hybrid apps (i.e., those having both Android code and web code) the web content can interact with the underlying Android framework through Java interfaces and WebViewClient objects. Thus, while rendering web content a hybrid app can execute malicious Javascript code that can access the sensitive data on the device, bypassing the sandbox model usually adopted by standalone browsers. Researchers already analyzed the security issues of WebView objects, by focusing on Javascript interfaces. However, we believe that there are other aspects related to the rendering of web content in Android apps, such as WebViewClient objects, that could lead to security issues.

In this paper, we introduce three new types of vulnerabilities related to WebView, that expose new attack surfaces concerning the most well-known vulnerability related to JavaScript interfaces. To detect these new types of vulnerabilities, we designed *WebVSec*, a static analysis system that relies on a set of custom inference rules, heuristically formalized. By designing *WebVSec* to detect also the vulnerability already described in the state-of-art, we were able to compare *WebVSec* with *BabelView* on a set of 2000 applications. *BabelView* was found not able to detect our new three types of vulnerabilities and also less precise and efficient in detecting the already known vulnerability. In particular, over the 2000 analyzed apps, *WebVSec* and *BabelView* identified 48 and 18 vulnerable apps, respectively. Among those, *WebVSec* found 20 apps having a specific type of vulnerabilities and 36 apps having another type of vulnerabilities, while *BabelView* found 11 and 0 apps, respectively. In terms of efficiency, *WebVSec* took 27.16 hours to analyze the whole set of 2000 applications against the 63.64 hours required by *BabelView*.

1. Introduction

The use of Android on mobile and tablet devices has grown over the past years. Android covers almost 75% of the mobile market share worldwide, while mobile devices almost 53% Statcounter (0000). The Android platform has recently introduced an approach to render web content in Android applications through the use of `WebView` components. `WebView` enables the displaying of web content and relies on WebKit for rendering pages. However, the wide use of `WebView` is mainly motivated by the introduction of hybrid applications, in which core source code is written in HTML and JavaScript through

* Corresponding author. .
E-mail addresses: maelzawawy@cu.edu.eg (M.A. El-Zawawy), elosiouk@math.unipd.it (E. Losiouk), conti@math.unipd.it (M. Conti).
https://doi.org/10.1016/j.cose.2021.102395

cross-platform tools, such as Apache Cordova Apache (0000). Currently, one-third of mobile developers use cross-platform tools for writing their app, while the rest of the developers use native tools hyb (2020).

The design model of `WebView` objects highly differs from the one adopted in standalone browsers. The latter adopts a sandbox mechanism that introduces an access control approach to prevent untrusted Javascript code from accessing sensitive data on the mobile device Ferri et al. (2010). Thus, in a standalone browser not only web pages are isolated from each other, but they are also isolated from the system. On the contrary, `WebView` objects use binding interfaces (through the `addJavascriptInterface` API) to allow Javascript code to access private data and system resources on the mobile device. Enabling the interaction between Javascript code and the underlying system breaks the security model used in standalone browsers. The malicious JavaScript can be part of any web page loaded in the `WebView`. Advertisement libraries used in the `WebView` may also allow malicious JavaScript to access private data. Overall, current `WebView` technologies are not robust enough to prevent vulnerabilities, such as Cross-Site Request Forgery, Cross-site scripting, and JSON hijacking Li et al. (2017); Luo et al. (2011, 2012); Mutchler et al. (2015). Consequently, attackers can manipulate `WebViewClient` listeners and access `WebView` interfaces through the injection of malicious JavaScript Fahl et al. (2012); Mutchler et al. (2015).

Previous works Rizzo et al. (2018); Yang et al. (2017) addressed the security issues raised by `WebView` interfaces by mainly focusing on the vulnerabilities involved in the usage of JavaScript interfaces. However, no previous work considered the security issues raised by `WebViewClients`.

In this paper, we identify three new types of `WebView` vulnerabilities, which are somehow related to the one already addressed by the state-of-art, but that introduce new attack surfaces. We heuristically designed seven rules that allow the detection of all such vulnerabilities and developed the $\mathcal{W}$eb$\mathcal{VS}$ec system: a static analysis tool that relies on our seven inference rules to detect `WebView` vulnerabilities. We evaluated precision, efficiency, and effectiveness of $\mathcal{W}$eb$\mathcal{VS}$ec over a set of 2000 applications and compared the results with *BabelView*, a state-of-the-art technique Rizzo et al. (2018). With our study, we prove that the current state-of-art works did not comprehensively address `WebView` technologies, thus failing in detecting our newly identified vulnerabilities. Moreover, $\mathcal{W}$eb$\mathcal{VS}$ec was found to overcome both in terms of precision and efficiency.

Contributions The contributions of this paper are as follows:

1. We identified three new types of vulnerabilities related to the Android `WebView` design model.
2. We heuristically designed seven inference rules to detect the existing and the newly found vulnerabilities.
3. We designed and developed $\mathcal{W}$eb$\mathcal{VS}$ec, a system that implements our rules to detect the above-mentioned vulnerabilities.
4. We experimentally compared $\mathcal{W}$eb$\mathcal{VS}$ec against *BabelView*, the state-of-the-art tool for `WebView` vulnerabilities detection. Over the 2000 analyzed apps, $\mathcal{W}$eb$\mathcal{VS}$ec and *BabelView* identified 48 and 18 vulnerable apps, respectively. Among those, $\mathcal{W}$eb$\mathcal{VS}$ec found 20 apps having a specific type of vulnerabilities and 36 apps having another type of vulnerabilities, while *BabelView* found 11 and 0 apps, respectively. In terms of efficiency, $\mathcal{W}$eb$\mathcal{VS}$ec took 27.16 hours to analyze the whole set of 2000 applications against the 63.64 hours required by *BabelView*.

Organization The rest of the paper is organized as follows. Section 2 presents the necessary background, while in Section 3, we illustrate previous works addressing vulnerabilities in Android `WebView` objects. Section 4 introduces the threat model we consider in this paper, while Section 5 presents the `WebView` vulnerabilities we studied. The system design and implementation of our solution (i.e., $\mathcal{W}$eb$\mathcal{VS}$ec) are presented in Section 6 and Section 7, respectively. Section 8 presents the results we achieved by applying both $\mathcal{W}$eb$\mathcal{VS}$ec to a datset of 2000 applications. We conclude the paper in Section 9.

2. Background

A `WebView` Developers (0000) is an Android system component enabling Android applications to render web pages and interact with web servers. It is possible to write and show HTML code inside an app via `WebView`. Listing 1 presents an example that embeds a browser in an Android app to display *Google* search engine. This is done in three steps: defining a `WebView` object (line 1), enabling the execution of JavaScript within the object (line 2), and using the API `loadUrl` to load a web page (line 3).

`WebView` enables many forms of interaction between Android applications and web pages. Among those, we focus on *running Java from Javascript* and *event monitoring*. The *running Java from Javascript* mode consists of Java code, belonging to the app, being executed by Javascript code, belonging to a web page. The *event monitoring* form, instead, refers to the events that are registered in callback methods of `WebViewClient` objects and executed in response to specific events that occur on the web page.

2.1. *Running java from javascript.*

The JavaScript code in a `WebView` can invoke the Java code of some classes belonging to the Android app where it is executed Developers (0000). In the rest of the paper, we refer to these classes as interface classes. Objects of interface classes can be associated with `WebView` objects through the `addJavascriptInterface` API. Then, the Javascript code of the `WebView` can invoke only the public methods of interface classes that are annotated by `@JavaScriptInterface`. The annotation prevents attackers from executing arbitrary methods of interfaces by using Java reflection API. The annotations are not necessary for applications targeting Android versions before 4.2 (SDK17) InfoSecurity (2013). This is so because `@JavaScriptInterface` annotation did not exist in versions before 4.2.

Listing 2 presents an example of interface class usage. In particular, the interface `webviewInterface` is first instantiated and bound to the `webview` object (line 1),

```
1  private WebView webview;
2  webview.getSettings().setJavaScriptEnabled(true);
3  webview.loadUrl("http://www.google.com");
```

Listing 1 – WebView Example.

```
webview.addJavascriptInterface(new webviewInterface()," 
    myInterfaceObj");
public class webviewInterface {
   @JavascriptInterface
   public void readDeviceId(){
      toastDeviceId(telephonyManager.getDeviceId());}

   private void toastDeviceId(String DeviceId) {
      Toast.makeText(getappContext(), DeviceId, Toast.
    LENGTH_SHORT).show();}}
```

Listing 2 – Interface Example.

```
<script type="text/javascript">
   function  Show()
   {
   myInterfaceObj.readDeviceId();
   }
</script>
```

Listing 3 – Example of Java invocation from JavaScript code.

previously created in Listing 1. The binding object is called `myInterfaceObj`. The JavaScript code of a web page loaded in `webview` can use `myInterfaceObj` to run the annotated method `readDeviceId()`, as shown in Listing 3. On the contrary, the method `toastDeviceId(String)` of `webviewInterface` can not be invoked since it not annotated. However, the invocation of an annotated method might cause the invocation of a non-annotated one, as shown in the code of `readDeviceId()`.

Listing 2 clearly shows the extension of Same Origin Policy (SOP) applied to the context of `WebView`. Generally speaking, SOP refers to the access control policies applied to resources that might be shared between two web pages. In particular, a web page can access the data of another web page if both share the same origin. In the context of Android `WebView`, such access control policies are not applied to sensitive data stored in the underlying OS. Thus, any Javascript code can access such data. This is justified as follows. Suppose that `addJavascriptInterface` is used to attach an interface *I* to a WebView *W*. This has the consequence of allowing all pages loaded in *W* to call methods of *I*, and hence access the same sensitive data accessible from *I*. Therefore, in this case, SOP is defeated as web pages from one origin are allowed to affect those from others.

2.2. *Event monitoring.*

Android provides an important class called `WebViewClient`. The methods (listeners) of this class enables apps to listen and respond to events occurring within `WebView`. We will call these methods `WVC-listeners` all over the paper. Each `WVC-listener` is triggered upon the occurrence of a specific `WebView` event assigned to it. `WVC-listeners` can access information of their events and update the sequence of `WebView` event actions. According to Android implementation of the `WVC-listeners`, they do nothing. Therefore, it is up to the programmer to override any `WC-listener` with any required specific implementation.

Utilizing `WebViewClient` concepts requires binding an object of `WebViewClient` to a `WebView` object via the API `setWebViewClient`. An example of this is given in line 1 of Listing 4. The same listing shows overriding `WVC-listeners` such as `onPageStarted`, which is triggered when the page has started loading and `onPageFinished` which is triggered when the page has finished loading. The modified `onPageStarted` reads the device ID and stores it in the field `DeviceId`. Then `onPageFinished` toasts the field value. `WC-listeners` include the following callback methods: `shouldOverrideUrlLoading`, `doUpdateVisitedHistory`, `onFormResubmission`, `onLoadResource`, `onPageCommitVisible`, and `onReceivedClientCertRequest`.

3. Related work

One of the Android security directions that have been receiving intense research efforts is `WebView` vulnerabilities Luo et al. (2011). The main cause of existing vulnerabilities over the JavaScript Bridge is the fact that traditional security models of web context Georgiev et al. (2014) conflict with the

```
webview.setWebViewClient(new WebViewClient(){
 private String DeviceId;
 @Override
 public void onPageFinished(WebView view,String url){
   Toast.makeText(getappContext(), DeviceId, Toast.
    LENGTH_SHORT).show();}
 @Override
 public void onPageStarted(WebView view, String url, Bitmap
    favicon){
   DeviceId = telephonyManager.getDeviceId();}});
```

Listing 4 – WebViewClient Example.

lack of privilege isolation Jin et al. (2015). This conflict was partially treated by *NoFrak* Georgiev et al. (2014) that extended SOP to cover local resources. The extension idea was employed again in *MobileIFC* Singh (2013) to resolve access control between the web and mobile frameworks.

