# BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications

Junyang Bai, Weiping Wang, *Member, IEEE*, Yan Qin, Shigeng Zhang, *Member, IEEE*, Jianxin Wang, *Senior Member, IEEE*, and Yi Pan, *Senior Member, IEEE*

***Abstract*— Hybrid applications (apps) are becoming more and more popular due to their cross-platform capabilities and high performance. These apps use the JavaScript (JS) bridge communication scheme to interoperate between native code and Web code. Although greatly extending the functionalities of hybrid apps by enabling cross-language invocations and making them more powerful, the bridge communication scheme might also cause some new security issues, e.g., cross-language code injection attacks and privacy leaks. In this paper, we propose BRIDGETAINT, a bi-directional dynamic taint tracking method that can detect bridge security issues in hybrid apps. BRIDGETAINT uses a method different from existing ones to track tainted data: it records the taint information of sensitive data when the data are transmitted through the bridge, and uses a cross-language taint mapping method to restore the taint tags of corresponding data. Such a novel design enables BRIDGETAINT to dynamically track tainted data during the execution of the app and analyze hybrid apps developed using frameworks, which cannot be done with existing solutions based on static code analyses. Based on BRIDGETAINT, we implement the BRIDGEINSPECTOR tool to detect cross-language privacy leaks and code injection attacks in hybrid apps using JS bridges. A benchmark called BRIDGEBENCH is also developed for bridge communication security test. The experimental results on BRIDGEBENCH and 1172 apps from Android market demonstrate that BRIDGEINSPECTOR can effectively detect potential privacy leaks and cross-language code injection attacks in hybrid apps using bridge communications.**

***Index Terms*— Android hybrid application, dynamic data tracking, data leaks, cross-site scripting, JavaScript bridges.**

Manuscript received November 28, 2017; revised April 13, 2018 and June 8, 2018; accepted June 25, 2018. Date of publication July 12, 2018; date of current version August 28, 2018. This work was supported by the National Natural Science Foundation of China under Grant 61672543 and Grant 61772559. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Mauro Conti. *(Corresponding author: Weiping Wang.)*

J. Bai, W. Wang, Y. Qin, and J. Wang are with the School of Information Science and Engineering, Central South University, Changsha 410083, China (e-mail: hs0ring@csu.edu.cn; wpwang@csu.edu.cn; yanqin@csu.edu.cn; jxwang@csu.edu.cn).

S. Zhang is with the School of Information Science and Engineering, Central South University, Changsha 410083, China, and also with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: sgzhang@csu.edu.cn).

Y. Pan is with the Department of Computer Science, Georgia State University, Atlanta, GA 30302-4110 USA (e-mail: yipan@gsu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2855650

## I. INTRODUCTION

WITH the prevalence of mobile devices in recent years, smartphones have become an essential part of our daily life. Various smartphone operating systems (OS), including Google Android, Apple iOS and Windows Mobile, have been widely used. A recent report [1] shows that Android is currently the most popular mobile operating system with 85% market share. As the most commonly used operating system on smartphones, Android also faces many security problems [39]. It is important to detect attacks in Android applications (apps) [6], [11], [13], [23], [41] to protect data privacy of mobile users.

The rapid growth of smartphones has led to the emergence of many feature-rich and diversified mobile apps, ranging from social communication to online shopping. Generally, mobile apps can be classified into three categories: native apps, mobile Web apps, and hybrid apps. Native apps are developed for specific platforms. They are usually developed with platform-specific languages, e.g., Java for Android and Objective-C for iOS. Native apps can invoke application programming interfaces (APIs) provided by the platform to directly access hardware and local resources, making them achieve high performance and satisfy user experiences. However, native apps are tightly coupled with platforms, making them suffer from poor portability. The developers have to build multiple versions for different platforms, which causes high cost in both development and maintenance of the app. In contrast, mobile Web apps are typically implemented with platform-independent Web technologies like HTML, CSS, and JavaScript (JS). They are interpreted to run by web browsers built in mobile platforms, without compiling and installing. However, web code in mobile Web apps, especially those developed with JS, can access only very restricted hardware or local resources due to limitations of the browser environment and security considerations, which makes their performance lower than native apps.

The recently emerged hybrid apps try to achieve both optimized performance and cross-platform capabilities by combining native code and Web code. Because hybrid apps can take advantages of both native apps and Web apps, they are becoming more and more popular in recent years. Moreover, the adoption of the new HTML5 technology in hybrid apps makes their performance very close to native ones. A recent report [3] shows that for a large portion of developers (32.7%) hybrid apps are the preferred development model.

Hybrid apps introduce some new security threats that are different from the ones in native apps and mobile Web apps.

In order to interoperate between the native code and Web code, hybrid apps use a communication mechanism called `JavaScript bridge`. This mechanism provides a method to transmit data through the language boundary, which makes it easier to develop the app. On the other hand, however, JS bridge can also be exploited by attackers to compromise user devices. For example, the attacker can inject malicious code coming from native channels (e.g., Wi-Fi or Bluetooth) into the Web environment via the bridge interface and launch cross-language code injection attacks. If these codes are executed within the web context, they can steal users' private data and even perform other sophisticated attacks, e.g., denial of service attacks. As hybrid apps are more and more popular, it is very important to efficiently detect security issues in hybrid apps.

Although there are few works on tracking tainted data in Java-to-JS communications in hybrid apps (e.g., HybriDroid [22]), they cannot detect bi-directional privacy leaks. Moreover, existing solutions use static code analyses and cannot be used to analyze hybrid apps developed with frameworks like Cordova. Existing works on taint tracking for native apps and mobile Web apps [6], [13], [14], [16], [29], [35], [40], [42] cannot be applied to analyze cross-language data flows in hybrid apps. In this paper, we propose a new method called BRIDGETAINT that can dynamically track bi-directional tainted data in bridge communications to detect cross-language privacy leaks. BRIDGETAINT uses a method different from existing ones to track tainted data: it records the taint information of sensitive data when the data are transmitted through the bridge, and uses a cross-language taint mapping method to restore the taint tag of corresponding data. This novel design enables BRIDGETAINT to dynamically track tainted data during app execution and analyze hybrid apps developed using frameworks. Based on BRIDGETAINT, we implement the BRIDGEINSPECTOR tool to detect cross-language privacy leaks and code injection attacks in hybrid apps using JS bridges. We also develop a benchmark called BRIDGEBENCH for bridge communication security test and evaluate the performance of BRIDGEINSPECTOR on BRIDGEBENCH and other hybrid apps.

To our knowledge, BRIDGETAINT is the first work that can dynamically track tainted data in bi-directions of the bridge for hybrid apps. We summarize the major contributions made in this paper as follows:

- We propose a new method to track cross-language data flow and tainted data for hybrid apps. Different from existing works that uses static code analyses, our method dynamically records the taint information of sensitive data transmitted through the bridge interface and then restores the taint tags of the corresponding data according to the mappings between the recorded JS and Java data. This novel design enables our method to track tainted data in hybrid apps developed with frameworks like Cordova.
- We develop a bridge security test benchmark for Android hybrid apps by using Cordova, named BRIDGEBENCH. BRIDGEBENCH includes 38 test apps, out of which 32 apps have the risk of cross-language privacy leaks. The other six apps are vulnerable to cross-language code injection attacks. BRIDGEBENCH can be used as a benchmark to test the performance for bridge security detection systems.
- Based on BRIDGETAINT, we design and implement BRIDGEINSPECTOR, a cross-bridge data tracking and security detection system for Android hybrid apps. Experimental results on BRIDGEBENCH and 1172 Android market apps show that, BRIDGEINSPECTOR can effectively track the data flows of bridge communication. Meanwhile, it can also detect other attacks to hybrid apps like cross-language code injection attacks.

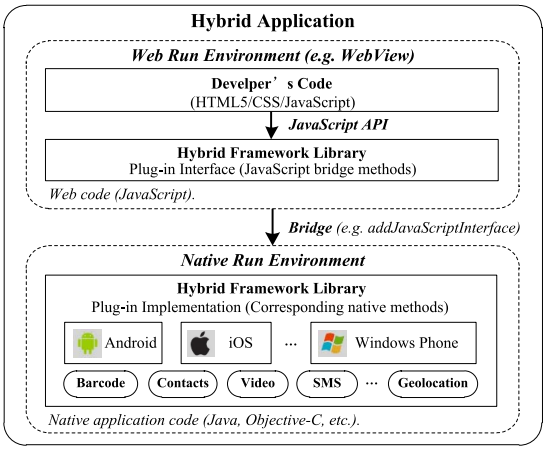


Fig. 1. The general architecture of hybrid apps.

The rest of this paper is organized as follows. Section II describes the architecture of hybrid apps and systematically analyzes the cross-language security risks introduced by JS bridges. Section III presents the mechanism of our cross-bridge taint tracking method BRIDGETAINT. Section IV gives the detailed design and implementation of BRIDGEINSPECTOR. Section V evaluates the effectiveness of BRIDGEINSPECTOR. Section VI compares BRIDGETAINT with HybriDroid. Section VII reviews related work. Finally, we conclude the paper in Section VIII.

## II. SECURITY ANALYSIS OF HYBRID APPS

In this section, we first introduce the general architecture of hybrid apps and analyze the communication mechanism between the native code and the Web code, then systematically analyze the potential security risks caused by the bridge communication, and finally summarize the unsafe data flows that lead to these risks.

### A. Architecture of Hybrid Apps

Figure 1 shows the general architecture of hybrid apps. The code in the lower layer is loaded by the native components and executed in the native runtime environment. The code in the higher layer is rendered by an embedded Web component provided by the mobile platform (e.g., `WebView` on Android) and interpreted by its JS engine. To enable Web code to access the hardware resources and use the platform APIs, the Web component provides a special communication mechanism called bridge communication for interoperation between Web code and native code. The most widely used method to implement bridge communication in Android is to use the `addJavaScriptInterface` API of the WebView.

