Droids in Disarray: Detecting *Frame Confusion* in Hybrid Android Apps

Davide Caputo, Luca Verderame, Simone Aonzo, and Alessio Merlo

DIBRIS - University of Genoa, Viale F. Causa, 13, I-16145, Genoa, Italy.
{davide.caputo,luca.verderame,simone.aonzo,alessio.merlo}@unige.it

Abstract. Frame Confusion is a vulnerability affecting hybrid applications which allows circumventing the isolation granted by the Same-Origin Policy. The detection of such vulnerability is still carried out manually by application developers, but the process is error-prone and often underestimated. In this paper, we propose a sound and complete methodology to detect the Frame Confusion on Android as well as a publicly-released tool (i.e., FCDroid) which implements such methodology and allows to detect the Frame Confusion in hybrid applications, automatically. We also discuss an empirical assessment carried out on a set of 50K applications using FCDroid, which revealed that a lot of hybrid applications suffer from Frame Confusion. Finally, we show how to exploit Frame Confusion on a news application to steal the user's credentials.

Keywords: Frame Confusion · Android Security · Static Analysis · Dynamic Analysis

1 Introduction

Nowadays, the landscape of mobile devices is mostly divided between Android and iOS, with a market share of 74% and 23%, respectively[1]. From a technical standpoint, Android and iOS have remarkable differences both in terms of OS architecture and Software Development Kit (SDK). Such heterogeneity negatively impacts the application (hereafter, app) development process, as companies must rely on different developer teams (be them internal or outsourced) for each platform, thereby increasing the costs of both app development and maintenance. A promising way to overcome the limitation posed by such multi-platform development process is a *cross-platform* framework, which allows to implement an app using a unique programming language and automatically generate a corresponding Android and iOS version. Cross-platform frameworks based on web technologies (i.e., HTML, CSS, and JavaScript), like Cordova [11] or PhoneGap [26], allow for the development of the so-called *hybrid applications*, which combine elements of both *native* (i.e., OS-specific) and *web* apps.
Hybrid apps allow developers to write code based on platform-neutral web technologies and wrap them into a single native app that can render HTML/CSS content and execute JavaScript – like a standard web browser – through a component called *WebView* on Android, and *WKWebView* in iOS. Such component acts as a bridge between the web

[1] `http://gs.statcounter.com/os-market-share/mobile/worldwide`

(i.e., the JavaScript code) and the native world (i.e., the Java or Swift code) through the definition of ad-hoc interfaces. Such interfaces (called *JavaScriptInterfaces* in Android or *WKScriptMessageHandlers* in iOS) allow the developer to define a set of function calls that can be mutually invoked by the two worlds using asynchronous callbacks. As a result, they grant access to the complete set of OS functionality to hybrid apps, thereby making them equivalent to native apps.
However, from a security standpoint, the interaction between the native and the web worlds – which rely on different security models and requirements – can expose hybrid apps to ad-hoc and complex vulnerabilities, like those described in [15,22,23,9,20]. Among them, the *Frame Confusion* vulnerability [22] in hybrid apps has been discovered some years ago and it has been fixed on iOS[2] but not on Android (neither in the latest version, i.e., Android Pie 9.0). To this regard, we argue that a lot of hybrid apps still suffer from such vulnerability and that there is still a lack of i) an extensive analysis of Frame Confusion, ii) a methodology to automatically detect Frame Confusion in hybrid apps, and iii) a reliable solution to mitigate the problem.
Frame Confusion is basically due to the ability of JavaScript code to invoke Android native code through web pages containing at least an *Iframe* element. Such element allows loading external contents (e.g., advertisements, video and payment systems) from domains which differ from the domain of the hybrid app. For this reason, any *Iframe* is in charge of *containerizing* the rendered sub-page, and should execute content only within the scope of its own domain, as prescribed by the Same-Origin Policy (SOP). However, in case of web pages with multiple Iframes, the WebView is unable to identify the Iframe that invokes a function in the native code, and thus the result of the invocation is always executed in the main app page, thereby inducing the confusion problem. Such misbehavior occurs as the *JavaScriptInterface* is bound by the OS to the entire WebView element, without any distinction among the domains (and thus the Iframes) that invoke the function calls. Therefore, the Frame Confusion vulnerability allows to bypass the isolation granted by the Iframe security model and to build a communication channel between web pages belonging to different domains, (i.e., the main app page and the inner Iframes). As a consequence, such vulnerability can affect the confidentiality and the integrity of hybrid apps: a malicious Iframe can, for instance, force the main app to expose private information (like session cookies or internal app files) or mount sophisticated phishing attacks.

Contribution of the paper. In this work, we focus on the Frame Confusion vulnerability on Android. Therefore, hereafter we refer specifically to the Android OS.
Our contribution is three-fold. First, we propose a methodology for systematically detecting the Frame Confusion vulnerability in hybrid apps on Android. Then, we present *FCDroid*, a tool that implements such methodology to automatically identify hybrid apps on Android that suffer from the Frame Confusion vulnerability. FCDroid combines static and dynamic analysis techniques in order to reduce false positive and false negative rates. Finally, we discuss the results of an extensive analysis carried out through FCDroid on a set of $50,000$ apps downloaded from the Google Play Store. The experimental results indicate that the 49.35% of the analyzed apps are hybrid, as they use

[2] `https://cordova.apache.org/docs/en/latest/guide/appdev/security/index.html#iframes-and-the-callback-id-mechanism`

the WebView component and enable JavaScript execution, while about 6.63% of them (i.e., 1637 apps) were found to be vulnerable to Frame Confusion for a total of more than 250.000.000 app installations worldwide. To further validate the proposed methodology, we have manually analyzed some of these vulnerable apps to find out possible attacks exploiting the Frame Confusion vulnerability. To this regard, we were able to exploit Frame Confusion in an Asian news application that has more than 1M users worldwide; such attack allows to steal the user's credential of the primary social media website.