Many types of possible attacks against Android `WebView` related to the vulnerabilities treated in our work were presented by Luo et al. Luo et al. (2011). Among the previous works, the two ones that are closer to $\mathcal{W}$eb$\mathcal{V}\mathcal{S}$ec are *BabelView*Rizzo et al. (2018) and *BridgeScope* Yang et al. (2017). Relying on static information analysis, *BabelView* aims at evaluating the potential impact of JavaScript injection attacks in `WebView`. The main idea is to instrument applications to introduce possible attacker behaviors. However, *BabelView* does not cover all the vulnerabilities we will illustrate in this paper. *BridgeScope* aims at assessing JavaScript interfaces using static analysis. Similar to our work and to *BabelView*, *BridgeScope* evaluates possible paths to and from methods of interface classes. *BridgeScope* relies on a custom flow analysis. The main limitation of BridgeScope is the lack of analysis of listener methods of `WebViewClient` classes. With respect to *BridgeScope* and *BabelView*, $\mathcal{W}$eb$\mathcal{V}\mathcal{S}$ec focuses on a wider scope concerning the detection of `WebView` vulnerabilities.

Chin et al. Chin and Wagner (2013) studied the relationship between file-based cross-zone scripting attacks and excess authorization, on one side, and `WebView` vulnerabilities, on the other side. JavaScript interface implementations that misuse or do not consider Transport Layer Security (TLS) were considered vulnerable by Neugschwandtner et al. in Neugschwandtner et al. (2013). Their work linked requiring privacy critical permissions to `WebView` vulnerabilities. Without considering the expose of JavaScript interfaces, unsafe navigation and content retrieval in `WebView` are treated in Mutchler et al. (2015) by Mutchler et al. Middleware frameworks of third-party hybrid applications are investigated in Georgiev et al. (2014). Yang et al. in Yang et al. (2019) presented a `WebView` vulnerability related to web iframe/popup, namely thenon Differential Context Vulnerabilities (DCVs). These vulnerabilities show that iframe/popup can open holes on Android `WebView` defense mechanisms to gain privileges. DCV facilitates performing phishing attacks, destroying the integrity of web messaging, and accessing sensitive functionalities. In Hu et al. (2018), Hu et al. introduced Android bugs related to `WebView` interaction mechanisms. Their work studied the causes and consequences of these bugs. Unlike our current paper, above-reviewed papers did not study security issues related to `WebViewClient` objects, accompanying `WebView`.

Previous works Hassanshahi et al. (2015); Jin et al. (2014) have focused on the techniques for injecting malicious code into `WebView`. HTML5-based hybrid applications deal with many types of objects. This enables a wide range of cross-site-scripting attacks Jin et al. (2014). These attacks are enabled when the user loads the malicious page within the `WebWiew`. The other type, Web-to-Application injection attacks (W2AI) is based on Intent hyperlinks reacting to link clicking in the browser Hassanshahi et al. (2015). *Draco* Tuncay et al. (2016) is a framework for uniform and fine-grained access control of JavaScript execution in `WebView`.

In Li et al. (2017), Li et al. introduced a new type of attacks, namely Cross-App `WebView` infection, that is caused by the possibility of `WebView` to send navigation requests to another one via Intents and URL schema. Cross-App `WebView` infection can result in unauthorized execution of app components and enables multi-app colluding attacks. Origin Stripping Vulnerabilities (OSV) are triggered upon calling the `window.postMessage` API when the identity of the sender is not distinguishable and source origin is not safely obtainable. This is true as well for Android hybrid applications. A new technique, *OSV-Hunter* was introduced in Yang et al. (2018) to detect OSV.

4. Threat model

The threat model studied in this paper focuses on the ways in which Android applications may be attacked by malicious web pages. The applications are assumed to be benign ones that are serving web applications. Our model considers first-party applications (owned by the served web applications) and third-party applications (that are not owned by the served web application injecting malicious JavaScript code). An overview of our attack scenario is presented in Fig. 1. The scenario assumes that the attacker tries to deceive the victim app to load a malicious web page into the app `WebView`. Then, malicious JavaScript code of the loaded web page launches attacks on app `WebView` as follows. Once loaded into the `WebView`, the malicious JavaScript code can run the `WebView` interface methods (annotated with `@JavaScriptInterface`) to obtain the sensitive data. Allowing the malicious JavaScript to run the interface methods is visualized in the figure as

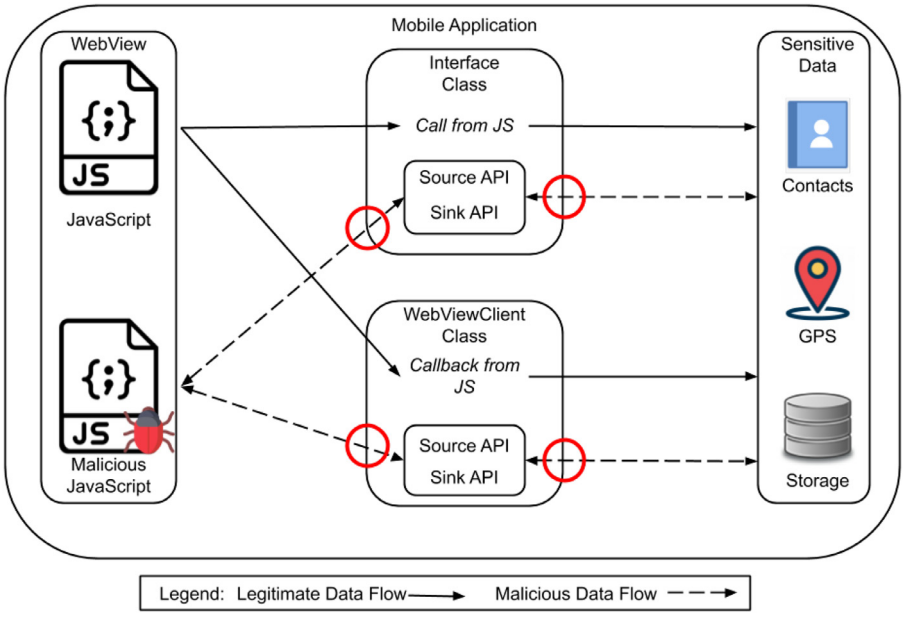

Fig. 1 – Threat model.

`WebView` holes facilitating leakage of sensitive data. Malicious data flow is marked with circles.

Our model also considers threats resulting from binding `WebViewClient` classes to WebView ones (through the `setWebViewClient` API). Via manipulating `WVC-listeners`, attackers can gain access to sensitive data. One way of this manipulation is by letting the malicious JavaScript code generate webpage events (e.g., page reload) that trigger `WVC-listeners`. Therefore, it may become possible to read sensitive data via one of `WVC-listeners` and send this data out via another `WVC-listener`. It is also possible to design an attack in which an annotated method of an interface cooperates with a `WVC-listeners`. Up to our knowledge, attacks that involve `WVC-listeners` are not treated by state-of-the-art techniques, such as *BabelView* Rizzo et al. (2018) and *BridgeScope* Yang et al. (2017).

Our thread model identifies high-impact WebView vulnerabilities in Android applications. This is done via evaluating the privileges that an attacker would gain by:

- injecting arbitrary JavaScript code into HTML or scripts loaded in a WebView.
- manipulating the `WVC-listeners` to abuse the source and/or sink APIs included in them.

The threat model of this paper assumes that the JavaScript code is arbitrary and fully controlled by the attacker. Hence, we do not need to analyze JavaScript code. We also do not need to worry about the manipulation techniques of `WVC-listeners`. Abusing methods of JavaScript interfaces and `WVC-listeners` requires the attacker to know their names. A simple reverse engineering process can provide such information.

5. New webview vulnerabilities

This paper considers four types of vulnerabilities that attackers can utilize to launch WebView attacks through injected JavaScript code and `WVC-listeners` manipulation. Here, we first formally present the four types of vulnerabilities and then provide an illustrative example, taken from a testbed Android application we developed.

Definition 1 and Definition 2 introduce execution paths that play a vital role in vulnerability formalization.

Definition 1. An *interface path* is an execution path that is embedded in an annotated interface method. The set of all interface paths in an app is denoted by $\mathcal{P}_I$.

Definition 2. A *WebViewClient path* is an execution path that is embedded in a `WVC-listeners`. The set of all WebViewClient paths in an app is denoted by $\mathcal{P}_W$.

Type 1 Vulnerability (Interface-Interface Vulnerability). An Android application has a type 1 vulnerability if it has n execution paths, $P_1, \ldots, P_n$, such that:

a. $\forall 1 \leq k \leq n,\ P_k \in \mathcal{P}_I$,
b. P_1 has a source API that reads a sourced value, s.
c. $\forall 1 \leq k \leq n,\ P_k$ may move the sourced value s to other variables.
d. P_n has a sink API that leaks one of the variables that has the sourced value s.

In other words, an Android app has a type 1 vulnerability if it has an ordered sequence of $n (n \geq 1)$ interface paths such that: the first path reads sensitive data; all paths may move the read data to other containers (registers); the last path leaks the sensitive data from any of its containers.

Table 1 – Description and detection of the four vulnerabilities addressed by WebViewSec in the state-of-the-art solutions, i.e., BabelView and BridgeScope.

Vulnerability	Vulnerability Description	Vulnerability Detection
Type 1	*BabelView* Rizzo et al. (2018)	*BabelView* Rizzo et al. (2018)
Type 2	-	-
Type 3	-	-
Type 4	-	-

Type 2 Vulnerability (Interface-WebViewClient Vulnerability). This type has the same definition as type one, except that the condition (a) above is replaced by

$$((P_1 \in \mathcal{P}_I \wedge P_n \in \mathcal{P}_W) \vee (P_1 \in \mathcal{P}_W \wedge P_n \in \mathcal{P}_I)) \wedge 2 \leq k \leq n-1,\ P_K \in \mathcal{P}_I \cup \mathcal{P}_W.$$

In other words, the first or the last path (but not both) has to be a WebViewClient one. Moreover, any of the remaining paths can be an interface or a WebViewClient one.

Type 3 Vulnerability (WebViewClient-WebViewClient Vulnerability). This type has the same definition as type one, except that the condition (a) above is replaced by

$$\forall 1 \leq k \leq n,\ P_k \in \mathcal{P}_W.$$

In other words, all paths have to be WebViewClient ones.

Type 4 Vulnerability (Reverse Vulnerability). An Android application has this type of vulnerability if it has a path $P \in \mathcal{P}_I \cup \mathcal{P}_W$ such that:

a. P has a sink API (i.e., K) at some program point (i.e., u) that leaks the value of some variable (i.e., v).
b. P has a source API (i.e., S) at some program point (i.e., w) such that $w > u$ (i.e., the program point w comes before the program point u) and S reads the value stored in v.

In other words, a type 4 vulnerability occurs if there is an interface or WebViewClient path that has a source API preceded by a sink API. For this vulnerability type, there also must be a data flow path from the source API to the sink API using a global variable. For manipulating this vulnerability type, the attacker should run the path twice: in the first run the source gets the sensitive information and in the second run the sink leaks the information. Therefore, this vulnerability is tricky and, up to our knowledge, has never been revealed in the literature.

As shown in Table 1, up to our knowledge, the state-of-the-art tools neither illustrate the type 2, type 3, and type 4 vulnerabilities nor propose techniques to detect them in Android applications. One of the contributions of this paper is to fill this gap.

5.1. *Half vulnerabilities.*

A WebView vulnerability requires a path that has a source method reading sensitive data into a variable. If this variable keeps its content unchanged till the end of its hosting execution path, we call the variable a *sourced variable*. This path constitutes half the vulnerability or the leaking process. The other logical requirement is a path that has a sink method sending out sensitive data included in a variable. If this variable does not get updated in its execution path, we call this variable a *sinked variable*. This last path constitutes the other half of the vulnerability. The sensitive data reaches the sinked variable from a sourced variable during data flow.

A reverse vulnerability occurs when two half vulnerabilities of different types exist in the same method. Furthermore, the sourced variable of one of the half vulnerabilities is the sinked variable of the other half vulnerability.