Some hybrid frameworks provide mobile-specific web libraries and native plug-ins, e.g., barcode, contacts and SMS plug-ins. These plug-ins encapsulate native codes for

accessing hardware resources in different mobile platforms into unified JS bridges. The developers can invoke native functionalities on various platforms by simply adding these plug-ins and calling their bridge methods. This greatly helps developers to create powerful hybrid apps quickly.

### *B. Cross-Language Interfaces*

Although the cross-language communication capabilities provided by different mobile platforms vary slightly, their basic mechanisms are similar. These mechanisms are usually implemented based on APIs within the Web components. Through these APIs, the native code and the web code can invoke each other and transfer data between different languages. In the following, we use the Android framework's communication mechanisms as an example to explain the cross-language communication interface.

Android provides two kinds of communication mechanisms for inter-operation between Java and JavaScript: `bridge communication` and `callback communication`. The first mechanism mainly depends on the WebView's addJavaScriptInterface API. By using this API, developers can inject Java objects into the Web context. The JS code can access the injected objects and call their native methods. During the method invocation, data can be passed between Java and JS as method arguments or method results.

Note that, the cross-language interfaces implemented on this API have serious security issues on all Android prior to version 4.2. More specifically, the JS code can access all public methods of the injected objects. Even worse, attackers can call into the Java Reflection APIs exposed by the bridge interfaces and execute arbitrary commands, causing system-level damages. However, after Android 4.2, Google has patched the above security vulnerabilities of the WebView, and allows JS to invoke only methods explicitly annotated with '@JavascriptInterface'.

The second mechanism is implemented by using WebView events and their callback methods provided by WebViewClient class and WebChromeClient class. For chosen specific events, hybrid app developers can customize their event handlers by overwriting their callback methods corresponding to those events. When the events are fired, the web container will trigger their event handlers and invoke the overwritten Java callback methods. The data are passed into Java as method arguments.

The bridge communication is used much more widely than the callback communication because it supports data delivery more directly. In this paper, we consider the cross-language security risks and data tracking methods for the bridge communication.

### *C. Cross-Language Security Risk Analysis*

In the development process of hybrid apps, the lack of necessary validation and proper sanitization for data transmitted via the bridge interfaces might lead to serious security problems, because these data may come from unsafe third parties and malicious attackers. The risks might come from two different aspects, as analyzed below.

*1) Code Injection Attacks:* The bridge interfaces expand the attack surface of hybrid apps, bringing a serious risk of code injection. For example, with the help of the bridge communication, the attackers can inject malicious JS code into Java, causing Intent injection attacks. Similarly, they can inject malicious native data into JavaScript, causing cross-site scripting (XSS) attacks.

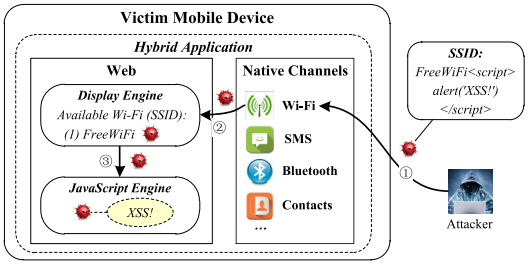


Fig. 2. Attack scenario of cross-language XSS in hybrid apps.

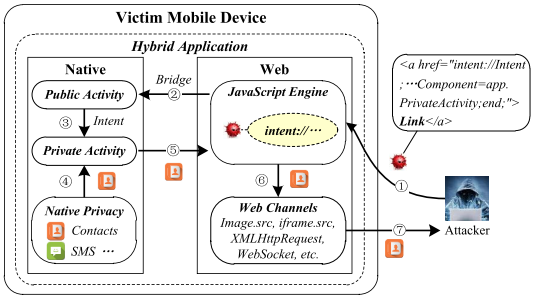


Fig. 3. An example of cross-language Intent injection attacks in hybrid apps.

Figure 2 shows a realistic scenario of such cross-language XSS attacks in a Wi-Fi scanning app. The app uses JS bridges to acquire nearby Wi-Fi access points (APs) and directly displays the information of available APs in the page via the unsafe document object model (DOM) APIs. To launch the attack, the attacker first forges a Wi-Fi site and injects the malicious JS code into its SSID (①), and then waits for the victim to connect to the fake Wi-Fi. Once the victim starts the app to scan nearby Wi-Fi (②), the display engine will parse the SSID and invoke the JS engine to execute the malicious code (③). Compared with web apps, hybrid apps have a wider range of channels that can be leveraged to attack app users. The attackers can exploit Wi-Fi, SMS, Bluetooth or other channels to launch powerful cross-language XSS attacks.

Figure 3 illustrates an example of cross-language Intent injection attacks. Intent is an important mechanism for application interaction and communication in the Android framework. It is usually used to start an app's internal private components or other apps' components. In this example, the attacker first injects a malicious Intent link (①) into the Web page of the app in some way (e.g., through malicious ads). Because the app does not properly sanitize and validate the Intent, it will directly parse the Intent when the victim user clicks the link (②) and invoke the internal private component of the app (③-⑦). Compared to the public components, the private components can often obtain personal data stored in the app. Thus, the attacker can further exploit the private components to steal user privacy or perform permission escalation attacks.

*2) Privacy Leaks:* The bridge interfaces also open channels of information disclosure, posing a significant risk of privacy leaks to mobile users. In the traditional native apps, the privacy preserved in mobile devices can be accessed by only platform-specific code. However, with the help of hybrid apps, the attacker can use JS bridges to steal private user data across

TABLE I
SOURCES AND SINKS

| | Environment (language) | Sensitive objects of sources and security-critical APIs of sinks |
|---|---|---|
| Sources | Native (Java) | *NFC**, *Contacts**, Mic, PhoneNumber, Location(GPS), Location(Net, Last), *Bluetooth**, Camera, Accelerometer, *SMS**, Device(IMEI, IMSI), *Wi-Fi**, ICCID, *Barcode**, Account, History |
| | Web (JavaScript) | Doc.cookie, Doc.domain, Doc.lastModified, Doc.referrer, Doc.title, *Doc.URL**, *Loc.hash**, Loc.host, Loc.hostname, *Loc.href**, Loc.pathname, Loc.port, Loc.protocol, *Loc.search**, *WebSocket.onmessage**, *WebStorage.getItem** |
| Sinks | Native (Java) | sendSms#, Java.io.OutputStream# |
| | Web (JavaScript) | Loc.href#, XMLHttpRequest.send#, WebSocket.send#, Iframe.src#, Image.src#, *Doc.write*$, *Doc.writeln*$, *Eval*$, *setTimeout*$, *setInterval*$, *Element.innerHTML*$, *Element.outerHTML*$ |

Doc: document object. Loc: location object. *: Untrusted data sources. #: Network communication APIs. $: Code execution APIs.

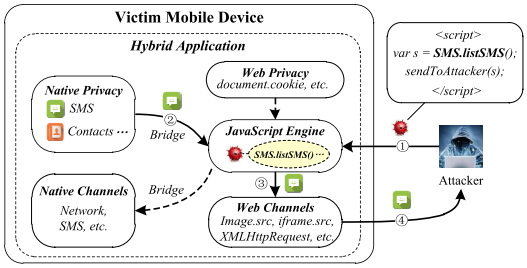


Fig. 4. An attack scenario of stealing user data via bridge interfaces in hybrid apps.

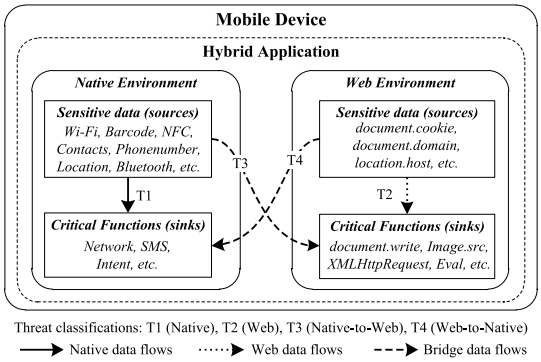


Fig. 5. Categories of security threats in hybrid apps.

different platforms, exposing massive users to a dangerous situation.

Figure 4 demonstrates how the attacker uses web code to steal privacy of mobile users. The attacker first injects a malicious code into the Web pages (①). The code contains a JS bridge method, which can read and retrieve all the SMS messages stored on the mobile device. When the hybrid app loads the Web page, the malicious code is executed by the WebView's JS engine (②) and the victim's SMS data is eventually sent to the attacker via the Web channels (③-④). Similarly, the attacker can also use malicious Java code to steal online privacy preserved in web browsers, such as the user's cookies.

### D. Security Threats Summary

*1) Classification:* Actually, the essential reason of the above risks is that the sensitive data directly flow into the security-sensitive functions, resulting in unexpected operations of applications. According to the programming languages in which the data generation points (called `sources`) and the data receiving points (called `sinks`) are located, we classify the associated security threats of hybrid apps into four categories: native, web, web-to-native and native-to-web (shown in Figure 5).

The native and web threat are security issues introduced by single programming language, as their corresponding data sources and receiving functions lie in the same language code. For instance, a simple example is that the SMS content are sent through Java network APIs, causing native data disclosure. The other example is that web cookies are leaked over XMLHttpRequest() method.

Different from the first two categories, the latter two ones are cross-language threats caused by multiple programming languages. It means that through the bridge interfaces, the data of sensitive sources in one language code may reach the security-critical point of another language. The cross-language code injection and privacy leaks described in subsection II-C are the typical examples of such threats.

*2) Sensitive Data Sources and Security-Critical Sinks:* In order to identify different kinds of security risks in hybrid apps, we divide the sensitive data sources into two types: private sources and untrusted sources. We also divide the security-critical sinks into two types: communication sinks and code execution sinks. Table I lists all the sources and sinks considered in this paper.