Organization of the paper. The rest of the paper is organized as follows: Section 2 introduces some technical background on hybrid apps and the Frame Confusion, while Section 3 discusses the detection methodology. Section 4 presents FCDroid, while Section 5 shows the experimental results. Section 6 discusses the exploitation of the Frame Confusion on an actual news app, and Section 7 presents some related work. Finally, Section 8 concludes the paper.

2 Technical Background

Landscape of Mobile Apps. Mobile apps can be divided into three categories, namely, i) *native*, ii) *web*, and iii) *hybrid* apps.

Native apps are binary, platform-specific files which are installed on the device and execute by interacting with a set of API calls exposed by the mobile OS. As a consequence, they must be developed in the OS-specific language (i.e., Java/Kotlin for Android and Objective-C/Swift for iOS), and they have full and direct access to the OS API. On one hand, native apps exhibit the best performance for CPU-intensive workloads, while, on the other hand, they need to be re-implemented to execute on a different mobile OS. As this is a daunting task mostly for small-medium enterprises, there is a growing trend towards web or hybrid apps.

Web apps render HTML5 and execute Javascript code within the device browser (which is a native app). For this reason, they are highly portable and platform-independent, but the interaction with the underlying OS is limited to the API accessible by the browser itself. Consequently, they have restricted functionalities and, in general, limited performance.

Hybrid apps have been proposed to overcome the limitations of both native and web apps, namely granting i) portability over platforms, ii) access to the whole OS API and, iii) reasonable performance. Hybrid apps are programmed once in cross-platform web technologies (i.e., HTML, CSS, and JavaScript) as web apps, and then wrapped into a platform-specific native container, i.e., the WebView. The WebView may also allow the interaction between the web and the native part, acting like a *bridge* between the web code and the host OS API, thereby allowing to render HTML/CSS content, execute JavaScript code, and get access to the full OS API.

WebView. The WebView is an Android app component which embeds a mini-browser for rendering HTML/web pages and execute JavaScript code in mobile apps.

The WebView allows defining ad-hoc interfaces, called `JavascriptInterfaces`, that enable to invoke Java methods from the JavaScript code. This feature allows cross-platform frameworks (e.g., Cordova, PhoneGap) to design a set of plugins that can be embedded in apps and offer platform-specific functionality, such as the API for the file-system or the GPS location. To enable JavaScript interfaces, the developer needs to bind a set of Java methods to a WebView component using the `addJavascriptInterface` method. The communication between the JavaScript and the Java code is handled by the WebView using *asynchronous* callbacks. In detail, when some JavaScript code invokes Java code through an interface bounded to the WebView, it does not wait for the result: instead, when the result is ready, the Java code outside the WebView invokes a JavaScript callback function, passing the result back to the web page. This mechanism provides improved app performance and responsiveness, particularly in the case of long-running operations that would block the UI.

WebView Security Mechanisms. As the WebView deals with web content that can include untrusted HTML and JavaScript code, it can suffer from well-known web security vulnerabilities such as cross-site scripting [7,17,6] or file-based cross-zone scripting [9]. As countermeasures, the Android OS includes a set of mechanisms aimed at limiting the capability of the WebView to the minimum functionality required by hybrid apps. By default, the WebView does not execute JavaScript, thus requiring developers to enable this feature using the `setJavascriptEnabled` method. Besides, the application can either enable or disable the access of the WebView to specific resources like files, databases or geolocation [30] through the `WebSettings` object.
Since API 17, the Java methods – which are exposed through a JavaScript interface – need to be explicitly annotated with the `@JavascriptInterface` [16]. The aim is to restrict the access to the OS API, in order to prevent the invocation of any public Java method through code reflection [1].
To further increase the resilience of the WebView component against untrusted contents, since API level 21 the Android OS implements the WebView as an independent app, thus offering a centralized update mechanism that relieves the developer from the burden of manually updating each hybrid app [31].
Moreover, since API Level 26 the WebView renderer executes in a separate process [33]. Finally, since Android 8, the WebView incorporates Google's Safe Browsing protections to detect and warn users about potentially dangerous websites. Unfortunately, this option needs to be explicitly enabled by the developer through a specific tag in the Android Manifest [32].

2.1 Frame Confusion

Frame Confusion is a vulnerability affecting hybrid apps that allows malicious interactions among the main web page (hereafter, *main frame*) and different web domains hosted in inner Iframe elements (hereafter, *child frames*) through the asynchronous bridge between the Java and the JavaScript code, granted by the WebView. To this aim, the WebView maintains a map that links the WebView instance with a list of function calls in the native code registered to the *JavascriptInterface*. However, such a map does not include any restriction on web domains (and thus web pages) that can access the attached interfaces. Thus, if the main frame contains multiple child frames, each of them can independently and asynchronously access all the interfaces bound to the WebView component, in order to interact with the Java code. For this reason, the WebView is

forced to return the results of each native method invocation to the main frame and not to the actual caller, be it the main frame or a child frame, thereby causing potentially unintended interactions between different frames, i.e., the *Frame Confusion*.
Such interaction between the native and the web worlds allows bypassing the Same Origin Policy (SOP), which - in a standard web browser - completely isolates the contents of the main frame from the child frames, since they belong to different domains.

Attacking and exploiting the Frame Confusion. The Frame Confusion vulnerability can be exploited either by using a compromised child frame or the main frame, as shown in Figure 1 (taken from [22]). In detail, if an attacker is able to compromise a child frame (Fig. 1a), he triggers the invocation of native function calls through the WebView App (step 1), which computes the result (step 2) and sends the callback to the main frame (step 3). For instance, a malicious advertisement campaign - embedded in a child frame - can exploit this attack and affects the main frame, by, e.g., inducing an unwanted phone call or force the sending of an SMS.
On the other hand, in case of a compromised main frame (Fig. 1b), the attacker is able to intercept all the callbacks triggered by the child frames, thus leading to possible information leaks. As an example, a benign child frame could inadvertently expose sensitive information like, e.g., the GPS location or the result of a SQL query, to the main frame in control of the attacker, through a native method invocation.