5.2. *Examples of vulnerabilities.*

In the code example shown in Listing 5, we provide examples of the four vulnerability types. The listing assumes a WebView object `ThreatWebview` that is associated with a JavaScript Interface (`ThreatwebviewInterface`) and a WebViewClient objects. Invoking the method f3() using `myThreatInterfaceObj.f3()` causes the device ID to be read into the global variable `id`. A following invocation to f4(), using `myThreatInterfaceObj.f4()`, would leak the device ID into the device memory (via the editor which is SharedPreferences.Editor editor object).

Examples of different types of vulnerabilities from Listing 5 are the following, where $\mathcal{P}(m)$ denotes the main execution path of the method m:

- Type 1: Executing the path of the method $f3$ followed by the path of the method $f4$ is an example of Type 1; this example is expressed in our notation as $[\mathcal{P}(f3), \mathcal{P}(f4)]$. Other examples of this vulnerability type are $[\mathcal{P}(f1)]$, and $[\mathcal{P}(f3), \mathcal{P}(f1), \mathcal{P}(f4)]$.
- Type 2: an example of this vulnerability type is $[\mathcal{P}(onLoadResource, \mathcal{P}(f4))]$.
- Type 3: an example of this vulnerability type is $[\mathcal{P}(onLoadResource), \mathcal{P}(onPageFinished)]$.
- Type 4, an example of this vulnerability type is $[\mathcal{P}(onPageStarted)]$.

6. Design of our tool: $\mathcal{W}$eb$\mathcal{VS}$ec

This section presents the design of $\mathcal{W}$eb$\mathcal{VS}$ec, our proposed system for detecting WebView vulnerabilities presented in Section 5. $\mathcal{W}$eb$\mathcal{VS}$ec relies on two heuristically designed sets of inference rules: the first set abstracts the Dalvik byte code of Android applications, while the second 1 aims to detect the four vulnerabilities addressed in this paper. Table 2 presents semantics of notation used in this section.

Fig. 2 illustrates the general workflow of $\mathcal{W}$eb$\mathcal{VS}$ec, which steps are described below.

6.1. *Decompilation (step 1).*

$\mathcal{W}$eb$\mathcal{VS}$ec receives in input an APK file, which corresponds to the app under investigation. The decompilation aims at reverse-engineering the APK file to obtain its manifest and dex files, on top of which $\mathcal{W}$eb$\mathcal{VS}$ec works.

6.2. *Identification of interface and webviewclient classes (step 2).*

The objective of this module is to identify $\mathcal{M}_{\mathcal{A}}, \mathcal{I}_c, \mathcal{I}_{c17}, \mathcal{I}_u, \mathcal{W}$ and $\mathcal{W}_u$. The identification of interface methods needs

```
int id;
ThreatWebview.addJavascriptInterface(new
    ThreatwebviewInterface(),"myThreatInterfaceObj");
public class ThreatwebviewInterface {
   @JavascriptInterface
   public void f1(){
      f2(telephonyManager.getDeviceId());}
   private void f2(String s1) {
      id = s1;
      f3();}
   @JavascriptInterface
   public void f3() {
      editor.putString("DeviceId", id).apply();
      id=telephonyManager.getDeviceId();}
   @JavascriptInterface
   public void f4(){
      f5(); }
   private void f5(){
      editor.putString("DeviceId", id).apply();}}

ThreatWebview.setWebViewClient(new WebViewClient(){
   private String idWebClient;
   @Override
   public void onPageFinished(WebView view, String url){
      editor.putString("DeviceId", id).apply();}
   @Override
   public void onPageStarted(WebView view, String url,
    Bitmap favicon){
      editor.putString("DeviceId", id).apply();
      id = telephonyManager.getDeviceId();}
   @Override
   public void onLoadResource(WebView view, String url){
      id = telephonyManager.getDeviceId();}});
```

Listing 5 – Interface Example.

Table 2 – Notation Summary.

Notation	Description
$\mathcal{M}_A$	Set of methods annotated with `JavascriptInterface`.
$\mathcal{I}_c$	Set of interface classes of $\mathcal{M}_A$.
$\mathcal{I}_{c17}$	Set of interface classes for applications whose minSdk ≤ 17.
$\mathcal{I}_u$	Set of classes using WebView.
$\mathcal{W}$	Set of WebViewClient classes.
$\mathcal{W}_u$	Set of classes using WebViewClient.
$\mathcal{C}$	Set of all classes treated by the analysis.
$\mathcal{M}_I$	Set of all interface methods treated by the analysis.
$\mathcal{M}_W$	Set of all WebViewClient methods treated by the analysis.
$\mathcal{M}_a$	Set of all access methods in interface and WebViewClient classes.
$\mathcal{M}_r$	A map from methods to their used number of registers.
$\mathcal{M}$	Set of all methods treated by the analysis.
th_m	An upper bound on the number of methods to be analyzed.
p	An execution path of a method.
id_p	ID of a path p.
id_m	ID of a method m.
$\mathcal{L}(p)$	List of abstracted instructions of the path p.
$\mathcal{L}id(p)$	List of IDs of method paths called in a path p.
$\mathcal{N}$	Set of Dalvik instruction names.
$\mathcal{O}$	Set of operand lists of Dalvik instructions.
$\mathcal{I}nst$	Set of pairs $(\mathcal{N}, \mathcal{O})$ representing Dalvik instructions.
$\mathcal{I}nstAbs$	Set of abstracted Dalvik instructions.

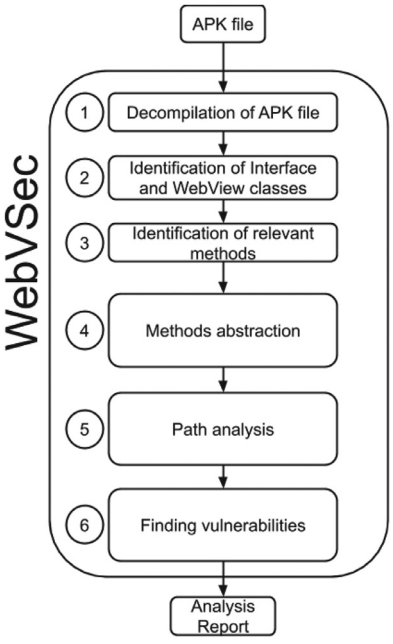

Fig. 2 – Overview of $\mathcal{W}$eb$\mathcal{V}\mathcal{S}$ec.

to consider both applications with *Min Sdk* greater than or equal 17 and applications with *Min Sdk* lower than 17: the former have interface methods annotated with `@JavascriptInterface`, while the latter does not. However, for the latter, we found out that it is common to include annotated and also non-annotated interface methods.

6.3. *Identification of relevant methods (step 3).*

The objective of this module is to identify the relevant methods (i.e., $\mathcal{M}$) which might contain the vulnerabilities studied in this paper. Algorithm 1 presents the logic of this module. The algorithm starts by collecting all the relevant classes in one list, $\mathcal{C}$ (line 1) and all the methods for each of these classes (line $2-5$). If the number of found methods exceeds a certain threshold th_m (line 6), the algorithm reconstructs the list $\mathcal{M}$ to reduce its length (line $9-27$). This reduction is done by considering three sets of methods (i.e., $\mathcal{M}_I, \mathcal{M}_W, \mathcal{M}_a$) to build $\mathcal{M}$. The first set, $\mathcal{M}_I$ (constructed in the steps $9-17$), includes annotated methods ($\mathcal{M}_A$, step 9) and *init* and *onCreate* methods of classes in $\mathcal{I}_c$ (line 10–13). If the *minSdk* of the application is less than 17, $\mathcal{M}_I$ is augmented with methods of classes in $\mathcal{I}_{c17}$ (line 14–17). The second set, $\mathcal{M}_W$, is the set of WebViewClient methods (line 18–20). The third set, $\mathcal{M}_a$, is the set of *access* methods of classes $\mathcal{C}$ (line 22–25). Finally, the algorithm calculates the number of arguments for methods in $\mathcal{M}$ (inline 27).

6.4. *Methods abstraction (step 4).*

Our experiments showed that analyzing abstracted versions of method paths is more efficient and convenient than analyzing the paths in their native Dalvik code. Therefore, this module presents a new technique for abstracting Dalvik instructions related to our studied vulnerabilities. This module starts by building execution paths for methods in $\mathcal{M}$.

In Table 3, we collected the set of Dalvik instructions that are most related to our studied vulnerabilities. Each instruction is a pair $(\mathcal{N}, \mathcal{O})$ where $\mathcal{N}$ is the instruction name and $\mathcal{O}$ a list of operands. The instruction name belongs to one of the following eight categories: removing_source_ins, read, write, invoke, move, return, and constant instructions. Each element of the operands list belongs to one of the following four categories: register, constant, field, method. Considering, for example, the *method category*, this encompasses two elements, composed of two pieces. The first piece is a list of pairs $[0, integer]^*$, where the 0 indicates that the integer is a register ID. The other piece is a triple [*integer*, *integer*, *method signature*] that uniquely identifies a method. The first piece represents the arguments of the second piece.

Table 4 shows the syntax of the abstracted Dalvik instructions. We have eight categories of abstracted instructions as follows:

1. *register-register* - it abstracts instructions that move data between registers.
2. *register-field* - it abstracts instructions that move data between registers and class fields.
3. *method call* - it abstracts the Dalvik invocation instructions. The abstraction is done in two steps. The first step produces the first instruction form in this category. The second step augments the instruction built in the first step with the component [*RESULT*, *integer*] that identifies the register holding the result of the invocation (if any).
4. *sink call* - it abstracts the instructions invoking sink methods, where $integer^*$ is the set of register IDs including leaked registers.
5. *source call* - it abstracts instructions that read sensitive data and store it into the register whose ID is the *integer*. This particular abstraction is practically done in two steps. The result of the first step is the abstracted instruction [*SOURCE*] and the second step augments this result with the register ID to produce [*SOURCE*, *integer*].
6. *move call* - it abstracts the Dalvik instructions that move results of method invocations to other registers.
7. *return* - it abstracts the Dalvik return instructions.
8. *constant* - it abstracts the instructions that assign constants of different types into registers.

Table 5 presents the inference rules used for abstracting Dalvik instructions from the syntax illustrated in Table 3 towards the syntax shown in Table 4. The rules assume a path p whose Dalvik instructions are to be abstracted and appended to the list $\mathcal{L}(p)$. The following symbols are used in the rules:

- $s_1 \overset{\text{def}}{\Longleftrightarrow} \mathcal{O}[len(\mathcal{O}) - 1]$, and
- $s_2 \overset{\text{def}}{\Longleftrightarrow} s_1[len(s_1) - 1]$.

Below, we illustrate each abstraction rule shown in Table 5 with some reference examples of rule application. All the examples are taken from a testbed Android app and are presented as follows:

TheDalvikinstruction
Thecorrespondingabstraction.

Rule 1. This abstracts some of the category of *read* instructions. In particular, Rule 1 treats the cases of assigning the con-

Algorithm 1 Find_All_Relevant_Methods().

Input: The Sets of classes $\mathcal{I}_c, \mathcal{I}_{c17}, \mathcal{W}, \mathcal{I}_u$, and $\mathcal{W}_u$.
Output: The list $\mathcal{M}$ of all methods relevant to the studied vulnerabilities.
Steps:

$\mathcal{C} \leftarrow \mathcal{I}_c + \mathcal{I}_{c17} + \mathcal{W} + \mathcal{I}_u + \mathcal{W}_u$.
$\mathcal{M} \leftarrow []$.
for each $c \in \mathcal{C}$ **do**
  for each $m \in c.methods$ **do**
    $\mathcal{M}.append(m)$.
if $len(\mathcal{M}) \leq th_m$ **then**
  return $\mathcal{M}$.
else
  $\mathcal{M}_I \leftarrow \mathcal{M}_{\mathcal{A}}$.
  for each $c \in \mathcal{I}_c$ **do**
    for each $m \in c.methods$ **do**
      if $m.name \in \{'init', 'onCreate'\}$ **then**
        $\mathcal{M}_I.append(m)$.
  if minSdk ≤ 17 **then**
    for each $c \in \mathcal{I}_{c17}$ **do**
      for each $m \in c.methods$ **do**
        $\mathcal{M}_I.append(m)$.
  for each $c \in \mathcal{W}$ **do**
    for each $m \in c.methods$ **do**
      $\mathcal{M}_W.append(m)$.
  $\mathcal{M} \leftarrow \mathcal{M}_I + \mathcal{M}_W$
  for each $c \in \mathcal{C}$ **do**
    for each $m \in c.methods$ **do**
      if $'access' \in m.name$ **then**
        $\mathcal{M}_a.append(m)$.
  $\mathcal{M} \leftarrow \mathcal{M} + \mathcal{M}_a$
$\mathcal{M}_r \leftarrow Find\_Method\_Register\_Numbers(\mathcal{M})$.
return $\mathcal{M}$.