*a) Sources:* The private sources mainly contain important information stored on the mobile device or the Web browser. These sources usually include personal data (e.g., phone number, contact lists and location), device information (e.g., IMEI/IMSI number), and online privacy (e.g., session cookies). It will cause information disclosure when these private data are directly passed to the network APIs over HTTP channels. Different from the private sources, the untrusted sources mainly refer to the data that can be directly manipulated or indirectly contaminated by attackers (identified by the * in Table I), such as data obtained through the native external channels (such as Bluetooth, Wi-Fi) or internal channels (e.g., SMS), and users' searching keywords preserved in the properties of JS objects (such as location.search). In addition, because WebSocket messages and Web Storage data [7] can be easily tampered and constructed by attackers as well, these data are also treated as untrusted sources. While the risk of leaking untrusted data is much smaller than private data, serious code injection attacks may be launched once these untrusted data are transferred into the code execution functions.

*b) Sinks:* The sinks are security-critical JS and Java APIs that provide the ability of network communication and code execution (denoted by the # and the $ in Table I, respectively). The commonly used communication APIs include Java methods OutputStream() and sendSms(), and

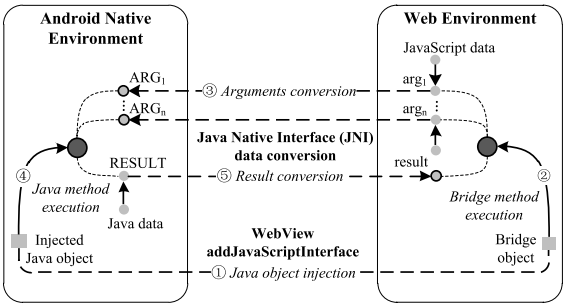


Fig. 6. The data transmission mechanism of the bridge communication.

```
1  /*  Android Native Java code  */
2  @Override
3  public void onCreate(Bundle savedInstanceState) {
4    WebView w = new WebView(this);
5    WebSettings ws = w.getSettings();
6    ws.setJavaScriptEnabled(true);
7    w.addJavascriptInterface(new JavaObject(), "bridge");
8    w.loadUrl("file:///android_asset/index.html");
9  }
10 class JavaObject {
11   @JavaScriptInterface
12   public String send(String data) {
13     func2(data);
14     tm = (TelephonyManager) this.getSystemService(Context.TELEPHONY_SERVICE
           );
15     String c = tm.getDeviceId();
16     return c;
17   }
18 }
19 /*  JS code in index.html  */
20 var a = document.cookie;
21 var b = bridge.send(a);
22 func1(b);
```

Listing 1. A snippet code of cross-language data transmission in hybrid apps.

JS method XMLHttpRequest(). Besides, the notorious Eval() method, and the setTimeout() method are the typical code execution sinks.

In the previously described four types of security threats, the native and web threats can be detected by many research tools by using data flow tracking methods. The tracking idea is to first mark the data of sensitive sources with taint tags, then propagate the tags by static analysis or dynamical execution of the program, and finally check the taint tags of the function parameters at the critical sinks to determine whether the sinks can be affected by the tainted data.

However, for the native-to-web and web-to-native issues, currently there is no systematic detection method that can analyze their cross-language data flows. Jin *et al.* studied the native-to-web code injection attacks and proposed a static taint analysis method to detect such attacks, but they did not consider privacy leaks and web-to-native threats. Therefore, the main focus of this paper is to provide a general solution for these two kinds of cross-language security threats, and to propose an effective mechanism for cross-bridge data tracking and security detection.

## III. Bi-Directional Cross-Bridge Taint Tracking

As described in Section I, the traditional taint tracking methods cannot analyze data flows of the bridge communication that cross language boundaries. In this section, we present a novel cross-bridge taint tracking method called BridgeTaint, which can bidirectionally propagate the taint markings of sensitive data between different languages.

### A. Data Transmission of Bridge Communication

To achieve bi-directional cross-bridge taint tracking, we first need to figure out how the data is transmitted across the bridge interfaces. To this end, we systematically inspect the semantics of data transmission in the bridge communication of Android. Briefly, when a given bridge method is executed by Android, JS data are passed into the Java environment as the method arguments, and Java data are returned to the JS environment as the method results. The detailed process of the bridge method invocation and data transmission are shown in Figure 6, containing the following five steps:

① *Java object injection.* The developer first uses the API addJavaScriptInterface to inject a Java object (called `injected object`) into the Web context. This API has two parameters: The first is the name of the injected object, and the second is the name of the exposed JS object (called `bridge object`) that refers to the injected object.

② *Bridge method execution.* When the WebView calls the method of the bridge object (called `bridge method`), the Android system first checks whether the target Java object corresponding to the bridge object exists or not, and then searches the corresponding Java method from the method list of the target Java object by using argument numbers and argument types.

③ *Arguments conversion.* When Android finds the target Java method, it converts the argument values of the bridge method to the corresponding Java values by `JNI` (Java Native Interface) data conversion mechanism of . In this way, Android transfers the JS data into the Java environment.

④ *Java method execution.* After completing the arguments conversion, the Android system executes the target native method with the converted argument values.

⑤ *Result conversion.* Once Android finishes the execution of the native method, it converts the result value to the corresponding JS value by using JNI and returns the result to the WebView.

To better understand the data transmission process and the threats of the bridge communication, we use a simple code snippet (shown in List 1) of a hybrid app to illustrate how sensitive data is passed between JS and Java by the attackers. This malicious app uses JS bridges to steal the user's IMEI number and cookies. First, the attacker declares a Java class named 'JavaObject' including a 'send()' method (line 12-18). Then, the attacker injects an instance of 'JavaObject' into the WebView as a JS object named 'bridge' (line 7) via the 'addJavaScriptInterface'. When the app loads the 'index.html' page, the malicious JS code gets the cookie (line 20) and sends it to Java by calling the 'bridge.send()' method on line 21. The cookie is then transferred to the corresponding Java method 'send()' on line 12. The method sends the received cookie to the attacker (line 13) via the 'func2()' method, and returns the IMEI to the JS (lines 15-16). Finally, the IMEI number is leaked to the attacker through the JS method named 'fun1()' on line 22. To detect such sophisticated data leaks and other potential threats in cross-language environments, we propose BridgeTaint, which can track data flows crossing the bridge interfaces.

### B. Cross-Language Taint Tracking

From previous description, we know that the cross-language data transmission in the Android framework mainly relies on two types of system mechanisms: the bridge method invocation and the JNI data conversion. Therefore, to accomplish the cross-language taint tracking, we must arrange the taint propagation scheme within these mechanisms. Considering the complexity of the JNI, the WebView, and the underlying

TABLE II
THE BRIDGE SINKS AND THE BRIDGE SOURCES OF BRIDGETAINT

| Types | Environment (language) | Descriptions | Operations |
|---|---|---|---|
| Bridge Sinks | Web (JavaScript) | JavaScript bridge methods | Taint logging |
| | Native (Java) | Corresponding Java methods related to the bridge methods | Taint logging |
| Bridge Sources | Web (JavaScript) | Result value of bridge methods in JavaScript | Taint recovery |
| | Native (Java) | Argument values of the corresponding Java methods | Taint recovery |

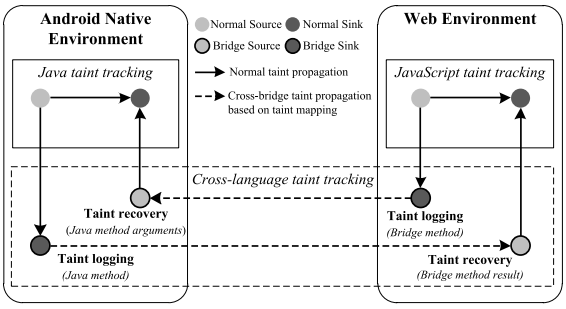


Fig. 7. Diagram of BRIDGETAINT.

data structures, the cost of implementing a complete taint propagation scheme directly in these components would be too high and the scalability would also be poor. Thus, we present a simple and effective approach to propagate taint tags bidirectionally between the JS and the Java environment.

The approach indirectly shares and exchanges the taint information of sensitive data from different languages by performing different operations in two stages. In the first stage, we carry out a `logging` procedure to record the taint information of sensitive data in the related methods of cross-language interfaces before the data are converted by JNI. Then in the second stage, we perform a `recovery` operation to restore the taint tag of the corresponding data according to the recorded tag in the first stage. Figure 7 is the diagram of BRIDGETAINT, in which the solid line part is the traditional taint tracking method for specific languages, and the dashed line part is the taint propagation mechanism for the bridge interfaces.

*1) Bridge Sinks:* We refer the methods implemented based on the bridge communication as the `bridge sinks` (shown in Table II). These sinks include all the JS bridge methods and their corresponding Java methods in hybrid apps, which provide the capability for transmitting data across different languages. When the sensitive data flows into these special sinks, BRIDGETAINT records the transmitted data and provides mappings for sensitive data between multilingual environments. The recorded information includes the argument values of the bridge method and the return value of the Java method. In the example code shown in List 1, the recorded data are the variables `a` (line 21) and `c` (line 16) in the 'bridge.send()' method and the 'send()' method, respectively. The detailed recorded contents and information will be described in the **Taint Logging** subsection.

*2) Bridge Sources:* During the execution of bridge methods, Android passes the sensitive data from one language to another by using the JNI conversion. The converted data corresponding to the sensitive data are called the `bridge sources`. Thus, our goal in the second stage is to locate these converted data and restore their taint tags, and to track these data as new sources in the further tracking. For a specific bridge method in hybrid apps, the bridge sources include the result of the bridge method and the arguments of its corresponding Java method. While the sensitive data are transmitted by the bridge methods, BRIDGETAINT restores the taint tags of the data in these sources according to the recorded cross-language mappings. Thus, the taint tags are indirectly propagated from one language to another language. In the example given in List 1, the corresponding mapping data to the variables 'a' and 'c' are the variables '`data`' (line 12) and 'b' (line 21), respectively.

*3) Precise Mapping Between JS and Java:* Considering the diversity of data and bridge methods in hybrid apps, it is necessary to obtain and log the concrete data types and the transmitting bridge methods of a given sensitive data, to achieve accurate taint mappings between JS and Java. To this end, the logging procedure must first ensure that the recorded taint information of the transmitted sensitive data is accurate. Besides, the recorded data types and the method name also need to be unique; otherwise the restore operation cannot locate the corresponding bridge source related to the transmitted data.