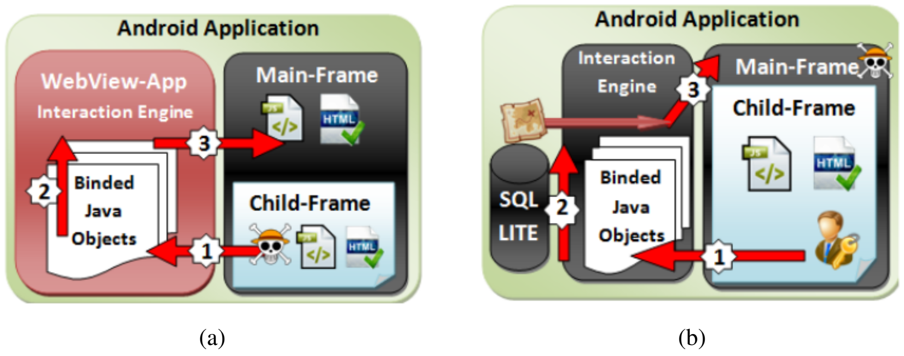

Fig. 1: Exploitation of Frame Confusion from (a) the child frame and (b) the main frame.

The exploitation of the Frame Confusion vulnerability requires the attacker to affect any web domain in the main or a child Iframes that has access to the JavaScript interfaces. This condition is achieved through:

- *The direct control of a web page*. In such a scenario, the attacker can be able either to take control over an existing web domain or to create an ad-hoc website, e.g., a malicious advertisement campaign.
- *The injection of malicious code in an existing web page*. In this case, the attacker can exploit a weakness in the communication protocol of the hybrid app, e.g., a

clear-text communication or a misconfigured SSL connection, to mount a Man-In-The-Middle attack[3] and inject malicious code in the loaded web pages.

It is worth noticing that the presence of other vulnerabilities in the JavaScript code, e.g., the adoption of JavaScript libraries with known vulnerabilities [25] or the presence of XSS vulnerabilities [28,4,6], further boosts the exploiting capabilities of the attacker.

Mitigations. As described above, the Frame Confusion allows violating the SOP by circumventing the sandbox of Iframes. Unfortunately, despite the recent security mechanisms added in the WebView component, the Frame Confusion is still unfixed at any Android API level. Still, the web world offers an extra set of security mechanisms that are able to restrict the communication among the main frame and the child frames, thus preventing the Frame Confusion vulnerability, i.e.:

– the Iframe `sandbox` attribute [29], which enables a set of extra restrictions on any content hosted by an Iframe and, among them, it allows blocking the execution of JavaScript code. Although effective in principle, this mechanism completely prevents the execution of *any* JavaScript code, thus limiting the functionalities of the web page.
– the Content Security Policy (CSP) [10] that allows for the definition of fine-grained restrictions on the execution of JavaScript code, including the possibility to define a set of trusted domains that are able to execute JavaScript, in a white-listing fashion. Although effective against the loading of an undesired web domain, the CSP cannot prevent the injection of the malicious code in a white-listed domain, thereby resulting ineffective against the Frame Confusion.

Furthermore, previous security mechanisms are not enabled by default, thus leaving the burden of their configuration to the developer. All in all, at the current state of the art, none of the existing security mechanisms are able to effectively prevent the Frame Confusion.

3 A Frame Confusion Detection Methodology

The lack of a solution for preventing the Frame Confusion asks for – at least – a methodology to automatically detect such vulnerability. Unfortunately, at the current state of the art, the only way to detect Frame Confusion is through manual source-code inspection, mostly carried out by app developers. Such activity is error-prone and requires good skills in security analysis by the developing team. Furthermore, the complexity of Frame Confusion leads developers to false positives/negatives or, in the worst case, to underestimate or ignore the problem. To overcome this limitation, we propose a novel methodology for the automatic identification of the Frame Confusion in Android. To achieve such result, we first define a blueprint of the Frame Confusion vulnerability, and then we build an analysis flow that is able to detect it automatically, by exploiting a fruitful combination of static and dynamic analysis techniques.

[3] `https://www.owasp.org/index.php/Man-in-the-middle_attack`

3.1 Vulnerability Blueprint

The design of an automatic and rigorous analysis flow for the Frame Confusion vulnerability demands for the selection of a set of features that *enable* the vulnerability. To this aim, we argue that a minimal set of such features is the following:

1. the app requires the Internet permission in order to access web domains using a WebView component;
2. the app uses at least a WebView (`W`) that is configured to execute JavaScript code;
3. `W` sets at least a `JavascriptInterface` (`JI`);
4. `JI` injects at least a public Java method (`m`) that can be accessed from the JavaScript code;
5. in case of an app targeted to API level 17 or higher, `m` needs to be further annotated with the `@javascriptinterface` tag;
6. `W` loads at least a web page (`WP`) that contains one or more Iframe elements;
7. `WP` does not enforce any mitigation technique among those described in the previous section.

3.2 Detection Algorithm

The Frame Confusion detection methodology can be summarized by the pseudocode listed in Algorithm 1. Given a generic Android app in *.apk* format, the algorithm begins by retrieving a list of the Android permissions used by the app (row 1). If the list does not include the Internet permission, then the app cannot use the WebView component, and therefore it is marked as *not vulnerable* (rows 2-4). Otherwise, the algorithm computes the list of all the invoked methods of the app (row 5) in order to locate the presence of `setJavaScriptEnabled`, and `addJavascriptInterface` APIs.
If a `setJavaScriptEnabled` invocation (row 9) is recognized, the algorithm further investigates the flag parameter of the call (rows 10-14). A `True` value indicates that the WebView enables the execution of JavaScript and thus its object reference is retrieved (row 12) and included in the list of those that enable JavaScript (row 13).
Instead, the presence of a `addJavascriptInterface` indicates that a WebView component is configured to expose a bridge between Java and JavaScript. If this is the case, the algorithm extracts *i)* the WebView object from which the `addJavascriptInterface` method is invoked (row 17), and *ii)* the Java object injected in the `JavascriptInterface` (row 18). After that, the algorithm needs to detect if the Java object injected in the interface contains public methods that can potentially be accessed from JavaScript code (rows 19-27). Moreover, in case of apps targeted to API level 17 or above, the public methods of the object need to be further annotated with the `@javascriptinterface` tag (rows 19-22). If the injected Java object contains methods accessible from JavaScript, then the corresponding WebView instance can be added to the list of those that expose potentially vulnerable interfaces (row 21 or row 25).
Next, if the analysis is not able to find at least a WebView - with JavaScript enabled - that contains a JavaScript interface with exposed Java methods, then the app is marked as not vulnerable (rows 29-34). Otherwise, the analysis collects from the Website collector module all the website pages accessed by the WebView that are i) included in the