Table 3 – Syntax of Dalvik Instructions.

$\mathcal{N}$	::=		Instruction name
		neg-int | *not-int* | *long-to-int* | *new-instance* | *array-length* | *add-int* | ...	Removing_Source_ins
		| *iget*∗ | *sget*∗ | *aget*∗ | ...	read instructions
		| *iput*∗ | *sput*∗ | *aput*∗ | ...	write instructions
		| *invoke*∗ | ...	invoke instructions
		| *move-*∗ | ...	move instructions
		| *return-*∗ | ...	return instructions
		| *const-*∗ | ...	constant instructions
$\mathcal{O}p$	::=		Operand item
		[0, *integer*]	register operand
		| [1,*real number*]	constant operand
		| [*integer*, *integer*, *field signature*]	field operand
		| [0, *integer*]*[*integer*, *integer*, *method signature*]	method operand
$\mathcal{O}$	::=	$\mathcal{O}p^*$	
$\mathcal{I}nst$	::=	$(\mathcal{N}, \mathcal{O})$	Dalvik instructions

tent of a store (register or a field) into a register. Therefore, the resulting abstracted instruction belongs to the *register-register* or *register-field* categories. The first component of the resulting instruction is the destination register and the second component is the source store. Examples of the abstraction applied by Rule 1 are as follows:

$(iget - object, [(0, 0), (0, 0), (258, 3987, MainActivity; -> editor)])$
$[0, MainActivity; -> editor]$.

Table 4 – Syntax of Abstracted Dalvik Instructions.

$\mathcal{I}nstAbs$ ::=		instruction abstraction
	[*integer, integer*]	register-register
	| [*string*, *integer*] | [*integer*, *string*]	register-field
	| [*CALL*, id_m, *integer**, $\mathcal{O}$]	method call
	| [*CALL*, id_m, *integer**, $\mathcal{O}$, [*RESULT*, *integer*]]	
	| [*SINK*, *integer**]	sink call
	| [*SOURCE*] | [*SOURCE*, *integer*]	source call
	| [*RESULT*, *integer*]	move call
	| [*RETURN, VOID*] | [*RETURN*, *integer*]	return
	| [*integer*, *CONSTANT*]	constant

Table 5 – Abstraction Rules.

$$\frac{iget \in \mathcal{N} \vee sget \in \mathcal{N} \quad len(\mathcal{O})=3}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([\mathcal{O}[0][1], s_2])} \quad (1)$$

$$\frac{aget \in \mathcal{N} \quad len(\mathcal{O})=3}{(\mathcal{N},\ \mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([\mathcal{O}[0][1],\ \mathcal{O}[1][1]])} \quad (2)$$

$$\frac{iput \in \mathcal{N} \vee sput \mathcal{N} \quad len(\mathcal{O})=3}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([s_2, \mathcal{O}[0][1]])} \quad (3)$$

$$\frac{aput \in \mathcal{N} \quad len(\mathcal{O})=3}{(\mathcal{N},\ \mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([\mathcal{O}[1][1],\ \mathcal{O}[0][1]])} \quad (4)$$

$$\frac{\mathcal{N} \in invokeinstructions \quad \exists m \in \mathcal{M}: s_2 \in m}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([CALL, id_m, Reg(\mathcal{O}), \mathcal{O}])} \quad (5)$$

$$\frac{\mathcal{N} \in invokeinstructions \quad \forall m \in \mathcal{M}: s_2 \notin m \quad s_2 \in Sinks}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([Sink, Reg(\mathcal{O})])} \quad (6)$$

$$\frac{\mathcal{N} \in invokeinstructions \quad \forall m \in \mathcal{M}: s_2 \notin m \quad s_2 \in Sources}{(\mathcal{N},\ \mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([Source])} \quad (7)$$

$$\frac{\mathcal{N} \in moveinstructions len(\mathcal{O}) = 1 \mathcal{L}(p)[-1] = [SOURCE]}{(\mathcal{N},\ \mathcal{O}) \rightsquigarrow \mathcal{L}(p)[-1].append(\mathcal{O}[0][1])} \quad (8)$$

$$\frac{\mathcal{N} \in moveinstructions len(\mathcal{O}) = 1 \mathcal{L}(p)[-1][0] = CALL}{(\mathcal{N},\ \mathcal{O}) \rightsquigarrow \mathcal{L}(p)[-1].append([RESULT,\ \mathcal{O}[0][1]])} \quad (9)$$

$$\frac{\mathcal{N} \in moveinstructions \quad len(\mathcal{O})>1}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([\mathcal{O}[0][1],\ \mathcal{O}[1][1]])} \quad (10)$$

$$\frac{\mathcal{N} \in returninstructions \quad len(\mathcal{O})=0}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([RETURN, VOID])} \quad (11)$$

$$\frac{\mathcal{N} \in returninstructions \quad len(\mathcal{O})>0}{(\mathcal{N},\mathcal{O}) \rightsquigarrow \mathcal{L}(p).append([RETURN,\ \mathcal{O}[0][1]])} \quad (12)$$

$(iget - object, [(0, 0), (0, 0), (258, 3990, MainActivity; -> telephonyManager)])[0, MainActivity; -> telephonyManager]$.

Rule 2. This rule abstracts also some of the category of *read* instructions. More specifically, Rule 2 treats the cases of assigning the value of an array location into a register. Therefore, the resulting abstracted instruction belongs to the *register-register* category, where the second component is the register ID holding the array reference.

Rule 3. This rule abstracts subset of the instructions in the *write* category. Rule 3 is similar to Rule 1. However, the order of components of the resulting abstracted instruction is reversed. An example of the abstraction applied by Rule 3 is as follows:

$(iput, [(0, 0), (0, 3), (258, 3986, webviewInterface; -> xInter)])$
$[webviewInterface; -> xInter, 0]$.

Rule 4. This rule abstracts subset of the instructions in the *write* category. More specifically, the rule treats instructions writing into arrays. Rule 4 is similar to Rule 2. However, the order of components of the resulting abstracted instruction is reversed.

Rule 5. This abstracts the category of *invoke* instructions. This rule is effective when the invocation is for a method in $\mathcal{M}$. In this case, the resulting abstracted instruction is a list of the string 'CALL', the method id, argument registers, and operand list of the Dalvik instruction. An example of the abstraction applied by this rule is as follows:

$(invoke - direct, [(0, 3), (0, 0), (0, 1), (256, 249, f2(String, String))])$
$[CALL, f2, [3, 0, 1], [(0, 3), (0, 0), (0, 1), (256, 249, f2(String, String))]]$.

Rule 6. This abstracts the category of *invoke* instructions. This rule is effective when the invocation is for a source method. In this case, the resulting abstracted instruction is a pair of the string 'SINK' and a list of the register arguments including the leaked register. An example of the abstraction applied by this rule is as follows:

$(invoke - interface, [(0, 0), (0, 2), (0, 1), (256, 2, Editor; -> putInt())])$
$[SINK, [0, 2, 1]]$.

Rule 7. This abstracts the category of *invoke* instructions. This rule is effective when the invocation is for a source method. In this case, the resulting abstracted instruction is just the string 'SOURCE' as a mark that a source invocation has been encountered. This abstracted instruction is augmented later with a register ID when encountering the relevant *move*

instruction. An example of the abstraction applied by this rule is as follows:

$(invoke - virtual, [(0, 0), (256, 5, TelephonyManager; -> getDeviceId())])$
$[SOURCE]$.

Rule 8. This abstracts the category of *move* instructions. This rule is effective when the last abstracted instruction is an invocation for a source method and the length of the operands list is 1. In this case, the rule augments the last element of $\mathcal{L}(p)$ with the ID of the register holding the sensitive data. An example of the abstraction applied by this rule is as follows:

$(move - result - object, [(0, 0)])$
$[SOURCE, 0]$.

Rule 9. This abstracts the category of *move* instructions. This rule is effective when the last abstracted instruction is an invocation for a method in $\mathcal{M}$ and the length of the operands list is 1. In this case, the rule augments the last element of $\mathcal{L}(p)$ with the ID of the register holding the result of the invocation.

Rule 10. This abstracts the category of *move* instructions. This rule is effective when the length of the operands list is greater than 1. In this case, an abstracted instruction of the category *register-register* is added to $\mathcal{L}(p)$ to express this data move between registers.

Rule 11. This abstracts the category of *return* instructions; in particular the ones whose list of operands is []. An example of the abstraction applied by this rule is as follows:

$(return - void, [])$
$[RETURN, VOID]$.

Rule 12. This rule abstracts elements of the category of *return* instructions whose list of operands is nonempty.

Rule 13. We identified a category of Dalvik instructions that have the effect of overwriting sensitive data of a register and replace it with non-sensitive data. We named this category as *Removing_Source_ins* and identified 112 Dalvik instructions (for example *add-int* and *new-instance*) belonging to it. These instructions, from the point of view of our analysis, have same effect as the *constant* category of our syntax. Therefore, Rule 13 treats equally the categories *Removing_Source_ins* and *constant*. The abstraction result is a pair of the ID of the updated register and the string *CONSTANT* as a mark that there is no sensitive data in this register. Examples of the abstraction applied by this rule are as follows:

$(const/4, [(0, 0), (1, 7)])$
$[0, CONSTANT]$.

$(const - string, [(0, 1), (257, 2034, android.permission.READ\_PHONE\_STATE)])$
$[1, CONSTANT]$.

Table 6 – Path Analysis Notation.

Notation	Description
$\mathcal{R}(p)$	The tuple $< \mathcal{R}_s, \mathcal{R}_l, \mathcal{R}_r, \mathcal{R}_m, \mathcal{R}_c, \mathcal{R}_k >$ contains analysis results of the execution path p.
$\mathcal{R}_s$	The set of registers and class fields that contain sensitive data read by a source API in the path p.
$\mathcal{R}_l$	A set of paths leaking data; containing a sink API that leaks data read by a source API.
$\mathcal{R}_r$	The register ID containing the data returned by p (if p returns data).
$\mathcal{R}_m$	A set of pairs of classes fields (tracing sensitive data move).
$\mathcal{R}_c$	A set of pairs (i, s) such that i is a register and s is a register or a filed whose value is assigned to i.
$\mathcal{R}_k$	A set of registers and fields whose contents are leaked by a sink API in p, but are not updated in p before being leaked.

6.5. *Path analysis (step 5).*

This module is a recursive one that analyzes abstracted paths (created by the previous module) of methods in $\mathcal{M}$. For an execution path p, this module analyzes the instructions of $\mathcal{L}(p)$ one by one and the analysis inputs are $(id_p, \mathcal{L}(p), \mathcal{R}(p), \mathcal{L}id(p))$, where $\mathcal{R}(p) =< \mathcal{R}_s, \mathcal{R}_l, \mathcal{R}_r, \mathcal{R}_m, \mathcal{R}_c, \mathcal{R}_k >$. Table 6 lists the semantics of these notations. Some comments on the module inputs are in order:

- Results of the analysis are built gradually into $\mathcal{R}(p)$. Therefore at a program point of p, $\mathcal{R}(p)$ would contain the results collected till that program point. The recursive nature of the module causes including this parameter in the module arguments.
- The set $\mathcal{L}id(p)$ (containing IDs of method called in p till the instruction under analysis) is included in the inputs to avoid infinite loops. Such loops are possible due the static nature of our analysis.
- $\mathcal{R}_m$ aims at tracing the move of sensitive data among class fields.
- $\mathcal{R}_c$ aims at watching contents of registers relevant to the analysis (not all registers).