*a) Taint logging:* To ensure the uniqueness of data mappings and distinguish between different bridge methods with arguments of different data types, we record complete relevant information of the transmitted data. The detailed recorded information includes the bridge method name, as well as the taint information of its parameters and the result of the corresponding Java method. We represent the recorded information as a triple (M, A, R). The taint information A and R are also a triple (S, V, T), where S is the data type, V is the original value and T is the taint tag in the Web and the Java environment.

Meanwhile, in order to assure that the recorded information is accurate, we arrange the logging operation in the underlying method of the JNI conversion. Because all the data transmitted between Java and JS need to go through the JNI interface, we log the relevant data in the implementation method of the interface. This arrangement can ensure the accuracy of the recorded information, and can also assure that there will be no side effects on the taint propagation, such as the loss or amplification of the taint tag. The detailed implementation will be described in subsection IV-B.

*b) Taint recovery:* We use two steps to guarantee the correctness and the effectiveness of taint recovery for a given bridge source. First, BRIDGETAINT finds the correct record of the bridge method that is related to the bridge source from taint logs. The record contains its original taint tag, which is logged when its corresponding data are crossing language boundaries. Second, its taint tag is restored to the logged tag in the matched record. Specifically, the taint tag of the result value of the bridge method and the parameter values of the native Java method are restored according to the return and arguments field in the taint record, respectively.

Figure 8 shows the taint recovery process of the example shown in List 1. When BRIDGETAINT detects the bridge

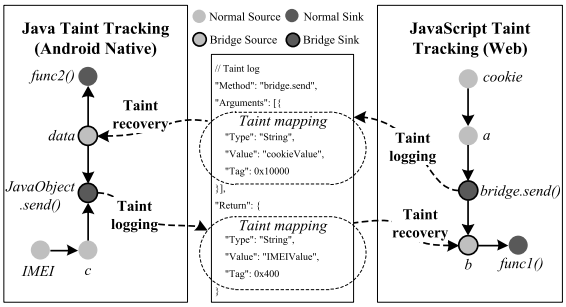


Fig. 8. Example for taint mapping of BRIDGETAINT.

sources (as shown in List 1, variable '`data`' and '`b`'), it searches the corresponding taint record associated with the bridge method 'bridge.send()'. And then, it assigns the taint tag of 0x400 and 0x10000 in the matched record to the bridge method's return value (variable '`b`') and the Java method's parameter (variable '`data`').

## IV. IMPLEMENTATION

In this section, we describe the implementation details of our cross-language taint tracking system called BRIDGEINSPECTOR. In BRIDGEINSPECTOR, we extend the sensitive sources of `TaintDroid` [13] to track the data flows of Java layer in hybrid apps, and utilize `JSTA` [34] to analyze the data flows of JS layer. The BRIDGETAINT method is used to propagate the taint tags between the Java and the JS environment.

### A. System Architecture

BRIDGEINSPECTOR is an offline detection system that can be leveraged to detect cross language security threats in Android hybrid apps. The system consists of three components, as shown in Figure 9. The Java and JS taint tracking components are applied to track sensitive Java and JS data, respectively. The cross-bridge taint propagation component is used to log and map sensitive data transmitted across language boundaries. The cross-bridge taint propagation component builds a bridge to spread the taint tags between the Java and JS taint tracking components.

### B. Java Taint Tracking Component

The Java taint tracking component tracks Java data in the app with a customized TaintDroid system. The customization is based on the original TaintDroid and extend its taint sources.

TaintDroid is an efficient system-level taint tracking system that analyzes and detects privacy leaks in Java code of Android apps. TaintDroid implements data tainting within the Java runtime environment, namely the Dalvik Virtual Machine (VM) interpreter. First, the system marks the sensitive data at the system interfaces where the data are accessed. It allocates an additional 32-bit memory tag for each sensitive data to record the taint status of the variable. Each bit in the tag relates to one of 32 kinds of sensitive sources, and the tag can represent at most 32 different sensitive sources. When executing Java bytecode, the system propagates the taint tags of variables according to the predefined taint propagation rules. Finally, the system checks whether the variable are tainted or not at the sinks to detect sensitive data leakage.

The original TaintDroid only defines 12 private Java sources listed in Table I. We extend TaintDroid to track the other 22 sources listed in the table.

Considering the potential security threats (e.g., privacy theft and code injection) to WebView, we add 6 Java taint sources and 16 JS taint sources into TaintDroid. The added sources are used to track the unsafe Java data channels and sensitive JS objects marked with * in the Table I. In the implementation, we insert the taint tags into the memory of the corresponding variables at the system interface where the data is fetched.

After extension there are totally 34 different types of taint sources. However, TaintDroid can only represent 32 sources with 32-bit taint tags. To fit to the 32 bit tag, we merge some similar sensitive sources. Specifically, the IMEI and IMSI number are merged into one source called the device identifier, and the Net-location and Last-location data are merged into one source called geographical sources, respectively.

The extended TaintDroid system has 32 different taint sources, as shown in Table I. The taint markings of sensitive sources are number from 0x00000001 to 0xFFFFFFFF following the order in the table.

### C. JS Taint Tracking Component

To implement JS taint tracking, we develop the JSTA tool to analyze and detect the flows of JS sensitive data. The taint sources in JSTA and their taint markings are consistent with the customized TaintDroid system, as shown in Table I.

The basic idea of JSTA is to insert the complete taint logic into the JS code through code rewriting, so that the rewritten code can automatically track sensitive data when they are executed. To mark taint sources, JSTA first converts the original sensitive data in the code to JS objects and adds an extra object property to store their taint tags. JSTA then encapsulates the operators of basic expressions (arithmetic, logic, assignment, etc.) and control statements (conditions, loops) as functions. These encapsulated functions complete the original data operations and spread the taint tags between operands. Finally, JSTA hooks critical sink functions to detect whether their parameters are tainted or not.

JSTA's code rewriting is implemented based on abstract syntax tree (AST) transformations. First, JSTA parses the JS code into an AST. It then traverses the AST and transforms it to a new AST according to a set of rewriting rules. These rules contain transformation rules of data conversion, function encapsulation and sink hooking described earlier. Finally, JSTA generates traceable JS code from the new AST.

### D. Cross-Bridge Taint Propagation Component

When TaintDroid and JSTA detect that there are sensitive data crossing the language boundaries, the cross-bridge taint propagation component is triggered to spread the taint tags of these data between the two systems by mapping tainted Java data into JSTA, and mapping tainted JS data into TaintDroid.

*1) Java-to-JS Taint Propagation:* As shown in Figure 9, the taint propagation from Java to JS is accomplished by two modules. First, the Java logging module records the tainted Java data transmitted by the bridge methods. After that, the Java-to-JS mapping module restores the taint tags of the corresponding JS data based on the recorded taint relationship between Java and JS data.

*a) Java taint logging module:* The bridge communication of Android is implemented in the function of the JavaInstanceJobjectV8 class named JavaInstanceJobject::invokeMethod. This function is used to interpret and execute the Java method invoked in JS code, and to perform JNI data conversions on

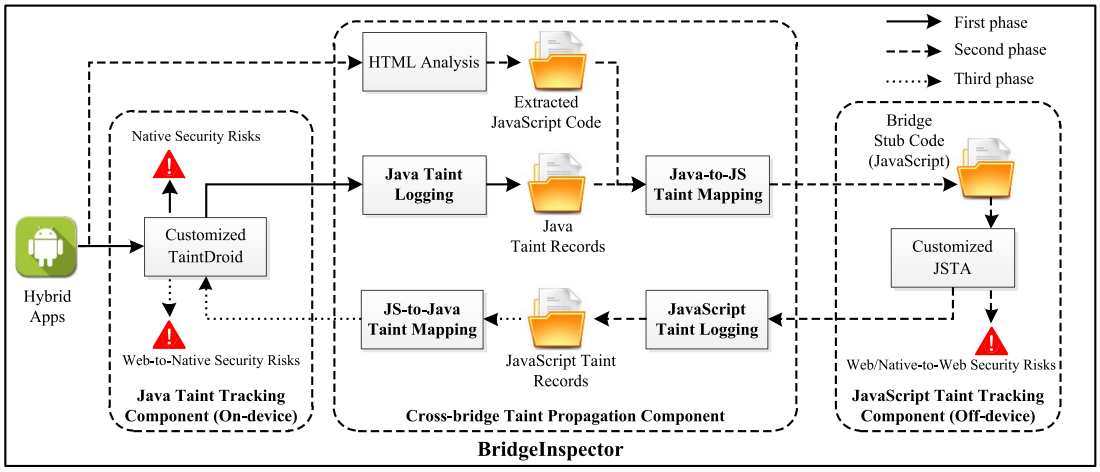


Fig. 9. The architecture of BRIDGEINSPECTOR.

```
{   "Method": "bridge.send",
    "Return": {"Type": "String", "Value": "IMEIValue", "Tag": 0x400 }}
```

Listing 2. An example Java taint record in JSON format.

```
/*  Automatic generated stub code of JS bridge  */
bridge = {  send: function() {
    // Return tainted Java data to JavaScript.
    return new taintedJavaData(0x400, "IMEIValue");}}
/*  Original JS code in index.html  */
var a = document.cookie;
var b = bridge.send(a);
func1(b);
```

Listing 3. An example stub code of JavaScript bridge.

data that transmitted across languages. Therefore, BRIDGEINSPECTOR system rewrites the above functions to implement the logging of the tainted Java data transmitted through the bridge interface. The recorded information includes the bridge method name, the data types, values, and taint tags of Java data.

During the bridge method invocations, the invokeMethod function calls the javaValueToJvalue function to convert Java data to JS data and return it to JS environment when the target Java method finishes executing. Therefore, we obtain the bridge method name by capturing the 'method' parameter of the function. The information of data types, values, and taint tags are obtained by intercepting the 'value' parameter of the javaValueToJvalue function.