Algorithm 1: Frame Confusion Detection

```
Input   : APK Package
Output: vulnerable, notVulnerable
listPermissions = getPermissionFromApk(app);
if "android.permission.INTERNET" not in listPermissions then
    return notVulnerable;
end
methodsList = getAllInvMet(app);
JSWebView = list();
IWebView = list();
foreach method in methodsList do
    if method.getName == "setJavaScriptEnabled" then
        flagParam = getFlagParam(method);
        if flagParam == True then
            webViewObj = getInvObj(method);
            JSWebView.add (webViewObj);
        end
    end
    else if method.getName == "addJavascriptInterface" then
        webViewObj = getInvObj(method);
        interface = getInterfaceObj(method);
        if getSDK(app) > 17 then
            if containAnnotatedPubMet (interface) then
                IWebView.add (webViewObj);
            end
        end
        else if containPubMet(interface) then
            IWebView.add (webViewObj);
        end
    end
end
if len (JSWebView) == 0 or len (IWebView) == 0 then
    return notVulnerable;
end
if len (IWebView ∩ JSWebView) == 0 then
    return notVulnerable;
end
resourceFiles = getAllResourceApk(app);
dumpWebStat = getStaticUrl(methodsList);
dumpWebDyn = getDynamicUrl(app);
filesToCheck = dumpWebDyn union resourceFiles union dumpWebStat;
vulnerablePages = list();
foreach file in filesToCheck do
    if isHTMLfile(file) or isJSfile(file) then
        if containIframe(file) then
            if not containCSP(file) and not containSandboxAtt(file) then
                vulnerablePages.add (file);
            end
        end
    end
end
if len (vulnerablePages) > 0 then
    return vulnerable;
end
return notVulnerable;
```

resources of the *.apk* package (row 35), ii) statically invoked by `loadURL` methods (row 36), and iii) dynamically reached during the execution of the app (row 37).
Thereafter, the algorithm collects every website that uses at least an Iframe element that loads an external page (either embedded in HTML pages or generated by JavaScript) and that does not enforce any of the mitigation techniques discussed in the previous section (rows 40-48). Finally, if the app loads at least one vulnerable website, it is marked as *vulnerable*. On the contrary, if the app uses the appropriate security mechanisms or does not use any Iframe is marked as *non vulnerable*.

4 The FCDroid tool

FCDroid[4] implements the proposed detection methodology to automatically identify the presence of the Frame Confusion vulnerability in Android apps.
The rest of the section discusses *i)* the implementation challenges addressed by FCDroid and *ii)* its architecture, emphasizing the underlying tools and technologies.

4.1 Implementation Challenges

The Frame Confusion detection methodology poses several challenges in terms of implementation. Indeed, an automatic detection tool needs to:

1. achieve maximum coverage, i.e., by detecting all possible app execution paths that may lead to the vulnerability;
2. recognize the actual configuration of WebView components, which may *dynamically* enable JavaScript or define new interfaces;
3. analyze all the web pages loaded inside some potentially vulnerable WebViews, by also considering those loaded according to i) the user's input, and ii) the value of runtime variables.

To address such challenges, an automatic tool can rely on static and dynamic analysis techniques. Static analysis techniques can examine all possible execution paths and variable values, not just those invoked during execution. However, static approaches can *i)* introduce false positives and *ii)* be unable to detect complex scenario, like, e.g., values provided by the user or resources loaded at runtime.
On the other hand, dynamic analysis techniques allow to detect the actual behavior of the app, but it is limited by *i)* the coverage of the analysis and *ii)* the time required for the analysis, thus producing potential false negatives.
To this aim, FCDroid combines static and dynamic analysis techniques to overcome the limitations of both techniques and achieve more accurate detection results.

4.2 FCDroid Architecture

The FCDroid architecture, depicted in Fig. 2, is composed by five main building blocks: the Static Analysis Module (SAM), the Dynamic Analysis Module (DAM), the WebSite Dumper (WD), the Frame Confusion Detector (FCD), and the Exploitation Checker (EC).

[4] FCDroid is available at `https://www.fcdroid.com`.

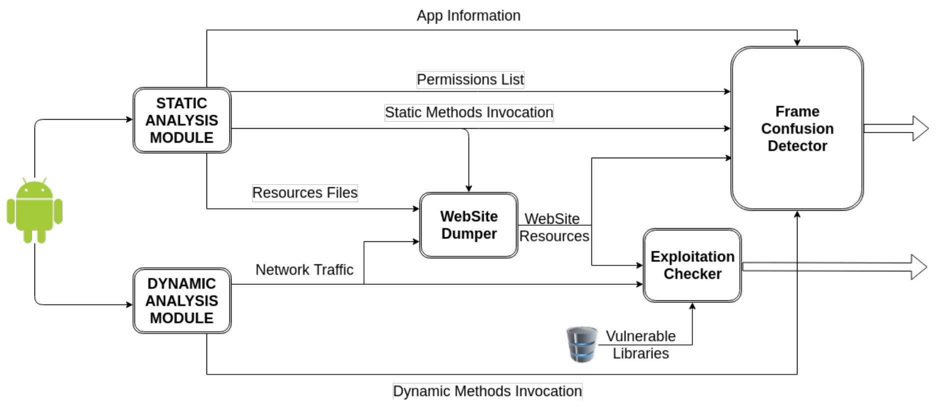

Fig. 2: The FCDroid Architecture.