The analysis is based on a heuristically developed set of inference rules that are presented in Table 7.

The rules mainly show the effect that instructions have on the entries of the tuple $\mathcal{R}(p)$. A primed entry of $\mathcal{R}(p)$, represents the new value of the variable after analyzing an instruction.

Rule 14. The rule shows that the *return* instruction has effect on $\mathcal{R}_c$ and $\mathcal{R}_s$ only. The remaining entries of $\mathcal{R}(p)$ are unchanged by this instruction. For $\mathcal{R}_c$, it gets restricted on the register i. For $\mathcal{R}_s$, it gets first restricted on fields, then it may get augmented with the register i or with $\mathcal{R}_r$ according to the conditions in the rule.

Rule 15. The rule treats the *CONSTANT* instructions, by removing the register i from $\mathcal{R}_c$ and $\mathcal{R}_s$. This is so as assigning a constant to the register overwrites any sensitive data in it and hence tracing its content becomes not important.

Table 7 – Analysis Rules.

$$\frac{\mathcal{R}_s^1=\mathcal{R}_s\setminus Registers \qquad \mathcal{R}_s^2=\begin{cases}\mathcal{R}_s^1, & i\notin\mathcal{R}_s\\ \mathcal{R}_s^1\cup\mathcal{R}_r, & \mathcal{R}_r!=[]\ \&\ i\in\mathcal{R}_s;\\ \mathcal{R}_s^1\cup\{i\}, & \mathcal{R}_r=[]\ \&\ i\in\mathcal{R}_s;\end{cases}}{[RETURN,i]:\mathcal{R}_c'\mapsto\mathcal{R}_c\setminus\{i\}\wedge\mathcal{R}_s'\mapsto\mathcal{R}_s^2} \quad (14)$$

$$[i, CONSTANT]:\mathcal{R}_s'\mapsto\mathcal{R}_s.remove(i)\wedge\mathcal{R}_c'\mapsto\mathcal{R}_c.remove((i,\_)) \quad (15)$$

$$[SOURCE, i]:\mathcal{R}_s'\mapsto\mathcal{R}_s.append(i)\wedge\mathcal{R}_c'\mapsto\mathcal{R}_c.remove((i,\_)) \quad (16)$$

$$\frac{\forall i\in L: i_o \text{ is the field whose value is assigned to } i,\ \text{if exists calculated using } \mathcal{R}_c \qquad \forall i\in L: i_{of} \text{ is a field whose value is assigned to } i_o,\ \text{if exists calculated using} \mathcal{R}_m}{[SINK,\ L]:\begin{matrix}\mathcal{R}_l'=\mathcal{R}_l\cup\{id_p\mid L\cap\mathcal{R}_s\neq\emptyset\}\wedge\\ \mathcal{R}_k'=\mathcal{R}_k\cup\{i_o,i_{of}\mid i\in L\wedge i_o\notin\mathcal{R}_s\wedge i_{of}\notin\mathcal{R}_s\}\\ i_1 \text{ is register ID}\end{matrix}} \quad (17)$$

$$[i_1,i_2]:\mathcal{R}_c'=\mathcal{R}_c[i_1\mapsto i_2]\wedge\mathcal{R}_s'=\begin{cases}\mathcal{R}_s.append(i_1), & i_2\in\mathcal{R}_s;\\ \mathcal{R}_s.remove(i_1), & \text{otherwise.}\end{cases} \quad (18)$$

$$\frac{i_1\in \text{is a class field} \qquad i_2^o \text{is the field whose value is assigned to } i_2,\ \text{if exists calculated using} \mathcal{R}_c}{[i_1,i_2]:\begin{matrix}\mathcal{R}_s'=\begin{cases}\mathcal{R}_s.append(i_1), & i_2\in\mathcal{R}_s;\\ \mathcal{R}_s.remove(i_1), & \text{otherwise.}\end{cases}\\ \mathcal{R}_m'=\begin{cases}\mathcal{R}_m.remove((i_1,\_)), & i_2^o\in \text{is a register};\\ \mathcal{R}_m.append((i_2^o,i_1)), & i_2^o!=i_1.\end{cases}\end{matrix}} \quad (19)$$

$$\frac{\begin{matrix}p_m \text{ is the path of } id_m \qquad \mathcal{R}_s^m=\mathcal{R}_s.remove(registers)\cup\{arg(j)\mid j\in Rgs\cap\mathcal{R}_s\}\\ \mathcal{R}_c^m=\{(arg(i_1),i_2)\mid(i_1,i_2)\in\mathcal{R}_c\wedge i_1\in Rgs\}\\ sRegs=\mathcal{R}_s.remove(fields) \qquad \mathcal{R}^m=\langle\mathcal{R}_s^m,\mathcal{R}_l,j,\mathcal{R}_m,\mathcal{R}_c^m,\mathcal{R}_k\rangle\\ \mathcal{R}^n(p_m)=Path\ Analysis(id_{p_m},\mathcal{L}(p_m),\mathcal{R}^m,\mathcal{L}id(p)+[id_m])\end{matrix}}{[CALL,id_m,Rgs,\mathcal{O},[RESULT,j]]:\begin{matrix}\mathcal{R}_s'=\mathcal{R}_s^n\cup sRegs\wedge\mathcal{R}_l'=\mathcal{R}_l^n\wedge\mathcal{R}_r'=\mathcal{R}_r^n\\ \mathcal{R}_m'=\mathcal{R}_m^n\wedge\mathcal{R}_c'=\mathcal{R}_c^n\wedge\mathcal{R}_k'=\mathcal{R}_k^n\end{matrix}} \quad (20)$$

Rule 16. The rule treats the *SOURCE* instructions by appending the sourced register to $\mathcal{R}_s$. The rule also removes the register from $\mathcal{R}_c$. The reason for this removal is that the register status became evident and hence it is not necessary to trace its content anymore. Derivation 21 provides an example of applying this rule.

$$\frac{\mathcal{R}(p)=<\{\},\{\},0,\{\},\{(0,MainActivity;->telephonyManager)\},\{(MainActivity;->editor,webviewInter face;->xInter)\}>}{[SOURCE,0]:\mathcal{R}_s'\mapsto\{0\}\wedge\mathcal{R}_c'\mapsto\{\}} \quad (21)$$

Rule 17. The rule treats the *SINK* instruction. In case $L\cap\mathcal{R}_s\neq\emptyset$ which means that one of the sink registers has sensitive data, then the path leaks data. Therefore the path ID is added to $\mathcal{R}_l$. For each $i\in L$, the rule adds to $\mathcal{R}_k$ all fields along the assignment path that leads to the value in i. This augmentation of $\mathcal{R}_k$ is done because, as leaking caused by i, other leaks may be caused by the fields added to $\mathcal{R}_k$.

Rule 18. The rule treats the *register-register* and *register-field* instructions whose first entry is a register. The rule updates $\mathcal{R}_c$ to record the data flow from i_2 to i_1. The rule also updates $\mathcal{R}_s$ by adding or removing i_1 depending on whether i_2 contains sensitive data. An example of applying Rule 18 is given in derivation 22.

$$\frac{\mathcal{R}=[\{\},\{\},0,\{\},\{\},\{\}]}{[0,MainActivity;->editor]:\mathcal{R}_c'\mapsto\{(0,'MainActivity;->editor')\}} \quad (22)$$

Rule 19. The rule treats the *register-field* instructions whose first entry is a field.

Rule 20. The rule treats the *method call* category of instructions. In this rule, the recursive side of the module is evident as the module calls itself. The rule builds arguments ($\mathcal{R}^m$) for the recursive call from arguments of current call ($\mathcal{R}$). For example, $\mathcal{R}_s^m$ is composed of fields of $\mathcal{R}_s$ plus the set of argument registers (denoted by *Rgs* in the rule) that contain sensitive data (i.e. in $\mathcal{R}_s$). However, the IDs of these argument registers are translated to their corresponding local registers in the method *m*. This is denoted by the operation *arg()* in definition of $\mathcal{R}_s^m$. The final result of the rule is derived from the result of the recursive call (denoted by $\mathcal{R}^n(p_m)$ in the rule): for instance, $\mathcal{R}_m'=\mathcal{R}_m^n$. Derivation 23 provides an example of applying Rule 20.

$$\frac{\mathcal{R}=[\{0\},\{\},\_,\{\},\{(0,telephonyManager)\},\{\}]}{\begin{matrix}[CALL,f2,[3,0,1],[(0,3),(0,0),(0,1),(256,249,webInter face;->f2(Str))]]:\\ \mathcal{R}_s'=\{0,id\}\wedge\mathcal{R}_l'=\{id_{f2}\}\wedge\mathcal{R}_r'=\{\}\wedge\mathcal{R}_m'=\{(telephonyManager,id)\}\wedge\\ \mathcal{R}_c'=\{(0,telephonyManager)\}\wedge\mathcal{R}_k'=\{editor,telephonyManager\}\end{matrix}} \quad (23)$$

6.6. *Finding vulnerabilities (step 6).*

This module utilizes the analysis results of the previous module to detect the WebView Vulnerabilities. This is done by implementing the four vulnerability definitions (illustrated in Section 5) on the analysis results. For instance, assuming that $\mathcal{R}=\{\mathcal{R}(p)=<\mathcal{R}_s^p,\mathcal{R}_l^p,\mathcal{R}_r^p,\mathcal{R}_m^p,\mathcal{R}_c^p,\mathcal{R}_k^p>\mid p\in\mathcal{P}\}$ is the set of analysis results for a set of paths $\mathcal{P}$, type 1 vulnerability occurs if one of the following conditions are verified:

- For some $p\in\mathcal{P}$, $\mathcal{R}_p^p\neq\emptyset$ and the method of this path is an interface one.
- There exist paths $p_1,\ldots,p_n\subseteq\mathcal{P}$, $i\in\mathcal{R}_s^1$, $j\in\mathcal{R}_k^n$, and fields $f_1,\ldots,f_n$ such that the methods of p_1 and p_n are interface ones and $(i,d_1)\in\mathcal{R}_m^1,\ldots,(d_n,j)\in\mathcal{R}_m^n$.

7. Implementation

This section illustrates the implementation details of $\mathcal{W}$eb$\mathcal{VS}$ec which design has been introduced in Section 6. $\mathcal{W}$eb$\mathcal{VS}$ec is implemented on top of Androguard Desnos (2011), a tool for reverse engineering Android applications.

7.1. *Decompilation (step 1).*

Androguard (Release 3.3.5) Desnos (2011) is used to implement the decompilation phase. Androguard results in three objects. Information about the APK (e.g., manifest content, package name, permissions) is included in the first object (i.e., $a_{\mathcal{A}}$). DEX files of the APK file are included in the second object (i.e., $d_{\mathcal{A}}$). The third object includes information about dex classes (i.e., $dx_{\mathcal{A}}$).

7.2. *Identification of interface and webviewclient classes (step 2).*

The implementation of this module relies on the Androgurad methods `get_item_type()` of $d_{\mathcal{A}}$ and `get_method_annotations()` that is included in one of the attributes of $d_{\mathcal{A}}$. These methods enable constructing the sets $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{I}_c$. To analyze applications that target a minSdk less than 17, where annotating interface methods is not necessary, we do as follows: this module searches for all the methods in classes of $dx_{\mathcal{A}}$ to find instructions that invoke the APIs `addJavascriptInterface` and `setWebViewClient`; the parameters of these instructions enables collecting $\mathcal{I}_{c17}, \mathcal{C}_w, \mathcal{W}$, and $\mathcal{W}_u$.

7.3. *Identification of relevant methods (step 3).*

The implementation of Algorithm 1 benefits from the objects produced by our reverse engineering tool. The implementation of steps 4, 11, 16, 19, and 23 uses the Androguard API `c.get_methods()` to extract methods of a class `c`. This API is included in the object $dx_{\mathcal{A}}$. The API `get_min_sdk_version()` (contained in the object $a_{\mathcal{A}}$) is used to implement step 14 of the algorithm, where the minSdk needs to be checked. Each method of the app has an Androguard object that includes an attribute called `name`. For a method `m`, this attribute is read via the code `m.name` to implement the steps 12 and 24.