For example, when the app code shown in List 1 uses the JS bridge to get the IMEI number, the tainted Java data are recorded in logs using JavaScript Object Notation (JSON) format, shown in List 2.

*b) Java-to-JS taint mapping module:* In order to map the recorded Java taint tags to correct JS data, the Java-to-JS mapping module first performs a static analysis of the app to locate the callsite of the corresponding bridge method, and then restores the taint tags of the returned JS data at the callsite to the recorded tags.

We resort to the Apktool [2] to perform the static analysis of the app. We first use the Apktool to decompile the hybrid app and extract the JS code. Then we leverage regular expression matching to quickly find the bridge method callsites in the code. To improve precision, we skip the comments section and only perform string matching of method names in the code section. However, if the JS code dynamically constructs an invocation to the bridge method through string concatenation, our method may miss some callsites and fail to map the taint. Issuing such cases, we manually analyze and find the method invocation points in the code.

To facilitate the JSTA system to directly track the mapped JS data, the mapping module automatically completes the taint restoration by constructing stub code of bridge methods. The detailed process is as follows. The module first parses the Java taint log and traverses all log. If there is a callsite in the JS code that corresponds to the target bridge method in the taint record, it constructs a return statement according to the information of data type, value, and taint tags in the record. The statement will return a JS value that carries a Java taint tag. List 3 shows the stub code constructed from the taint log in List 2. Because the bridge method's name is unique in the JS code, we can assure that the above code construction will not map the tainted Java data into the JS data returned by other bridge methods, which might cause the imprecision of the taint mappings.

In this way, the JS code that merges the bridge stub carries taint tags from the Java layer. The JSTA system will treat the JS data with Java taint tags as a new source and further track them in the JS execution environment.

*2) JS-to-Java Taint Propagation:* Similarly, the taint propagation from JS to Java is also composed of 2 modules, the JS taint logging module and the JS-to-Java taint mapping module. The logging module is responsible for recording the JS data transmitted by the bridge methods, and the mapping module restores the taint tags of the corresponding Java data in the TaintDroid according to the recorded taint information.

*a) JS taint logging module:* The logging module is implemented in JSTA to record the associated information of JS data. In the code rewriting process of JSTA, we apply hook techniques to insert a log function inside the stub code to capture and record the method name and parameters' taint information of JS bridges. For the stub method 'bridge.send()' shown in List 3, List 4 shows the code after inserting the log hooks. The hooks will record the name of the 'bridge.send()' method and the cookie parameters passed to Java in JSON logs (shown in List 5).

*b) JS-to-Java taint mapping module:* In the data transfer process of the bridge interface, the JNI data conversion function named jvalueToJavaValue will convert the JS data into the corresponding Java data and pass them to the target Java

```
1  /*  Bridge stub code with log hooks.  */
2  bridge = {  send: function() {
3      //  Hook function for logging method name.
4      log(arguments.callee.name);
5      //  Hook function for logging method arguments.
6      log(arguments);
7      //  Return tainted Java data to JavaScript.
8      return new taintedJavaData(0x400, "IMEIValue");}}
9  /*  Original JS code in index.html  */
10 var a = document.cookie;
11 var b = bridge.send(a);
12 func1(b);
```

Listing 4. Stub code with log hooks.

```
1  {   "Method": "bridge.send",
2      "Arguments": [{"Type":"String", "Value":"cookieValue", "Tag":0x10000}]}
```

Listing 5. An example JavaScript taint record in JSON format.

method when the bridge method is executed. Thus, we add a hook function within the jvalueToJavaValue function in order to restore the recorded JS taint tags into the correct Java data before the app executes the Java method.

The hook function intercepts the argument Jvalue of the jvalueToJavaValue function to find the correct tainted data in the JS log according to the bridge method name and the parameters' information (e.g., data type, data value, taint tags) corresponding to the parameters. When the tainted data are found, the hook function modifies the taint tag of the converted Java data to the matched tags in the log. This completes the propagation of the JS tags into the Java environment.

Although the name of JS bridge method's implementation might have different name, we can use their detailed parameters' information to differentiate between the methods with the same name, which can assure the correctness of the matched JS data. Therefore, the mapping module will not cause diffusion and omission of taint propagation.

### *E. System Deployment and Run*

*1) Deployment:* In order to detect actual malicious behaviors and discover real vulnerabilities in hybrid apps, we deploy the Java and JS taint tracking components of BRIDGEINSPECTOR in Android mobiles and desktop PCs respectively. We root a mobile device and flash the customized TaintDroid images to it. Specifically, the mobile we use is Google Nexus 4, running the modified TaintDroid (based on Android 4.3 system), with 2G RAM and AMD CPU. The PC is a Dell OPTIPLEX 380 workstation with one Intel Pentium E5300 CPU and 4G memory, running the 32-bit Ubuntu 16.04 with kernel 3.8.0-35-generic.

The reason for the above deployment is that JSTA runs at the application-level while TaintDroid runs on the Android system-level. If deploy both JSTA and TaintDroid in Android mobiles, TaintDroid will treat JSTA as an application-level app and track it, which might prevent the BRIDGEINSPECTOR system from working properly.

*2) Run:* The detailed execution process of BRIDGEINSPECTOR consists of three phases. We detect the Native, Web/Native-to-Web and Web-to-Native security threats in the three phases, respectively. In the first phase (draw with solid arrows in Figure 9), BRIDGEINSPECTOR dynamically executes hybrid apps in Android smartphones. During the testing, we manually complete the app registration, GUI traversal and activity operations. For example, to test an app with functionalities of barcode scanning and voice chatting, we manually click on the relevant buttons to scan the barcode and send voice messages. When the dynamic execution is finished, the Java tracking component outputs the Native risks and the logging module outputs the Java taint logs. In the second phase (draw with dashed arrows in Figure 9), BRIDGEINSPECTOR first obtains the bridge stubs through the mapping module, and then starts the JS tracking component to analyze the JS code that merges the bridge stubs and outputs the Web/Native-to-Web security analysis result. At the same time, the mapping component outputs the JS taint logs. In the last phase (as shown by the dotted arrows in Figure 9), BRIDGEINSPECTOR runs the app on the smartphone for the second time. The Java tracking component tracks the Java code again based on the JS log generated in the second phase, and outputs the Web-to-Native security risk of the app. The second execution is also performed by using manual testing.

The reason we adopt such a three-phase analysis is that the cross-language data flow analysis of the tracking component relies on the result logs that produced by the mapping component. As a result, BRIDGEINSPECTOR generates Java logs in the first phase to support the second phase analysis for Java-to-JS data flows. When the second phase tracking generates JS logs, the third phase analysis for the JS-to-Java data flows can be successfully completed.

TABLE III
DATASETS OF EXPERIMENTS

| Dataset | Total samples | Categories | # of apps |
|---|---|---|---|
| BridgeBench | 38 | Cordova test cases | 38 |
| Specific Apps | 117 | Maps & Navigation | 15 |
| | | Weather / Pedometer | 40 / 18 |
| | | Photography | 16 |
| | | Contact / SMS Backup | 13 / 15 |
| Popular Apps | 1055 | Tools / Wallpaper | 139 / 93 |
| | | Social / Lifestyle | 194 / 164 |
| | | News / Office | 41 / 141 |
| | | Video & Audio | 140 |
| | | Photography | 69 |
| | | Travel / Shopping | 61 / 13 |

## V. EVALUATION

We conduct extensive experiments to evaluate the performance of BRIDGEINSPECTOR in terms of completeness, effectiveness and applicability. In this section, we first give the details of experiment setup, and then report the results of the experiments along with analyses and discussions.

### *A. Experiment Setups and Design*

The first set of experiments aim to verify whether our system can completely detect all kinds of insecure data flows crossing the bridge interface. To this end, we combine each type of sensitive data with different critical functions in Table I to generate totally 154 cross-language data paths. We then design a test benchmark called BRIDGEBENCH, which contains 38 sampling apps that cover all possible cross-language data paths. We use BRIDGEBENCH to test the completeness of our system.

In the second set of experiments, we test the effectiveness of BRIDGEINSPECTOR with 117 apps intentionally gathered from Baidu Android market across six categories, including maps, weather, pedometer, photography, and contact/SMS backup. These apps provide users with customized value-added and optimization services based on specific mobile data, such as navigation and weather forecasts (location-based services, LBS [38]), and motion measurement (mobile sensor-based services).