Static Analysis Module (SAM). The Static Analysis Module relies on *Apktool* [34] to disassemble the app package (in the *.apk* format) and translate the Dalvik bytecode contained in the app into Smali [14] language. In addition to that, SAM brings the resources contained in the app back to their original form, e.g., from binary compiled XML files into textual XML files. Then, the module extracts the list of permissions requested by the app and the target Android API level according to the content of the `AndroidManifest.xml` file. Finally, the SAM inspects each extracted Smali file in order to locate all the API invocations related to the WebView component. In detail, the module detects:

– `setJavaScriptEnabled` that enables the JavaScript code in a WebView object. If found, the SAM also extracts the variable containing the boolean flag passed as an argument;
– `addJavascriptInterface`, that creates a JavaScript interface object. In this case, the SAM retrieves the Java class of the injected object and the name assigned to the interface;
– `loadUrl` and `evaluateJavaScript`, that allows the loading of specific URLs or JavaScript code inside the WebView. In case, the module also extracts the URL address or the script code, if statically defined;

The collected pieces of information are then sent to the WebSite Dumper and the Frame Confusion Detector to continue the analysis.

Dynamic Analysis Module (DAM). The Dynamic Analysis Module is in charge of executing the app into a controlled testing environment in order to monitor the stimulation of the WebView components at runtime. To this aim, it installs the app into an Android Emulator and stimulates the app automatically, trying to explore its possible execution states. This allows the DAM to *i)* monitor the invocations of WebView-related API along with their execution parameters, and *ii)* intercept all the network traffic generated by the app. In order to stimulate the app automatically, the DAM relies on DroidBot [21], an open-source tool that can automatically explore the app UI and mimic the interaction with a user. Unlike many existing input generators that rely on static analysis and instrumentation of the app to generate inputs, DroidBot works in black-box mode, i.e., it does not need to know in advance the structure of the app, and it is resilient to obfuscation techniques. In order to keep track of API invocations, the DAM module provides the Android emulator with an ApiMonitor module. ApiMonitor, based on the Xposed[5] framework, allows the DAM to intercept and collect each method executed by the app during the analysis, saving its invocation and the value of parameters on a JSON file. Furthermore, the DAM module intercepts and stores all the network traffic generated by the app using the HTTP/HTTPs proxy *mitmproxy* [12].

WebSite Dumper (WD). The WebSite Dumper module aims at extracting and retrieving all the websites invoked by the WebView components. To do that, it retrieves from the SAM and DAM modules the list of URLs accessed by app WebView components.
For each identified URL, the WebSite Dumper determines whether it refers to a local or a remote resource. In the first case, it collects and stores the static resource obtained by the app package. In the latter case, the WD module dumps the content of the remote website by downloading the web pages recursively, up to a maximum of 3 levels deep, by using the *wget* tool[6]. Finally, the module polishes the results and maintains only HTML and JavaScript files that will be inspected by both the FCD and the EC module.

Frame Confusion Detector (FCD). The Frame Confusion Detector module implements the core logic of FCDroid for the detection of the vulnerability. At first, FCD collects from the SAM the list of permissions of the app and verifies that the app requires the Internet permission. If so, the module analyzes the list of invoked APIs (both those statically extracted by the SAM module and those evaluated at runtime by the DAM) to verify the existence of at least a WebView instance that enables JavaScript and exposes a JavaScript interface. Furthermore, if an exposed interface is found, the FCD parses the class of the injected Java object to determine the existence of methods that can be accessed from JavaScript.
Finally, the FCD also needs to detect the amount of potentially-vulnerable webpages. To this aim, the module collects the websites dumped by the WD and checks whether a page contains at least an Iframe element and does not enforce any mitigation techniques (i.e., the `Content-Security-Policy` meta tag in the HTML header or the sandbox attribute). At the end of the analysis, the FCD module marks the application as *vulnerable* or *not vulnerable*.

Exploitation Checker (EC). The Exploitation Checker is the module responsible for the detection of app configurations that can boost the exploitation of the Frame Confusion Vulnerability. In details, the EC can identify:

– *The adoption of unencrypted communication channels*, by analyzing the network traffic generated by the DAM module and by extracting the list of URLs that are accessed in plain HTTP.

[5] `https://repo.xposed.info/`
[6] `https://www.gnu.org/software/wget/`

- *The presence of buggy/vulnerable Javascript libraries* by relying on the RetireJS [13] tool, which allows obtaining a list of known-to-be-vulnerable JavaScript libraries that are executed within the WebView.
- *The presence of JavaScript code vulnerable to DOM-XSS attacks*[7], by including a customized implementation of JSPrime [24]. Such tool inspects the JavaScript code in order to detect unsanitized input variables that could allow an attacker to execute arbitrary JavaScript code in the victim's WebView.

5 Experimental Results

We empirically assessed the reliability of the proposed methodology, by systematically analyzing 50.000 apps with FCDroid[8]. Such apps have been downloaded from the Google Play Store in December 2018, and they are the top free Android apps ranked by the number of installations and average ratings according to Androidrank [2]. Our experiments were conducted using an Intel Xeon 3106@1.70 GHz, with 32GB RAM, running Ubuntu 18.04.

Frame Confusion Identification. The FCDroid tool discovered that 49.35% of apps (i.e., 24675 out of 50000) are hybrid, since they use at least a WebView component, thereby highlighting the wide adoption of such component in the Android ecosystem.
As shown in Table 1, all the apps with at least one WebView component enable the execution of JavaScript, while 44.84% also attach (at least) a JavaScript interface, which contains properly annotated methods. Such methods can be invoked from the websites loaded inside the apps.
Furthermore, FCDroid inspected all the websites accessed by the hybrid apps obtaining the results described in Table 2. In detail, 1.2% (87k/6.7M) of websites contain at least an Iframe element; among those pages, the 27.96% include CPS policies while none of the visited pages enforces the sandbox attribute. Therefore, such findings indicate that most of the web pages that use Iframe elements are potentially vulnerable to Frame Confusion. Finally, *our analysis identifies that 6.63% (i.e., 1637) of hybrid apps are potentially vulnerable to Frame Confusion.* To estimate the impact of such results on the Android users' community, we cross-referenced our findings with the Google Play Store meta-data, obtaining that the total sum of official installations for vulnerable apps is greater than 250.000.000.