7.4. *Methods abstraction and path analysis (step 4 and step 5).*

The implementation of these modules uses several Androguard APIs including:

- `basic_blocks.gets()` that returns blocks of a method.
- `get_next()` that returns child blocks of a given one.
- `get_nb_instructions()` that return the number of instructions in a block. This is useful for finding the longest path of a method.

For each method, we first build its execution path in the form of lists of instruction blocks using the APIs `basic_blocks.gets()` and `get_next()`. Hence each method path is a list of blocks. For each path, we then iterate its blocks to extract the block instructions using the API `get_nb_instructions()`. The instruction name and operands are extracted from each instruction to determine a convenient abstraction rule (among the ones in Table 5) to be applied. This relies on the APIs `get_name()` and `get_operands()`. Hence each Dalvik path has a corresponding path of abstracted instructions which has the form of a list. During instruction iteration, some instructions may not match any rule. In this case, this instruction is not relevant to our analysis and hence gets dropped. For each abstracted path (list), we iterate its elements to analyze its content using convenient rules of Table 7. Applying an analysis rule mainly consists in updating the variables of Table 6, according to the applied rule.

The result of $\mathcal{W}$eb$\mathcal{VS}$ec is implemented as a text file that contains all the information collected in all phases of the system.

7.5. *Custom thresholds.*

Dense calculations in $\mathcal{W}$eb$\mathcal{VS}$ec might lead to large consumption of time and computation resources. Therefore, we heuristically fixed a set of thresholds to limit this consumption with very little effect on the precision of the $\mathcal{W}$eb$\mathcal{VS}$ec results. These thresholds are as follows:

- An upper bound (denoted by th_m) on the number of methods to be analyzed. We heuristically fixed th_m to 600. This number is reasonable because it covers almost all methods of interface and webViewClient classes in the apps of our dataset. These are the methods that would include the studied vulnerabilities in most cases. However, increasing this threshold results in studying methods of classes using the webView objects. This makes the analysis more inclusive than using a lower threshold and hence increases the probability of discovering vulnerabilities related to the added methods. However, heuristically, it is rare to find vulnerabilities related to these added methods.
- The number of paths of WebView interface and WebViewClient methods that are to be analyzed. We heuristically found that it is mostly enough to analyze the longest path. The experiments showed that this longest path in most cases includes key instructions relevant to $\mathcal{W}$eb$\mathcal{VS}$ec. This is so because in most cases the short paths inside the methods of interface and webViewClient represent *else* branches opting-out using webView objects. Therefore analyzing these short paths mostly does not affect the precision of $\mathcal{W}$eb$\mathcal{VS}$ec.

8. Evaluation

This section presents the results of our experiments conducted for evaluating $\mathcal{W}$eb$\mathcal{VS}$ec. We got the dataset from a reputable benchmark, namely AndroZoo Allix et al. (2016). All experiments were done on a Dell (Vostro) device with processor: Intel(R) Core(TM) i7-3612 QM CPU @ 2.10 GHz, 8.00 GB RAM, and Windows 10 (64-bits) operating system. All implementations were written in Python on top of Androguard Desnos (2011),

```
az -n 2000 -d 2019-01-01: -m  play.google.com -md sha256,
    pkg_name,apk_size,dex_date,markets,vt_detection,md5 -o
    output
```

Listing 6 – AndroZoo Command.

Table 8 – $\mathcal{W}$eb$\mathcal{VS}$ec Versus *BabelView*.

#	criteria	$\mathcal{W}$eb$\mathcal{VS}$ec	*BabelView*
1	Number of analyzed apps	1994	1994
2	Total analysis time (Sec)	97779.7	229137.7
3	Average analysis time per app (Sec)	49.0	114.9
4	Number of analyzed apps	1994	881
5	Number of apps whose minSdk <= 17	1872	⊗
6	Number of apps whose minSdk > 17	122	⊗

The symbol ⊗ means that the technique does not provide the statistics.

the tool we adopted for reverse engineering Android applications. We make all our results and data files available[1].

We downloaded a random set of 2000 applications, whose sha256 is available online[2], in early 2020. The instruction we used to download the dataset is given in Listing 6. We limited the download to applications that Androzoo obtained from the *Google Play Store*[3]. We also restricted our download to recent apps by specifying the publishing dates of dataset from 2019 onwards. The list after the option "-md" in the instruction is a metadata that AndroZoo provides for each downloaded APK.

Tables 8 and 9 contain the statistics we collected through our experiments where we compared $\mathcal{W}$eb$\mathcal{VS}$ec against *BabelView*, answering the following research questions:

RQ1. **Applicability, effectiveness, and performance**: How does $\mathcal{W}$eb$\mathcal{VS}$ec compare against alternative approaches inapplicability, discovering WebView vulnerabilities and in performance?

RQ2. **Half vulnerabilities**: How does $\mathcal{W}$eb$\mathcal{VS}$ec compare with alternative approaches in discovering paths that read a value and paths that send out a value?

RQ3. **Accuracy**: What is the accuracy of $\mathcal{W}$eb$\mathcal{VS}$ec and the alternative approaches?

We also conclude the current state of WebView use by programmers in Android applications with the following question:

RQ4. **WebView usage in Android ecosystem**: What is the current usage of all elements that contribute to the vulnerabilities of our threat model (e.g., WebView, interfaces, WebViewClient)?

8.1. *Applicability, effectiveness, and performance*

For a complete evaluation, we compared the applicability, effectiveness, and performance of $\mathcal{W}$eb$\mathcal{VS}$ec with one of the state-of-the-art techniques, *BabelView*Rizzo et al. (2018), which we selected for the following criteria. First, *BabelView* is a recent technique that addresses recent versions of Android. Second, the implementation of *BabelView* is available as a public tool suite. Third, *BabelView* is one of the most closely related techniques to $\mathcal{W}$eb$\mathcal{VS}$ec. *BabelView* was implemented using the Soot framework Vallée-Rai et al. (2010). To eliminate bias in favor of $\mathcal{W}$eb$\mathcal{VS}$ec, we used the list of sources and sinks provided by *BabelView*[4] in our experiments. We also ran $\mathcal{W}$eb$\mathcal{VS}$ec and *BabelView* on the same dataset.

Applicability. Running $\mathcal{W}$eb$\mathcal{VS}$ec on the 2000 applications of the data set resulted in only 6 cases that Androguard failed to decompile. The remaining 1994 applications were successfully analyzed by $\mathcal{W}$eb$\mathcal{VS}$ec. Therefore, we ran $\mathcal{W}$eb$\mathcal{VS}$ec and *BabelView* on these 1994 applications. Out of the 1994 applications, *BabelView* analyzed and produced reports for only 881 applications. However, $\mathcal{W}$eb$\mathcal{VS}$ec analyzed and produced reports for all 1994 applications. $\mathcal{W}$eb$\mathcal{VS}$ec revealed that almost 94% (1872 applications) of the analyzed sample specifies a minimum SDK that is less than or equal to 17. This confirms the applicability of $\mathcal{W}$eb$\mathcal{VS}$ec to applications that target old versions of Android, where the annotation of interface methods is not required. The annotation details were presented in Section 2.

Effectiveness. To evaluate whether $\mathcal{W}$eb$\mathcal{VS}$ec can discover vulnerability types presented in Section 4, we developed a toy Android app that includes all types of vulnerabilities. The app includes 30 vulnerabilities that are distributed among the 4 types. While our technique, $\mathcal{W}$eb$\mathcal{VS}$ec, discovered all the 30 vulnerabilities, *BabelView* did not manage to discover any of these vulnerabilities. The running example of the paper was taken from this application. Files of the toy application and its analysis results are available online[5].

[1] https://github.com/maelzawawy/WebVSec.git
[2] https://github.com/maelzawawy/WebVSec.git
[3] https://play.google.com/store/
[4] https://github.com/ClaudioRizzo/BabelView
[5] https://github.com/maelzawawy/WebVSec.git

Table 9 – $\mathcal{W}$ebV$\mathcal{S}$ec Versus *BabelView*.

#	criteria	$\mathcal{W}$ebV$\mathcal{S}$ec		*BabelView*	
		instances#	apps#	instances#	apps#
Vulnerabilities					
1	Total number of vulnerabilities	82	48	26	18
2	Type 1 vulnerabilities	44	20	18	11
3	Type 3 vulnerabilities	38	36	0	0
Half Vulnerabilities					
4	Sourced Vars	626	443	72	34
5	Sourced Vars in WebViewClient classes	472	409	0	0
6	Sinked Vars	3403	744	564	184
7	Sinked Vars in WebViewClient classes	2016	668	0	0
Statistics					
8	Interface classes of annotated methods ($\mathcal{I}_c$)	3110	991	⊗	⊗
9	Interface classes in apps targeting SDK < 17 ($\mathcal{I}_{c17}$)	3609	1320	⊗	⊗
10	Interface methods	53,367	1405	⊗	⊗
11	Classes use interfaces ($\mathcal{I}_u$)	4444	1404	⊗	⊗
12	WebViewClient classes ($\mathcal{W}$)	9615	1749	⊗	⊗
13	Classes use WebViewClient ($\mathcal{W}_u$)	15,050	1774	⊗	⊗
14	WebClient Methods	43,772	1748	⊗	⊗
15	Relevant methods	500152	1797	⊗	⊗
16	Relevant methods after reduction ($\mathcal{M}$)	366,708	1797	⊗	⊗
17	Access methods ($\mathcal{M}_a$)	13,525	148	⊗	⊗
18	Methods use sources	2567	827	⊗	⊗
19	All Sources used	5079	827	⊗	⊗
20	Methods use sources in WebViewClient	465	410	⊗	⊗
21	All sources in webViewClient	1502	410	⊗	⊗
22	Methods use sinks	18,763	1770	⊗	⊗
23	All sinks used	35,839	1770	⊗	⊗
24	Methods use sinks in WebViewClient	4608	1276	⊗	⊗
25	All sinks in WebViewClient	8794	1276	⊗	⊗

The symbol ⊗ means that the technique does not provide the statistics.

The total number of vulnerabilities discovered by $\mathcal{W}$ebV$\mathcal{S}$ec is 82 in 48 applications. This number is a breakdown into 44 of type 1 vulnerability and 38 of type 3 vulnerability. The number of interface classes which methods contribute to type 1 vulnerability is 21 in 20 applications. The number of WebViewClient classes which methods contribute to type 3 vulnerability is 38 in 36 applications. Table 10 presents the list of 48 vulnerable applications. Fig. 3 shows the number of vulnerabilities found by $\mathcal{W}$ebV$\mathcal{S}$ec in each application of the dataset used for the analysis.

On the other hand, although *BabelView* reported a total number of 19883 leaks (in 881 applications), only 26 leaks are related to interface classes (in 18 applications). Among the 26 leaks, only 18 leaks (in 11 applications) were of type 1 vulnerability. These leaks were reported by our technique, $\mathcal{W}$ebV$\mathcal{S}$ec, as well. The remaining 8 leaks (in 7 applications) were not reported by $\mathcal{W}$ebV$\mathcal{S}$ec because their paths include methods that are neither interfaces nor WebViewClient ones, which means they do not follow our threat model and they are not completely WebView vulnerabilities. However, the 8 leaks use interface classes. The full list of applications and analysis results are available online[6]. *BabelView* did not report any leak related to WebViewClient as $\mathcal{W}$ebV$\mathcal{S}$ec did. This is not surprising as *BabelView* does not target WebViewClient in the first place, which is a drawback of *BabelView*. These results confirm that $\mathcal{W}$ebV$\mathcal{S}$ec is more efficient, precise, and general than *BabelView*.

Table 11 presents 3 examples of applications and the number of leaks reported by *BabelView*. All these leaks are not related to interfaces at all. This confirms the limitations of *BabelView*.