TABLE IV
COVERAGE TEST RESULT OF BRIDGEBENCH

| ID | Sources of data-flows | | Sinks of data-flows and detection results | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Types of Sources | Cordova plug-ins for sensitive data acquisition | $Sk_{web1}$ | $Sk_{web2}$ | $Sk_{web3}$ | $Sk_{web4}$ | $Sk_{web5}$ | $Sk_{nat1}$ | $Sk_{nat2}$ |
| 1 | NFC | *phonegap-nfc* | ● | ● | ● | ● | ● | - | - |
| 2 | Contacts | *cordova-plugin-contacts* | ● | ● | ● | ● | ● | - | - |
| 3 | Mic | *cordova-plugin-media-capture* | ● | ● | ● | ● | ● | - | - |
| 4 | PhoneNumber | *cordova-plugin-sim* | ● | ● | ● | ● | ● | - | - |
| 5 | Location (GPS) | *cordova-plugin-geolocation* | ● | ● | ● | ● | ● | - | - |
| 6 | Location (Net, Last) | *cordova-plugin-geolocation* | ● | ● | ● | ● | ● | - | - |
| 7 | Bluetooth | *cordova-plugin-bluetooth-serial* | ● | ● | ● | ● | ● | - | - |
| 8 | Camera | *cordova-plugin-camera* | ● | ● | ● | ● | ● | - | - |
| 9 | Accelerometer | *cordova-plugin-device-motion* | ● | ● | ● | ● | ● | - | - |
| 10 | SMS | *cordova-plugin-sms* | ● | ● | ● | ● | ● | - | - |
| 11 | Device | *cordova-plugin-sim* | ● | ● | ● | ● | ● | - | - |
| 12 | Wi-Fi | *com.pylonproducts.wifiwizard* | ● | ● | ● | ● | ● | - | - |
| 13 | ICCID | *cordova-plugin-sim* | ● | ● | ● | ● | ● | - | - |
| 14 | Barcode | *phonegap-plugin-barcodescanner* | ● | ● | ● | ● | ● | - | - |
| 15 | Account | *com.polychrom.cordova.AccountManagerPlugin* | ● | ● | ● | ● | ● | - | - |
| 16 | History | *customized plug-in* | ● | ● | ● | ● | ● | - | - |
| 17 | Doc.cookie | *customized plug-in* | - | - | - | - | - | ● | ● |
| 18 | Doc.domain | *customized plug-in* | - | - | - | - | - | ● | ● |
| 19 | Doc.lastModified | *customized plug-in* | - | - | - | - | - | ● | ● |
| 20 | Doc.referrer | *customized plug-in* | - | - | - | - | - | ● | ● |
| 21 | Doc.title | *customized plug-in* | - | - | - | - | - | ● | ● |
| 22 | Doc.URL | *customized plug-in* | - | - | - | - | - | ● | ● |
| 23 | Loc.hash | *customized plug-in* | - | - | - | - | - | ● | ● |
| 24 | Loc.host | *customized plug-in* | - | - | - | - | - | ● | ● |
| 25 | Loc.hostname | *customized plug-in* | - | - | - | - | - | ● | ● |
| 26 | Loc.href | *customized plug-in* | - | - | - | - | - | ● | ● |
| 27 | Loc.pathname | *customized plug-in* | - | - | - | - | - | ● | ● |
| 28 | Loc.port | *customized plug-in* | - | - | - | - | - | ● | ● |
| 29 | Loc.protocol | *customized plug-in* | - | - | - | - | - | ● | ● |
| 30 | Loc.search | *customized plug-in* | - | - | - | - | - | ● | ● |
| 31 | WebSocket message | *customized plug-in* | - | - | - | - | - | ● | ● |
| 32 | WebStorage data | *customized plug-in* | - | - | - | - | - | ● | ● |
| | | | $Sk_{web6}$ | $Sk_{web7}$ | $Sk_{web8}$ | $Sk_{web9}$ | $Sk_{web10}$ | $Sk_{web11}$ | $Sk_{web12}$ |
| 33 | NFC | *phonegap-nfc* | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 34 | Contacts | *cordova-plugin-contacts* | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 35 | Bluetooth | *cordova-plugin-bluetooth-serial* | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 36 | SMS | *cordova-plugin-sms* | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 37 | Wi-Fi | *com.pylonproducts.wifiwizard* | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 38 | Barcode | *phonegap-plugin-barcodescanner* | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

Doc: document object. Loc: location object. ●: Cross-language data leaks. ○: Cross-language XSS vulnerabilities.
$Sk_{web1\text{-}5}$: JS network communication APIs. $Sk_{web6\text{-}12}$: JS code execution APIs. $Sk_{nat1}$: Java network API. $Sk_{nat2}$: Java SMS API.

On the basis of the above experiments, we finally investigate the situation of cross-language security risks in the current Android market. In the process of data analysis and research, we evaluate 1055 top popular apps from Baidu across 10 categories, e.g., system tools, theme wallpaper and social communications. The detailed classification and numbers of tested apps are shown in Table III.

We test all the apps manually in the evaluation and spend enough time to execute each app to guarantee that all the possible related paths can be covered. Actually, although we develop some simple automatic testing code for this purpose, considering the complexity of hybrid apps, we use manual testing in our experiments to execute bridge methods, aiming to cover all possible data paths associated with the bridge communication.

The idea of using automatic testing technique can be greatly helpful in reducing manpower and improving the efficiency in large-scale app testing [4], [5], [18], [27]. However, the main barrier when we to cooperate automatic testing with our BRIDGETAINT tool is that the current automatic testing tools cannot analyze events of Web pages wrapped in WebView, which makes it is difficult to automatically trigger data paths related to bridge communication in hybrid apps by using these tools. Besides, the automatic techniques have certain limitations in that when the app requires users to input some specific content or SMS verification code, it is possible that the automatic execution will be hindered and thus the testing paths are not fully covered.

### B. Path Coverage of Unsafe Data-Flows

Cordova is currently the most popular open-source hybrid app development framework that provides powerful cross-platform capabilities, many other development frameworks (e.g., Ionic, MobileFirst, and Onsen UI) are built on top of Cordova. Thus, we choose to use Cordova to implement BRIDGEBENCH.

BRIDGEBENCH covers a total of 154 data paths from sources to sinks. The detection results of these data paths are listed in Table IV, in which the first column is the source of the data path that generates the sensitive data, and the second column is the detection result of the data path from the same sensitive data to different critical sinks. We use $Sk_{webi}$ and $Sk_{nati}$ to denote the $i$-th sinks in the web and the native environment, respectively. For each path, we use the related Cordova plug-in to read the corresponding sensitive data. For the data that cannot be directly obtained by system plug-ins, we use our own custom plug-ins to get them. We then deal with these data using the key functions listed in Table I. The first group of samples (No.1-32) sends user privacy through network sinks. The second group of samples (No.33-38) utilizes

TABLE V
DETECTION RESULT OF 117 SERVICE-SPECIFIC APPS

| | **Maps & Navigation** | | | **Weather** | | | **Photography** | | | **Contact Backup** | | | **SMS Backup** | | | **Pedometer** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | R# | N# | B# | R# | N# | B# | R# | N# | B# | R# | N# | B# | R# | N# | B# | R# | N# | B# |
| NFC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Contacts | 15 | $8^{*}$ ($1^{+}$) | 0 | 39 | $20^{*}$ ($3^{+}$) | $1^{*}$ | 16 | 0 | 0 | ***13*** | ***12*** | ***1*** | 15 | 3 | 2 | 18 | $9^{*}$ ($1^{+}$) | 0 |
| Mic | 15 | 3 | 0 | 16 | 2 | 0 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PhoneNumber | 15 | 3 | 0 | 39 | 20 | 4 | 16 | 0 | 0 | 13 | 7 | 0 | 15 | 3 | 2 | 18 | 10 | 1 |
| Location | ***15*** | ***14*** | ***1*** | ***40*** | ***32*** | ***8*** | 16 | 2 | 4 | 13 | 3 | 0 | 15 | 4 | 0 | 18 | 9 | 1 |
| Bluetooth | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 |
| Camera | 0 | 0 | 0 | 4 | 1 | 0 | ***16*** | ***6*** | ***10*** | 0 | 0 | 0 | 0 | 0 | 0 | 18 | 2 | 1 |
| Accelerometer | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ***18*** | ***16*** | ***2*** |
| SMS | 15 | 0 | 0 | 39 | $2^{*}$ ($2^{+}$) | 0 | 16 | 0 | 0 | 13 | 9 | 1 | ***15*** | ***13*** | ***2*** | 18 | $1^{*}$ ($1^{+}$) | 0 |
| IMEI | 15 | 14 | 1 | 40 | 37 | 1 | 16 | 6 | 10 | 13 | 12 | 0 | 15 | 5 | 2 | 18 | 16 | 2 |
| Wi-Fi | 15 | 3 | 0 | 39 | 30 | 4 | 16 | 1 | 0 | 13 | 7 | 2 | 15 | 3 | 0 | 18 | 9 | 0 |
| ICCID | 15 | 3 | 0 | 7 | 5 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 3 | 0 |
| Barcode | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Account | 15 | 0 | 1 | 39 | 19 | 0 | 16 | 0 | 0 | 13 | 7 | 0 | 15 | 3 | 0 | 18 | 9 | 0 |
| History | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

R: read data. N: send data through native Java. B: send data through JS bridge. *: detected by BRIDGEINSPECTOR. +: detected by Play Protect.

the code execution APIs to directly process the native data acquired by the plug-in. Once these data contain malicious code, it will cause cross-language code injection attacks. From the test results we can see that BRIDGEINSPECTOR has successfully detected all unsafe data flows of cross-language privacy leaks and code injection vulnerabilities in all paths.

### C. Effectiveness on Service-Specific Android Apps

The data tracking result of the second experiment is shown in Table V, in which R represents that the app reads the corresponding data, N and B denote that the app send the data to its server via Java and JS bridge code, respectively.

As can be seen from Table V, typical LBS apps, including the navigation and the weather apps, have acquired the user's real-time location. Among the data acquisitions in these apps, 46 of them are completed by using Java code, and nine of them is done by utilizing JS bridge code. Similarly, the camera and the photography apps providing share functions are also detected having the behavior of accessing and sending the album and user photos. The other three types of apps also have the same behaviors of specific data acquisitions, such as the user contact list and SMS content acquisition in cloud backup apps, acceleration sensor data acquisition by the pedometer apps. The above results are consistent with the actual functionalities provided by the specific apps, which objectively indicate that the data tracking of the BRIDGEINSPECTOR is realistic and accurate.

In addition, we find that the collected private data of some service-specific apps exceed their application requirements, which brings serious risks to app users. In the 117 tested applications, we find 38 of them (marked with * in Table V) are suspected of stealing privacy (such as contacts and SMS messages) and verify them manually. We search them in Google Play and find 19 of them (such as veryzhun.vzcom, com.sportq.fit, com.tp.android.bfleftme, com.hotbody.fitzero, com.lofter.android) are still present in Google Play. This shows that our approach can find suspected applications that pass the detection of Google Play.