Exploitation Conditions. We further inspected the vulnerable apps with FCDroid, in order to detect the presence of other vulnerabilities in the app configuration that can make easier the exploitation of the Frame Confusion. As shown in Table 3, 59.98% of vulnerable apps access websites using an insecure connection, i.e., plain HTTP, while 27.48% contain code vulnerable to XSS attacks. Finally, 79.96% of vulnerable apps include JavaScript libraries with known security vulnerabilities.

[7] `https://www.owasp.org/index.php/DOM_Based_XSS`
[8] The complete list of experimental results is available at `https://www.fcdroid.com/results`.

Table 1: Statistics on the Frame Confusion blueprint.

	Percentage	**Ratio**
Internet Permission	96.45%	$48226/50k$
Use WebView	49.35%	$24675/50k$
JavaScript Enabled	49.35%	$24675/50k$
JavaScript Interface	44.84%	$22420/50k$

Table 2: Statistics on the web pages accessed by hybrid apps.

	Percentage	**Ratio**
Web pages with Iframes	1.2%	$87k/6.7M$
Web pages with Iframes and CSP	27.96%	$24108/87k$
Web pages with Iframes and sandbox attribute	0%	$0/87k$

Table 3: Statistics on the exploiting conditions of vulnerable apps.

	Percentage	**Ratio**
Insecure connections	59.98%	$982/1637$
XSS vulnerabilities	27.48%	$450/1637$
Vulnerable JS libraries	79.96%	$1309/1637$

Advantages and Limitations of FCDroid. FCDroid combines static and dynamic analysis techniques to maximize the detection accuracy. To prove that, we compared the analysis results obtained by the static and the dynamic analysis with the hybrid approach of FCDroid, as shown in Fig. 3. In detail, the static analysis allows to identify 12.43% of apps as potentially vulnerable to Frame Confusion. Unluckily, such amount contains both true and false positives. To discriminate, each app should be manually inspected, through a time-consuming and error-prone process. On the contrary, the dynamic analysis is not exhaustive, i.e., it may be unable to reach statically defined web pages, like those hardcoded in the app package, that are loaded in WebView components. Therefore, only 4.69% of apps are successfully detected as vulnerable (true positives) through dynamic analysis.

To overcome such limitations, the mixed approach (i.e., static and dynamic analysis) adopted by FCDroid detected 6.63% of (true positive) vulnerable apps automatically. Indeed, FCDroid allowed to automatically validate more than half of the potentially positive results detected by the static analysis. Furthermore, FCDroid increased the detection rate of the dynamic analysis by 1.94%.

Nonetheless, the experimental results also unveiled some limitations of the current FCDroid implementation. First, the dynamic analysis is limited to the public surface of the app (i.e., the one that does not require any user authentication) and executes in a predefined time-frame (i.e., 60 seconds). Furthermore, the implementation of FCDroid does not detect dynamically-generated Iframe elements, like, i.e., those created at runtime by the JavaScript code.

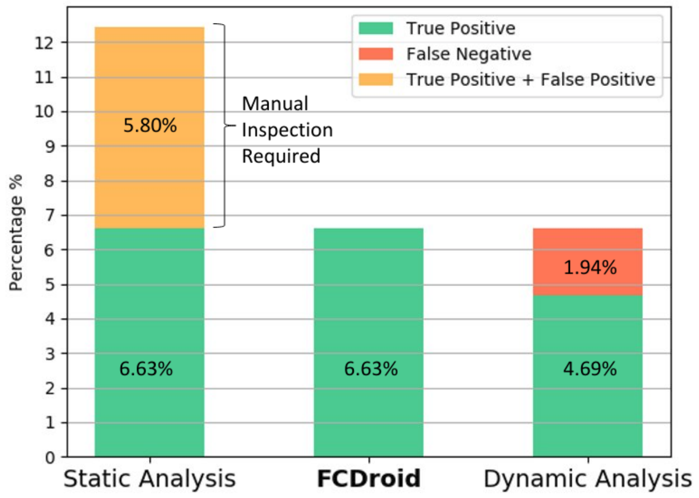

Fig. 3: FCDroid vs Static and Dynamic Analysis

6 Attacking and Exploiting the Frame Confusion

In this section, we discuss the impact of a successful exploitation of Frame Confusion by attacking a news app (i.e., YTN News[9]) which has been found vulnerable by FCDroid. At the moment of writing, YTN is available on the Google Play Store and has more than 1M downloads.
We manually reverse-engineered and analyzed the app: it uses the WebView component to load a home page with several Iframes. The Iframe at the bottom of the web page loads an advertisement (step 1 in Fig. 4). Our manual investigation confirms the FCDroid findings: the WebView uses HTTP, JavaScript is enabled, and there is an interface exposed through `addJavascriptInterface`. The interface exposes different methods that are able to get some information about the device. One of these exposed methods is named `liveLogin`. This method has three parameters of type *string*, the first two are converted into integers and used to customize the WebView, while the last one is passed as a parameter to the `loadUrl` method without any kind of sanitization. Therefore, an attacker can easily inject arbitrary JavaScript code or a web page that will be loaded in the main frame. In order to exploit the vulnerability, the attacker must control an Iframe. There exist two approaches to achieve such result: 1) if the attacker and the mobile device belong to the same network, the attacker can carry out a Man-in-the-Middle (MitM) attack, otherwise, 2) the attacker can create an ad-hoc advertising campaign. In our use case, we carried out a MitM attack, and we were able to control the advertisement (steps 2, 3 and 4 in Fig. 4). For the sake of precision, since the WebView uses HTTP, the attacker can also target the main frame; however, in this example,