Together with vulnerabilities *BabelView* reports their types depending on their actions. These types include writing to the File System and violation of the Same Origin Policy. The reports provided by $\mathcal{W}$ebV$\mathcal{S}$ec for each analyzed app are rich enough to straightforwardly extract similar vulnerability types. This is so because, for each reported vulnerability, $\mathcal{W}$ebV$\mathcal{S}$ec provides its detailed execution path. The path includes the APIs of source and sink methods. Hence, a simple classification of these APIs is enough to obtain vulnerability classification.

Performance. $\mathcal{W}$ebV$\mathcal{S}$ec and *BabelView* take 27.16 hours and 63.64 hours, respectively, for the total sample of 2000 apps. On average, while $\mathcal{W}$ebV$\mathcal{S}$ec needed 49 seconds to analyze an application, *BabelView* needed 114.9 seconds. Thus, $\mathcal{W}$ebV$\mathcal{S}$ec is more efficient than *BabelView*, as also shown in Fig. 4.

The *Soot* tool Vallée-Rai et al. (2010) that *BabelView* relies on is capable of optimizing the Java byte code during the decompilation of apk files. However, this optimization significantly increases the decompilation time. Although *Soot* optimization can be disabled, slow performance of the default configuration of *Soot* in many tools, such as ded Enck et al. (2011) and Dare

[6] https://github.com/maelzawawy/WebVSec.git

Table 10 – Vulnerable applications reported by $\mathcal{W}$eb$\mathcal{V}\mathcal{S}$ec.

ID	App	ID	App	ID	App
1	Nirvana Kurt Cobain Art LWP	2	Heart Keyboard Themes	3	Viburnumdecoflowers Wallpaper
4	Imagenes Frases de Felicidad	5	Hibernia College Connect	6	Cherry jam Keyboard
7	New Year Keyboard Designs	8	Poker	9	Blue Keyboard Glow GO
10	Rainbow Lips	11	Corner Cafe of Decatur, LLC	12	Pediatric Oncall
13	Estonia Radios	14	Platinum	15	Neon Keypad Green
16	Fb Video Downloader	17	Schadenengel	18	Fandom for IKON
19	Ambiguous	20	Light Fairytale	21	Rabbit speed Keyboard
22	Marry Run	23	Butterfly and Dew Drop	24	Guyana Golden Jubilee
25	Portada de Revista Fotomontaje	26	Martial Arts Pack 2 Live Wallpaper	27	I Love You Greeting Cards
28	Mystic land	29	Fandom for Miss A	30	Vacurect
31	Keyboard Plus Vibrations	32	Golden Age of Civilizations	33	Tumblr Cinnamon aroma
34	Cry of War	35	U.KNOU+	36	VIMO
37	Filler Classic	38	Arch Redfish	39	Parallax 3D
40	StockPortfoliosTrial	41	Bucket Ball	42	(encrypted name)
43	Silk Blue	44	MyRemocon	45	DsCast Music
46	Waterfall Forest Pack 2 Live Wallpaper	47	Neon Keyboard for Galaxy S5	48	Funny Photo Editor Stickers

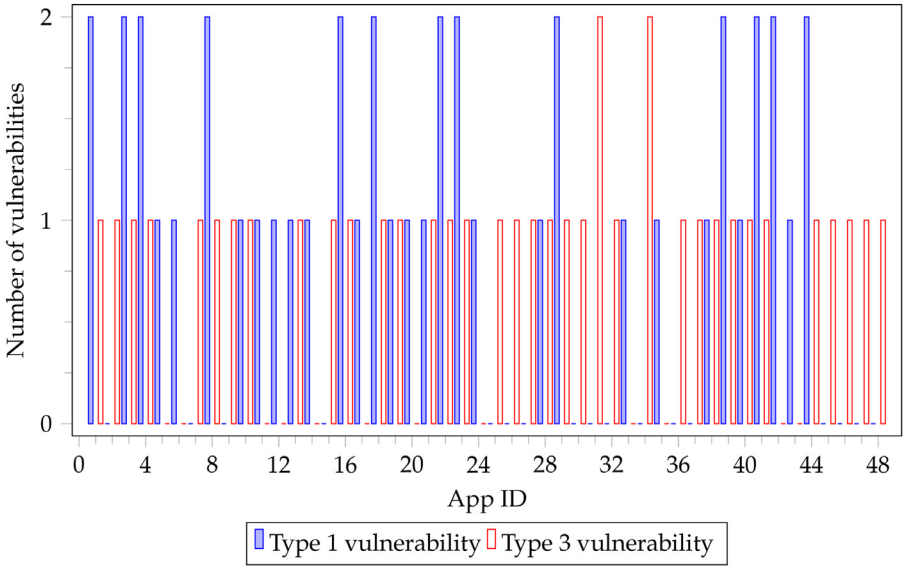

Fig. 3 – Number of vulnerabilities discovered by$\mathcal{W}$eb$\mathcal{V}\mathcal{S}$ec per applications and vulnerability types.

Table 11 – False positive cases for *BabelView*.

App	Leaks#
com.kmungu.kenoWorld 0A3DB67ACA30EBDD9FC9C7450DF489A4DCB846AC41DA9D5DD3B8AF358C2EA4CA	10
com.MIXWIX.MM 5C4B3813437AB4BB6E5E403C75A7F1FE59EC284B8C99623E83EE8AD195032C12	8
com.sadetta.sled E5FD8870738A1DC31A5B6E0CCF00F840C029804B6831FFE37CCEA4E132CE1FDB	32

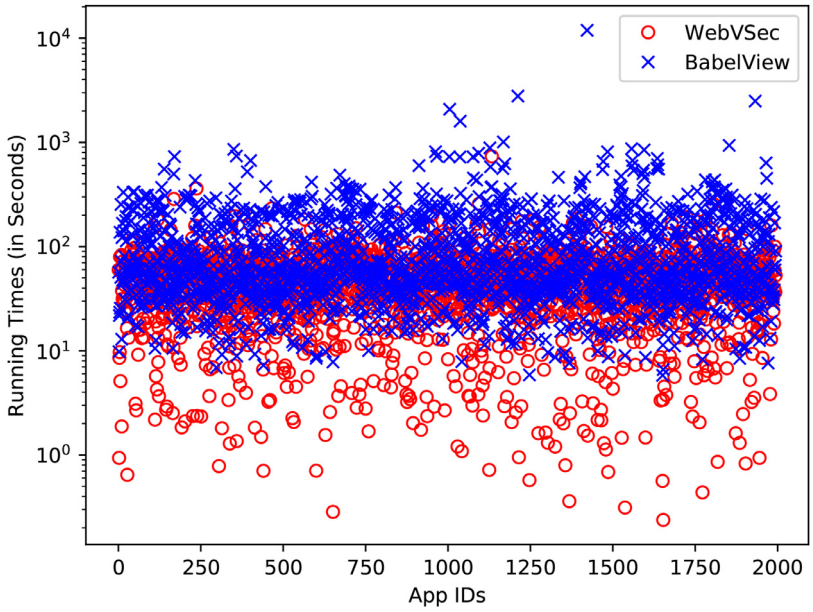

Fig. 4 – Comparing running times of $\mathcal{W}$eb$\mathcal{VS}$ec against *BabelView*.

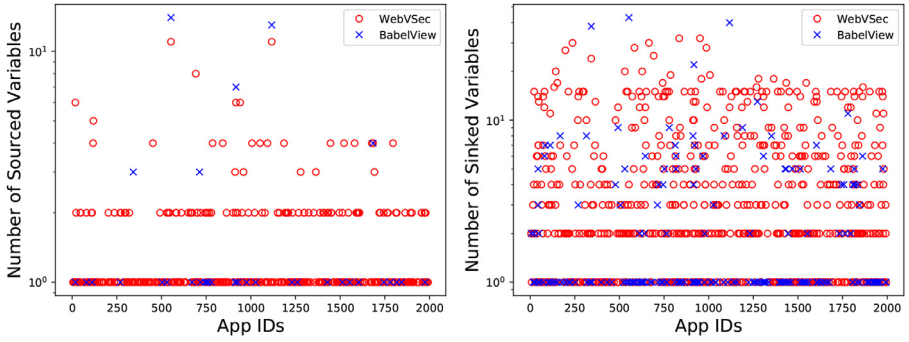

Fig. 5 – Comparing half vulnerabilities in $\mathcal{W}$eb$\mathcal{VS}$ec and *BabelView*.

Octeau et al. (2012), has steered researchers to use other tools (like *Androguard*) as discussed in Reaves et al. (2016). Although *Soot* is widely used, the challenges related to its adoption for analyzing Android apps are not solved yet Reaves et al. (2016): accuracy and soundness arguments Livshits et al. (2015) of *Soot* are examples of these challenges.

8.2. *Half vulnerabilities*

It is interesting to compare the effectiveness of $\mathcal{W}$eb$\mathcal{VS}$ec versus *BabelView* in discovering half vulnerabilities. $\mathcal{W}$eb$\mathcal{VS}$ec reported total number of 626 sourced variables, in 443 applications. The paths of WebViewClient classes hosted 472 of these sourced variables in 409 applications. $\mathcal{W}$eb$\mathcal{VS}$ec also reported 3403 sinked variables in 744 applications. The paths of WebViewClient classes hosted 2016 of these sinked variables, in 668 applications. Considering the leaking paths reported by *BabelView*, we counted 564 sinked variables in interface classes of 184 applications and 72 sourced variables in interface classes of 34 applications. Therefore, $\mathcal{W}$eb$\mathcal{VS}$ec is more effective and useful in discovering half vulnerabilities than *BabelView*. Fig. 5 presents visual comparisons between the capability of $\mathcal{W}$eb$\mathcal{VS}$ec and *BabelView* in revealing half vulnerabilities in applications of the dataset.

8.3. *Manual validation*

We carried out a manual validation to evaluate the accuracy of $\mathcal{W}$eb$\mathcal{VS}$ec which turned out to have no false positives as all reported vulnerabilities are the correct ones. We checked the interface and WebViewClient classes (in Dalvik code format) of randomly selected 50 applications among the ones that were reported OK by $\mathcal{W}$eb$\mathcal{VS}$ec). We did not discover any error in the reports of $\mathcal{W}$eb$\mathcal{VS}$ec for these applications. We also found that the analysis results of our toy application are in line with the source code we developed ourselves. Our manual verification confirms the high accuracy of $\mathcal{W}$eb$\mathcal{VS}$ec.

$\mathcal{W}$eb$\mathcal{VS}$ec is conservative in its nature for two reasons. The first one is that $\mathcal{W}$eb$\mathcal{VS}$ec is path-sensitive, which means that it does not approximate collected analysis data to obtain general results for multiple paths. For example, analysis results for each path resulting from conditional program-

ming commands are built separately rather than approximating one analysis result for such paths. The second reason is that $\mathcal{W}$eb$\mathcal{VS}$ec relies on formal inference rules to abstract and analyze paths. These reasons explain the high accuracy in the 50 apps sample. However, we must stress that the precision evaluation is concluded from analyzing results of 50 apps out of the 2000 apps of the dataset.

8.4. *Current state*

This section discusses the current state of Android applications using WebView, interfaces, WebViewClient, and other different elements that contribute to the vulnerabilities addressed in this paper.

8.4.1. *Interfaces.*

The number of classes containing methods annotated with *@JavascriptInterface* is 3110 classes. These classes belong to 991 applications (49.6% of the dataset). This reveals that annotating interface method is a common practice in implementing WebView. For applications that target minimum SDK less than 17, the number of defined interface classes is 3609 in 1320 applications (66.1% of the dataset). $\mathcal{W}$eb$\mathcal{VS}$ec counted 53367 methods in 1405 interface classes. The interface classes are used in 4444 classes in the 1404 application (70.5% of the dataset).

8.4.2. *Webviewclient.*

The dataset contained 9615 WebViewClient classes in 1749 applications (87.7% of the dataset). The number of methods found in WebViewClient classes is 43772. These classes were used by 15050 classes in 1774 applications (88.9% of the dataset). The number of applications that use WebViewClient is larger than the number of application defining WebViewClient classes because some applications use WebViewClient classes from libraries without needing to define them.