### D. Research on Popular Market Apps

*1) Total Statistics and Analysis:* In the tested 1055 popular market apps of the third experiment, BRIDGEINSPECTOR detects that up to 87.5% of them have security risks. In these issues, potential privacy leaks are particularly prominent,

TABLE VI
TOTAL STATISTICS OF DATA TRACKING IN 1055 MARKET APPS

| **Data sources** | **Arrival Sinks** | | | |
|---|---|---|---|---|
| | $Sk_{nat1}$ | $Sk_{nat2}$ | $Sk_{web1\text{-}5}$ | $Sk_{web6\text{-}12}$ |
| NFC | 14.41% (152) | 0.09% (1) | 2.75% (29) | - |
| Contacts | 1.90% (20) | - | - | - |
| Mic | 23.41% (247) | - | - | - |
| PhoneNumber | 0.76% (8) | - | 1.33% (14) | - |
| Location | ***29.95% (316)*** | 0.09% (1) | 2.27% (24) | - |
| Bluetooth | 4.45% (47) | 0.09% (1) | - | - |
| Camera | 19.81% (209) | - | - | - |
| Accelerometer | 2.46% (26) | - | 0.85% (9) | - |
| SMS | 2.09% (22) | - | 2.94% (31) | - |
| Device | ***80.38% (848)*** | - | ***3.60% (38)*** | - |
| Wi-Fi | 1.80% (19) | - | 0.28% (3) | 0.19% (2) |
| ICCID | 2.09% (22) | - | 2.37% (25) | - |
| Barcode | 3.03% (32) | - | - | 0.28% (3) |
| Account | 0.47% (5) | - | 1.14% (12) | - |
| History | 0.09% (1) | - | 0.85% (9) | - |
| Doc.cookie | 0.57% (6) | - | - | - |
| Doc.domain | 1.80% (19) | - | - | - |
| Doc.lastModified | - | - | - | - |
| Doc.referrer | 0.09% (1) | - | - | - |
| Doc.title | 2.56% (27) | - | - | - |
| Doc.URL | 0.57% (6) | - | - | - |
| Loc.hash | - | - | - | - |
| Loc.host | 0.47% (5) | - | - | - |
| Loc.hostname | - | - | - | - |
| Loc.href | 1.23% (13) | - | - | - |
| Loc.pathname | 0.76% (8) | - | - | - |
| Loc.port | - | - | - | - |
| Loc.protocol | 0.19% (2) | - | - | - |
| Loc.search | 0.66% (7) | - | - | - |
| WebSocket | - | - | - | - |
| WebStorage | 0.38% (4) | - | - | - |

Doc: document object. Loc: location object. $Sk_{nat1}$: Java network API. $Sk_{nat2}$: Java SMS API. $Sk_{web1\text{-}5}$: JS network communication APIs. $Sk_{web6\text{-}12}$: JS code execution APIs.

as many as 87% of the apps transfer user sensitive data. Specifically, Java communication is the main way of data transmission, 84.5% (891) of the apps send user's mobile privacy by using Java code. In addition, bridge communication has become an important means of information disclosure, 4.5% (48) and 3.0% (32) of the apps obtain user phone data and online privacy through JS bridge code. The statistics of these security issues are shown in Table VI, in which the first column indicates different types of sensitive data and the second column represents the percentage of apps of security risks caused by the corresponding data. On the whole, the most frequently disclosed user privacy sending through the

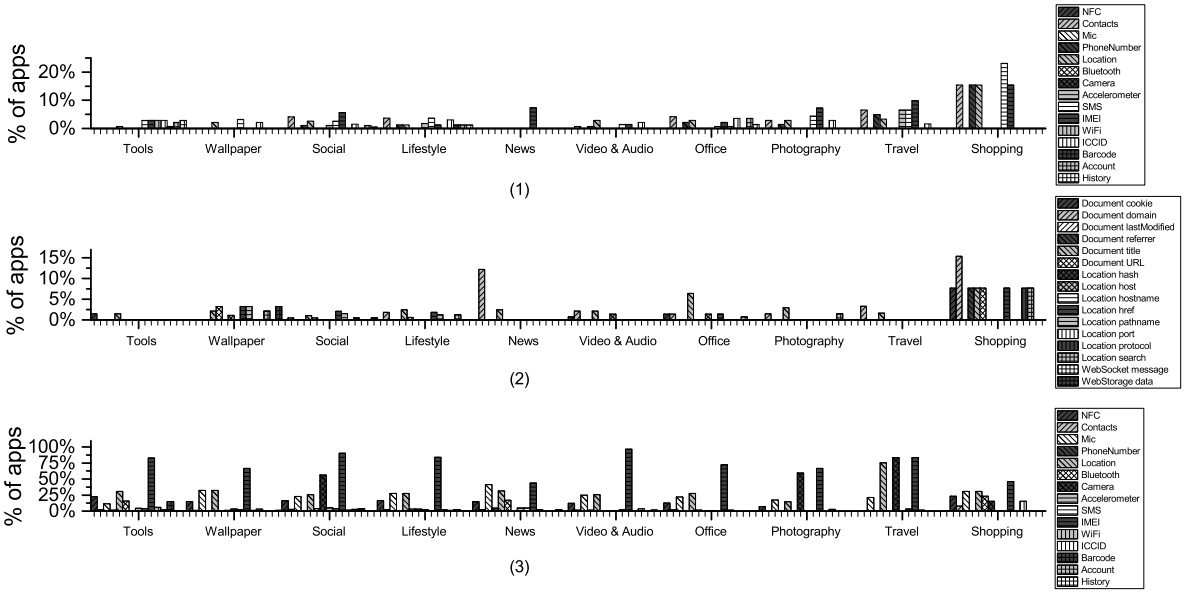


Fig. 10. Comparison of data transmission in 1055 popular market apps by app categories. (1) Native sensitive data transmitted through native-to-web communications. (2) Web sensitive data transmitted through web-to-native communications. (3) Native sensitive data transmitted through native Java communications.

native-to-web communication channels and the web-to-native communication channels are the IMEI numbers and the web page titles, respectively. For the Java communication channels, the transmitted data include the IMEI and the location data, as well as the NFC, the microphone data, the Bluetooth and the album information.

Besides, five apps are vulnerable to serious cross-language code injection attacks. We manually validate these vulnerable apps. First, we reverse engineer these apps by using the Apktool and find that some functions obtain untrusted data from the Wi-Fi and the barcode channels. These functions directly output these data into the DOM of Web page without any validation or filtering, resulting in XSS vulnerabilities. Then, we verify whether these vulnerabilities can be exploited by attackers. We manually construct a Wi-Fi site and barcode that contains crafted malicious code to attack these apps. When we use the apps to scan the malicious Wi-Fi and barcode, the vulnerabilities is triggered and the exploit codes are successfully executed.

*2) Detailed Data Transmission Analysis by App Categories:* Generally, the functionalities and services provided by apps depend on different sensitive data. In order to further understand whether their data is consistent with their requirements or not, and assess the impact of data transmission, we perform a statistical analysis on sensitive data transfers from the perspective of application categories and data types (as shown in Figure 10 and Figure 11, respectively).

*a) The comparison and cause analysis of data transmission in different apps:* From Figure 10 (1) and (3), we can see that, for the privacy stored on mobile devices, the potential data leak situation is most serious in social and lifestyle apps. Up to 9 kinds of user data is transmitted, such as IMEI and contact list. In addition, the situation in photography and travel apps are also significant, the exposed user privacy include albums, location and microphone data. Figure 10 (2) shows that the transmission of online privacy is also prominent. For instance, the wallpaper and the online-shopping apps collected 8 types of data, e.g., the titles, the cookies and other private data.

The main reason for the data transfers is from the following two aspects. First, part of them comes from the normal request of user data from app developers. Because in the app development process, developers usually need to obtain device ID (e.g., IMEI) to identify different users, and then send or read the SMS to verify and validate the legitimacy of the user identities, and finally provide the users with effective specific services (e.g., LBS). Therefore, the collection of various apps for IMEI, SMS, and other services related data is more common. Such as the accessing of user contact and album in social apps, is mainly used for address book matching and photo sharing. Second, the others are malicious data theft from attackers or ad servers. As can be seen from Figure 10 (1), some lifestyle apps have accessed contact list that are not relevant to their functionalities, showing potential malicious behaviors. In addition, a little office apps collect geolocation data to locate and track users in real-time, imposing serious risks.

*b) Risk analysis of different kinds of data transmission:* Figure 11 shows the transmission ratio of different types of sensitive data in market apps. In order to analyze the security risks of these data transfers, we classify them into two ways, user-aware data transfers and user-unaware data transfers. The first one is a form of active behavior which means that the users expose the privacy to the Internet absolutely by themselves; the latter is the user sensitive data are silently transmitted to remote servers of apps by using passive collection. For instance, NFC, album, microphone privacy, and the data which need user interactions to access are mostly leaked in the first way. The other data are exposed by the second way, such as location, contacts and other device privacy and web-side data.