[9] `https://play.google.com/store/apps/details?id=com.estsoft.android.ytn`

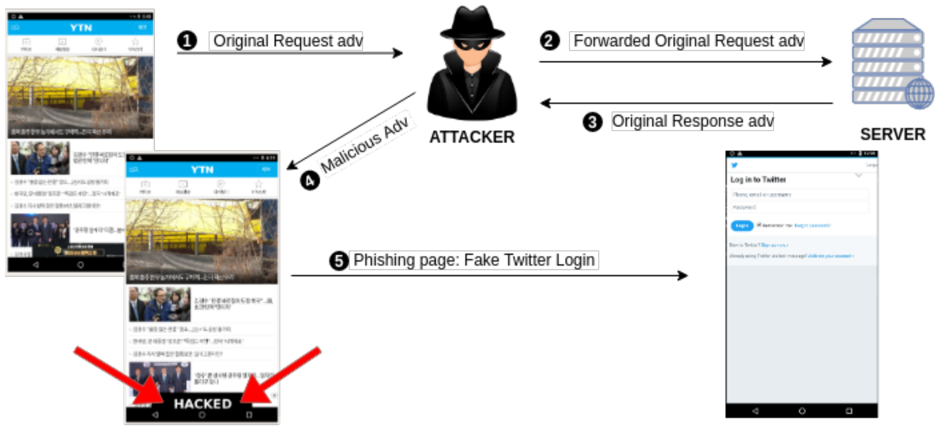

Fig. 4: YTN News: Flow of the attack.

we focused on a child frame, since we only aim to prove the exploitability of Frame Confusion. Thus, given the absence of any security mechanisms, we can access the exposed interface and exploit the Frame Confusion by invoking the `liveLogin` method with a URL pointing to our malicious web page (step 5 in Fig. 4) containing a fake Twitter login page.
It is worth pointing out that the WebView is a promising vector attack for phishing, as there are no GUI components that prompt the actual URL and the transport protocol (e.g., HTTP/HTTPS), thereby making hard to distinguish between the legitimate Twitter website and a well-crafted phishing site [3].
As a final remark, it is worth noticing that this is just one among a set of potential exploitation of Frame Confusion under such specific app settings. For instance, it is possible to download a large file (since the app has the `WRITE_EXTERNAL_STORAGE` permission) or continue to load the same pages within the WebView to carry out simple Denial-of-Service attacks.

7 Related Work

The steady growth of hybrid apps has attracted the attention of both academic and industrial security research communities. The main approaches for the security analysis of hybrid apps can be divided into static and dynamic. In static analysis methodologies, the hybrid app is analyzed according to its source (or binary) code without being executed. For instance, Lee at al. proposed HybriDroid [18], a static analysis framework that examines the inter-communication between the native and the web counterpart of the app to identify development bugs or potential leaks of sensitive information. Other works, like [27], [8], and [35] propose some detection methodologies for *code injection attacks* based on app-instrumentation or machine learning techniques. However, any of the proposed static analysis techniques suffer from the over-approximation of the app execution paths which drastically reduce the accuracy due to a high rate of

false positives [19]. On the other hand, dynamic analysis techniques aim at analyzing the security of the app runtime behavior in a controlled environment. The sole work based on dynamic analysis techniques for hybrid apps is BridgeTaint, proposed by Bai et al. [5]. BridgeTaint tracks sensitive data exchanged through the bridge and uses a cross-language taint mapping method to perform the taint analysis in both domains. Although dealing with the dynamic monitoring of the bridge between the Java and the JavaScript worlds, BridgeTaint is only focused on data analysis aimed at the identification of data leaks. Anyway, none of the approaches mentioned above is either able to identify the Frame Confusion vulnerability. The work proposed by Luo et al. [22] is the only research paper that explicitly discusses this vulnerability. Indeed, the authors – who also coined the term *"Frame Confusion"* – have also studied the security implications of the two-way interaction between the native and the web code in hybrid apps. Anyway, they focus only on detecting the security weaknesses of the WebView component and the JavaScript interfaces, as well as some statistics on the usage of the WebView API and JavaScript interfaces. Furthermore, their analysis is manual, and it has been carried out on a reduced dataset made by only 132 apps.
To the best of our knowledge, our methodology is the first approach allowing us to systematically detect the Frame Confusion vulnerability. Furthermore, the adoption of both static and dynamic analysis techniques allows overcoming the limitations of both approaches.

8 Conclusion

In this work, we have proposed a methodology for systematically detecting the Frame Confusion vulnerability in hybrid Android apps. Then, we have implemented a tool, FCDroid, based on our methodology, which combines static and dynamic analysis techniques to reduce false positive and false negative rates. The results obtained with FCDroid show that Frame Confusion is a concrete problem: among the top 50.000 apps by installations on the Google Play Store, 24675 use the WebView component and we find that 1637 apps (i.e., about the 6.63% among the ones with at least a WebView component) are vulnerable. Although we took into consideration only the top apps, the Frame Confusion already affects more than 250.000.000 installations. Moreover, we have also discovered that about 59.98% of such vulnerable apps load the page within the WebView using a clear-text connection thereby easing phishing attacks.
Future extension of this research will be the study of proper remediations that could prevent the Frame Confusion without disabling the execution of JavaScript inside Iframes in hybrid apps. As a final remark, our attack to the YTN news app[10] suggests that the WebView is a promising vector for phishing attacks, as the user has no way to discriminate whether she is interacting with the legitimate website or a well-crafted phishing one.

[10] We responsibly disclosed our finding to the app developers in January 2019.