8.4.3. *Relevant methods.*

The number of elements of the set $\mathcal{M}$ of methods selected by the third phase of $\mathcal{W}$eb$\mathcal{VS}$ec (*Identification of relevant methods*) is 500152 (in 1797 applications). On average each application has 278 methods of these methods. By applying the heuristic described in phase 3, the number of methods became 366708 which reduced the average number of methods (needing analysis) to 204 per application. Android creates *access* methods (we denoted them in phase 3 of $\mathcal{W}$eb$\mathcal{VS}$ec as $\mathcal{M}_a$) to access global variables. It is necessary to include these methods in $\mathcal{M}$. $\mathcal{W}$eb$\mathcal{VS}$ec found 13525 *access* methods (in 148 applications).

8.4.4. *Sources and sinks.*

Among all analyzed methods, 2567 methods (in 827 applications) invoked a source-API. The total number of these invocations is 5079. Almost 18% of these methods are WebViewClient ones (in 410 applications) which are responsible for almost 29% of these source-API invocations. On the other hand, 18763 analyzed methods (in 1770 applications) contained 35839 sink-API invocations. Almost a quarter of these methods are webViewClient ones (in 1276 applications) that are responsible for almost a quarter of the sink-API invocations. The exact numbers are shown in Table 9. It is worth noting that although almost 46% of the vulnerabilities discovered by $\mathcal{W}$eb$\mathcal{VS}$ec occurred in webViewClient classes, these classes are responsible for much less percentages of source and sink APIs invocations.

9. Conclusion and future work

Vulnerabilities of WebView objects have been widely studied by previous works, which identified the security issues associated with the JavaScript bridge. In particular, enabling the JavaScript code running in a remote webserver to access smartphone local sensitive data has paved the way for several attacks. Despite the solutions proposed by previous works to defend and detect such attacks, there are still new vulnerabilities affecting this feature of the Android ecosystem, which also impacts a large number of Android applications. The attacker's motivation to design exploits relying on `WebView` vulnerabilities concerns the opportunity to access sensitive data on the phone, just by running some malicious JavaScript code on a remote web server.

In this paper, we propose $\mathcal{W}$eb$\mathcal{VS}$ec a static analysis tool that relies on a set of heuristically designed inference rules to detect four types of vulnerabilities involving Android `WebView` objects. The evaluation of $\mathcal{W}$eb$\mathcal{VS}$ec over a set of 2000 applications led to the detection of 48 vulnerable applications (20 ones having Type 1 vulnerability and 36 having Type 3 vulnerability). On the contrary, *BabelView*, the state-of-the-art approach which we compared with, identified only 20 vulnerable apps (11 having Type 1 vulnerability and 0 having Type 3 vulnerability). Besides being more accurate in detecting vulnerabilities, $\mathcal{W}$eb$\mathcal{VS}$ec also overcomes *BabelView* in terms of efficiency, by requiring 27.16 hours for the analysis of the whole dataset concerning the 63.64 hours required by *BabelView*.

Possible future works include: evaluating the role of `WebView` objects in Android malware detection algorithms; designing a new more secure framework for invoking Java code of Android apps from the JavaScript code of websites; studying the security issues related to invoking JavaScript from Java code in an Android application.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Mohamed A. El-Zawawy: Writing – original draft, Writing – review & editing. **Eleonora Losiouk:** Writing – original draft, Writing – review & editing. **Mauro Conti:** Writing – original draft, Writing – review & editing.

REFERENCES

Allix K, Bissyandé TF, Klein J, Le Traon Y. Androzoo: Collecting millions of android apps for the research community. In: 2016

IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). IEEE; 2016. p. 468–71.
Apache. Apache Cordova. https://cordova.apache.org/, Last access in 2021.
Chin E, Wagner D. Bifocals: Analyzing webview vulnerabilities in android applications. In: International Workshop on Information Security Applications. Springer; 2013. p. 138–59.
Desnos, A., 2011. Android–Androguard: a full python tool to play with Android files. Available from: https://github.com/androguard/androguard/, Last access in 2018.
Developers, G., Building web apps in WebView. https://developer.android.com/guide/webapps/webview, Last access in 2020.
Developers, G., WebView. https://developer.android.com/reference/android/webkit/WebView, Last access in 2020.
Enck W, Octeau D, McDaniel PD, Chaudhuri S. A study of android application security., Vol. 2; 2011.
Fahl S, Harbach M, Muders T, Baumgärtner L, Freisleben B, Smith M. Why eve and mallory love android: An analysis of android ssl (in) security. In: Proceedings of the 2012 ACM conference on Computer and communications security; 2012. p. 50–61.
Ferri, L., Pichetti, L., Secchi, M., Secomandi, A., 2010. Sandbox web navigation. US Patent App. 12/359,457.
Georgiev M, Jana S, Shmatikov V. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks, Vol. 2014. NIH Public Access; 2014. p. 1.
Hassanshahi B, Jia Y, Yap RH, Saxena P, Liang Z. Web-to-application injection attacks on android: Characterization and detection. In: European Symposium on Research in Computer Security. Springer; 2015. p. 577–98.
Hu J, Wei L, Liu Y, Cheung S-C, Huang H. 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; 2018. p. 702–13.
Cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020. 2020, https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. Last access in 2020.
InfoSecurity M. Webview addjavascriptinterface remote code execution. MWR InfoSecurity, Sept 2013.
Jin X, Hu X, Ying K, Du W, Yin H, Peri GN. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security; 2014. p. 66–77.
Jin X, Wang L, Luo T, Du W. Fine-grained Access Control for Html5-Based Mobile Applications in Android. In: Information Security. Springer; 2015. p. 309–18.
Li T, Wang X, Zha M, Chen K, Wang X, Xing L, Bai X, Zhang N, Han X. 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; 2017. p. 829–44.
Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang B-YE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D. In defense of soundiness: a manifesto. Commun ACM 2015;58(2):44–6.
Luo T, Hao H, Du W, Wang Y, Yin H. Attacks on webview in the android system. In: Proceedings of the 27th Annual Computer Security Applications Conference; 2011. p. 343–52.
Luo T, Jin X, Ananthanarayanan A, Du W. Touchjacking attacks on web in android, ios, and windows phone. In: International Symposium on Foundations and Practice of Security. Springer; 2012. p. 227–43.
Mutchler P, Doupé A, Mitchell J, Kruegel C, Vigna G. A large-scale study of mobile web app security. In: Proceedings of the Mobile Security Technologies Workshop (MoST); 2015. p. 50.
Neugschwandtner M, Lindorfer M, Platzer C. In: 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '13). A view to a kill: Webview exploitation; 2013.
Octeau D, Jha S, McDaniel P. Retargeting android applications to java bytecode. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering; 2012. p. 1–11.
Reaves B, Bowers J, Gorski III SA, Anise O, Bobhate R, Cho R, Das H, Hussain S, Karachiwala H, Scaife N, et al. * Droid: assessment and evaluation of android application analysis tools. ACM Computing Surveys (CSUR) 2016;49(3):1–30.
Rizzo C, Cavallaro L, Kinder J. Babelview: Evaluating the impact of code injection attacks in mobile webviews. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer; 2018. p. 25–46.
Singh K. Practical context-aware permission control for hybrid mobile applications. In: International Workshop on Recent Advances in Intrusion Detection. Springer; 2013. p. 307–27.
Statcounter. Statcounter GlobalStats. https://gs.statcounter.com/, Last access in 2020.
Tuncay GS, Demetriou S, Gunter CA. Draco: A system for uniform and fine-grained access control for web code on android. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security; 2016. p. 104–15.
Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A Java Bytecode Optimization Framework. In: CASCON First Decade High Impact Papers; 2010. p. 214–24.
Yang G, Huang J, Gu G. Iframes/popups are dangerous in mobile webview: studying and mitigating differential context vulnerabilities. In: 28th {USENIX} Security Symposium ({USENIX} Security 19); 2019. p. 977–94.
Yang G, Huang J, Gu G, Mendoza A. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE; 2018. p. 742–55.
Yang G, Mendoza A, Zhang J, Gu G. Precisely and scalably vetting javascript bridge in android hybrid apps. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer; 2017. p. 143–66.

Mohamed A. El-Zawawy Mohamed A. El-Zawawy received a Ph.D. in Computer Science from the University of Birmingham in 2007, an M.Sc. in Computational Sciences in 2002 from Cairo University and a BSc. in Computer Science in 1999 from Cairo University. Dr.El-Zawawy is an associate professor of Computer Science at Faculty of Science, Cairo University Since 2014. During the period 2007- 2014 Dr El-Zawawy held the position of an Assistant Professor of Computer Science at Faculty of Science, Cairo University. During the year 2009, he held the position of an extra-ordinary senior research at the Institute of Cybernetics, Tallinn University of Technology, Estonia, and worked as a teaching assistant at Cairo University from 1999 to 2003 and later at Birmingham University from 2003 to 2007. Dr. El-Zawawy is interested in Android security and privacy, IoT security.

Eleonora Losiouk Eleonora Losiouk is a Postdoc Fellow working in the SPRITZ Group of the University of Padova, Italy. In 2018, she obtained her Ph.D. in Bioengineering and Bioinformatics from the University of Pavia, Italy. She has been a Visiting Fellow at the Ecole Polytechnique Federale de Lausanne in 2017. Her main research interests regard the security and privacy evaluation of the Android Operating System and the Information-Centric Networking. During her Ph.D. she published several papers in peer-reviewed journals and IEEE conferences.

Mauro Conti Mauro Conti is Full Professor at the University of Padua, Italy, and Affiliate Professor at the University of Washington, Seattle, USA. He obtained his Ph.D. from Sapienza University of Rome, Italy, in 2009. After his Ph.D., he was a Postdoc Researcher at Vrije Universiteit Amsterdam, The Netherlands. In 2011 he joined as Assistant Professor the University of Padua, where he became Associate Professor in 2015, and Full Professor in 2018. He has been Visiting Researcher at GMU (2008, 2016), UCLA (2010), UCI (2012, 2013, 2014, 2017), TU Darmstadt (2013), UF (2015), and FIU (2015, 2016, 2018). He has been awarded with a Marie Curie Fellowship (2012) by the European Commission, and with a Fellowship by the German DAAD (2013). His research is also funded by companies, including Cisco, Intel, and Huawei. His main research interest is in the area of security and privacy. In this area, he published more than 250 papers in topmost international peer-reviewed journals and conference. He is Area Editor-in-Chief for IEEE Communications Surveys & Tutorials, and Associate Editor for several journals, including IEEE Communications Surveys & Tutorials, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, and IEEE Transactions on Network and Service Management. He was Program Chair for TRUST 2015, ICISS 2016, WiSec 2017, and General Chair for SecureComm 2012 and ACM SACMAT 2013. He is Senior Member of the IEEE. s Full Professor at the University of Padua, Italy, and Affiliate Professor at the University of Washington, Seattle, USA. He obtained his Ph.D. from Sapienza University of Rome, Italy, in 2009. After his Ph.D., he was a Postdoc Researcher at Vrije Universiteit Amsterdam, The Netherlands. In 2011 he joined as Assistant Professor the University of Padua, where he became Associate Professor in 2015, and Full Professor in 2018. He has been Visiting Researcher at GMU (2008, 2016), UCLA (2010), UCI (2012, 2013, 2014, 2017), TU Darmstadt (2013), UF (2015), and FIU (2015, 2016, 2018). He has been awarded with a Marie Curie Fellowship (2012) by the European Commission, and with a Fellowship by the German DAAD (2013). His research is also funded bycompanies, including Cisco, Intel, and Huawei. His main research interest is in the area of security and privacy. In this area, he published more than 250 papers in topmost international peer-reviewed journals and conference. He is Area Editor-in-Chief for IEEE Communications Surveys & Tutorials, and Associate Editor for several journals, including IEEE Communications Surveys & Tutorials, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, and IEEE Transactions on Network and Service Management. He was Program Chair for TRUST 2015, ICISS 2016, WiSec 2017, and General Chair for SecureComm 2012 and ACM SACMAT 2013. He is Senior Member of the IEEE.