References

1. The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface. vol. 9379, pp. 126–138 (2015), http://link.springer.com/10.1007/978-3-319-26096-9_13
2. AndroidRank: Androidrank market data (2018), https://www.androidrank.org/
3. Aonzo, S., Merlo, A., Tavella, G., Fratantonio, Y.: Phishing attacks on modern android. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). Toronto, Canada (October 2018)
4. Backes, M., Gerling, S., Styprekowsky, P.V.: A Local Cross-Site Scripting Attack against Android Phones. Saarland University pp. 1–6 (2011), http://www.infsec.cs.uni-saarland.de/projects/android-vuln/
5. Bai, J., Wang, W., Qin, Y., Zhang, S., Wang, J., Pan, Y.: BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. IEEE Trans. Inf. Forensics Secur. **14**(3), 677–692 (2019). https://doi.org/10.1109/TIFS.2018.2855650
6. Bao, W., Yao, W., Zong, M., Wang, D.: Cross-site Scripting Attacks on Android Hybrid Applications. In: Proceedings of the 2017 International Conference on Cryptography, Security and Privacy - ICCSP '17. pp. 56–61. ACM Press, New York, New York, USA (2017). https://doi.org/10.1145/3058060.3058076, http://dblp.uni-trier.de/db/conf/iccsp/iccsp2017.html
7. Bhavani, A.B.: Cross-Site Scripting attacks on Android webView. International Journal of Computer Science and Network **2**(2), 1–5 (2013)
8. Chen, Y.L., Lee, H.M., Jeng, A.B., Wei, T.E.: DroidCIA: A novel detection method of code injection attacks on HTML5-based mobile apps. Proc. - 14th IEEE Int. Conf. Trust. Secur. Priv. Comput. Commun. Trust. 2015 **1**, 1014–1021 (2015). https://doi.org/10.1109/Trustcom.2015.477
9. Chin, E., Wagner, D.: Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8267 LNCS, pp. 138–159 (2014), http://link.springer.com/10.1007/978-3-319-05149-9_9
10. Content Security Policy: Content security policy (2016), https://content-security-policy.com, https://developers.google.com/web/fundamentals/security/csp/
11. Cordova, A.: https://cordova.apache.org (2018), https://cordova.apache.org/
12. Cortesi, A., Hils, M., Kriechbaumer, T., contributors: mitmproxy: A free and open source interactive HTTPS proxy (2010–), https://mitmproxy.org/, [Version 4.0]
13. Erlend, O.: RetireJS - Scanner detecting the use of JavaScript libraries with known vulnerabilities (2019), https://retirejs.github.io/retire.js/
14. Gruver, B.: Smali - Assembler/Disassembler for the dex format (2019), http://github.com/JesusFreke/smali/
15. Hu, J.: A Tale of Two Cities : How WebView Induces Bugs to Android Applications **1**, 702–713 (2018). https://doi.org/10.1145/3238147.3238180
16. JavascriptInterface: https://developer.android.com/reference/android/webkit/javascriptinterface (2019), https://developer.android.com/reference/android/webkit/JavascriptInterface
17. Jin, X., Hu, X., Ying, K., Du, W., Yin, H., Peri, G.N.: Code Injection Attacks on HTML5-based Mobile Apps. Proc. 2014 ACM SIGSAC Conf. Comput. Commun. Secur. - CCS '14 pp. 66–77 (2014). https://doi.org/10.1145/2660267.2660275, http://dl.acm.org/citation.cfm?doid=2660267.2660275

18. Lee, S., Dolby, J., Ryu, S.: HybriDroid: static analysis framework for Android hybrid applications. Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE 2016 pp. 250–261 (2016), `http://dl.acm.org/citation.cfm?doid=2970276.2970368`
19. Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L.: Static analysis of android apps: A systematic literature review. Inf. Softw. Technol. **88**, 67–95 (2017). https://doi.org/10.1016/j.infsof.2017.04.001
20. 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. Ccs pp. 829–844 (2017). https://doi.org/10.1145/3133956.3134021, `https://acmccs.github.io/papers/p829-liA.pdf`
21. Li, Y., Yang, Z., Guo, Y., Chen, X.: DroidBot: A lightweight UI-guided test input generator for android. Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017 pp. 23–26 (2017). https://doi.org/10.1109/ICSE-C.2017.8
22. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on WebView in the Android system. Proceedings of the 27th Annual Computer Security Applications Conference on - ACSAC '11 p. 343 (2011). https://doi.org/10.1145/2076732.2076781, `http://dl.acm.org/citation.cfm?doid=2076732.2076781`
23. Neugschwandtner, M., Lindorfer, M., Platzer, C.: A View to a Kill: WebView Exploitation. Leet (2013), `http://publik.tuwien.ac.at/files/PubDat{_}223415.pdf`
24. Nishant Das Patnaik, Sarathi Sabyasachi Sahoo : JSPrime (2013), `https://dpnishant.github.io/jsprime/`
25. OWASP: Using components with known vulnerabilities (2017), `https://www.owasp.org/index.php/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities`
26. PhoneGap, A.: https://phonegap.com (2018), `https://phonegap.com/`
27. Rizzo, C., Cavallaro, L., Kinder, J.: BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews (2017), `http://arxiv.org/abs/1709.05690`
28. Sedol, S., Johari, R.: Survey of Cross-site Scripting Attack in Android Apps. International Journal of Information and Computation Technology **4**(11), 1079–1084 (2014)
29. w3: Sandbox attribute (2018), `https://www.w3.org/wiki/Html/Elements/iframe`
30. WebSetting: https://developer.android.com/reference/android/webkit/websettings (2019), `https://developer.android.com/reference/android/webkit/WebSettings`
31. WebView: https://play.google.com/store/apps/details?id=com.google.android.webview&hl=en (2019), `https://play.google.com/store/apps/details?id=com.google.android.webview&hl=en`
32. WebViewSafeBrowsing: https://developer.android.com/guide/webapps/managing-webview (2018), `https://developer.android.com/guide/webapps/managing-webview`
33. WebViewSecurity: https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html (2017), `https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html`
34. Winiewski, R., Tumbleson, C.: Apktool A tool for reverse engineering Android apk files (2018), `http://ibotpeaches.github.io/Apktool/`
35. Yan, R., Xiao, X., Hu, G., Peng, S., Jiang, Y.: New deep learning method to detect code injection attacks on hybrid applications. Journal of Systems and Software **137**, 67–77 (2018). https://doi.org/10.1016/j.jss.2017.11.001, `https://doi.org/10.1016/j.jss.2017.11.001`