Often, the security risks of user-aware information disclosure are generally small. For example, as can be seen from Figure 11 (1) and (3), the NFC data's transmission ratio of the system tools and the social apps is high. For the microphone

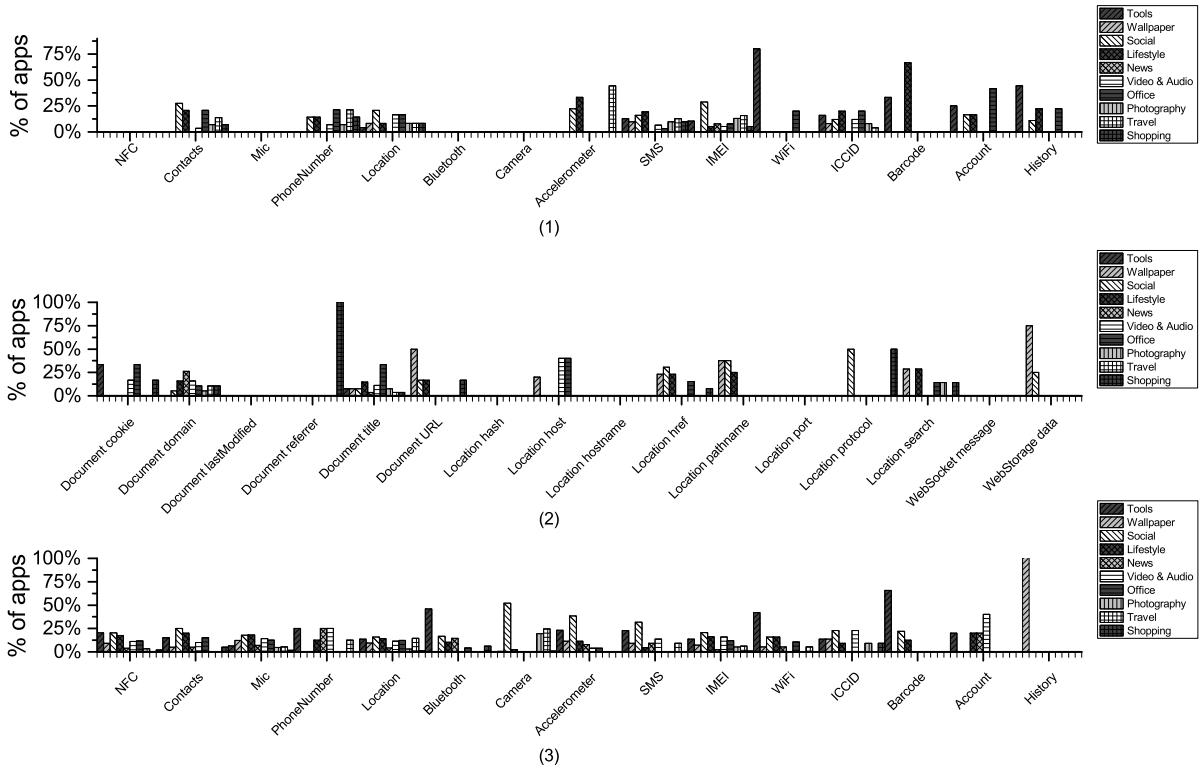


Fig. 11. Comparison of data transmission in 1055 popular market apps by sensitive data types. (1) Transmission ratio of native senstive data in different application categories by using native-to-web communications. (2) Transmission ratio of native senstive data in different application categories by using web-to-native communications. (3) Transmission ratio of native senstive data in different application categories by using native Java communications.

data, the social and the practical lifestyle apps have a high proportion of data transfers. Because these categories of apps provide rich functionalities of voice chatting, voice searching and NFC services, the users need to take the initiative to submit the relevant data to obtain efficient custom services, so the risk is relatively small.

Compare with the active data transfers, there is a serious risk for the passive data transfers. For example, the lifestyle apps collect acceleration data of user devices, the social apps leak user locations, and office apps access and transmit the Android account information of users. These data can be used for social engineering even if they cannot directly bring economic or intellectual values to attackers. The attacker may obtain users' login account name and location information, and further retrieve users' other sensitive data to perform more sophisticated social engineering attacks. In addition, the bridge interface is also a potential risk in hybrid apps. Through the bridge communication, the native code can obtain the web private data, including page titles, cookie, and users searching keywords. These data often contain users' browsing history and online transactions and other secret information (e.g., trading accounts, transaction passwords and transaction amount), once transmit to third-parties over network, bringing users a huge threat.

## VI. DISCUSSION

### A. Comparison With HybriDroid

HybriDroid is the most similar work to our paper, uses static taint analysis to detect Java-to-JS leaks. The cross-language taint mapping mechanism used in HybriDroid is based on the execution semantics of bridge communication, which is obtained by inspecting Android source code. The execution semantics are then used in the static analysis of the app code to taint data and find potential leaks. This mapping mechanism cannot analyze apps developed using frameworks. In contrast, BRIDGETAINT uses a dynamic method to record and map the data on the bidirectional transmission path by adding hook points in the underlying JNI implementation of WebView. Our method can record the name of objects in the bi-directional cross-language communication during execution and associate the objects, and thus can support analyzing apps developed using frameworks.

Below we use two examples to illustrate the contribution of our work over existing works like HybriDroid. In both examples the app reads a user's SMS message into a web page. In the first example (shown in Figure 12) this functionality is implemented by the developer's own code, and in the second example (shown in Figure 13) this functionality is implemented by using the Cordova framework.

1) In the first example, the app developer directly declares the Java object and its exposed JS bridge object by using the addJavaScriptInterface API of WebView. In this case, both BRIDGETAINT and HybriDroid can associate the cross-language data, but they use different methods.

HybriDroid directly obtains the JS object's corresponding Java object through code analysis, and correlates object methods and their method parameters. For example, as shown in Figure 12, HybriDroid maps the JS object 'SMS' to the Java object 'SMSObject' according to the statement 'w.addJavaScriptInterface(new SMSObject(), "SMS")', and thus it can associate SMSObject.getSMS() with SMS.getSMS(). The parameters of the two methods are the data transmitted through the bridge communication.

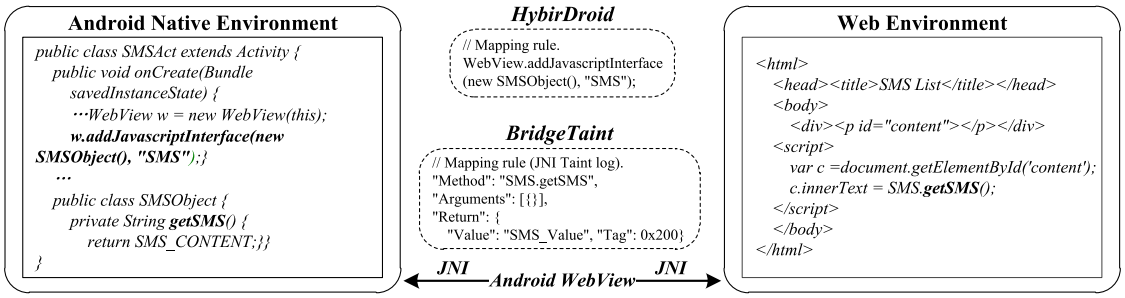


Fig. 12. Comparison of mapping mechanisms between BRIDGETAINT and HYBRIDROID (normal app).

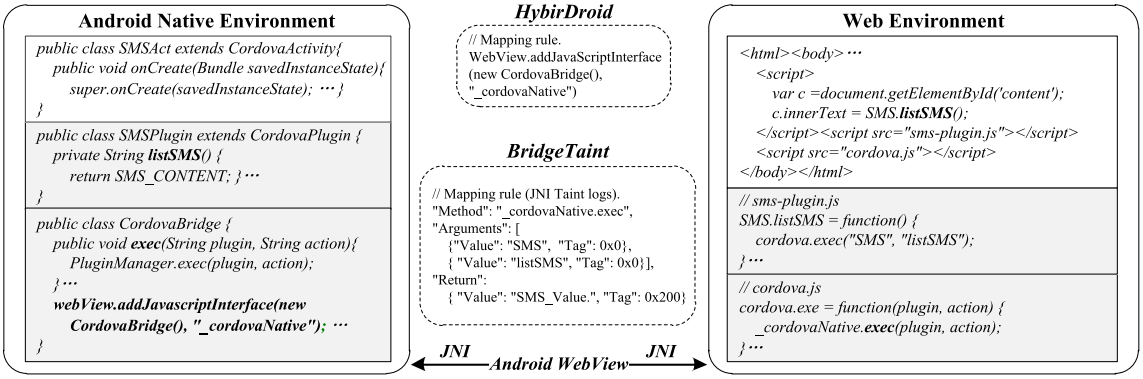


Fig. 13. Comparison of mapping mechanisms between BRIDGETAINT and HYBRIDROID (cordova-based app).

HybriDroid applies this mapping strategy to static analysis to implement Java-to-JS taint tracking.

In contrast, BRIDGETAINT uses a dynamic method to record and map the data on the transmission path, which is different from the one used in HybriDroid. To do this, BRIDGETAINT adds hook points in the underlying JNI implementation of WebView in order to record the actual bridge methods, the method parameters, and the method return values in the bidirectional channels. In this example, the transmitted bridge method name 'SMS.getSMS' and the returned value are recorded. BRIDGETAINT uses these recorded information to associate the SMS.getSMS() method with the SMSObject.getSMS() method.

2) In the second example, the app is developed by using the Cordova framework and thus its cross-language data transmissions are more complex. In this case, BRIDGETAINT can still match the data association, while HybriDroid cannot.

Cordova provides some add-on libraries called plug-ins, which offers a set of JS interfaces to invoke native device capabilities. The developer can directly use these JS interfaces to call native components without writing Java code. In the example shown in Figure 13, the function of displaying SMS messages on the page is implemented by utilizing Cordova's SMS plug-in. By simply importing the related plug-in library, the developer can use the JS method SMS.listSMS() to obtain the SMS content without writing his own code. The code marked in dark gray in Figure 13 is implemented in the Cordova library.

We first analyze why HybriDroid cannot match data association in this case. To ensure scalability, all the Cordova plug-ins are implemented on top of the WebView interface. They use the _cordovaNative.exe() method to pass the JS plug-in name and plug-in action as parameters to the exec() method of the CordovaBridge class at the Java side. The target plug-in methods are invoked by using the reflection mechanism provided in the exec() method of the PluginManager class. Figure 13 lists the core functions of the Cordova library that are actually invoked when SMS.listSMS() is executed. SMS.listSMS() uses the cordova.exec(plugin, action) method to pass the calling plugin name 'SMS' and the action 'listSMS' to the variables plugin and action in the bridge interface method _cordovaNative.exec(), which is then passed to Java via WebView. In the CordovaBridge class and the pluginManager class, the target listSMS() method of the SMSPlugin class is invoked via Java reflection.

From the above process we can see that the Cordova plug-in names and action names transmitted across languages can be determined only during the code execution. Direct static analysis on the arguments of the addJavaScriptInterface API cannot associate the cross-language data, because the actual arguments of WebView.addJavaScriptInterface() are fixed to 'CordovaBridge' and '_cordovaNative' in Cordova apps. Therefore, HybriDroid is unable to detect Cordova-based hybrid apps.

In contrast, our method can record the value and type of parameters passed by the bridge method during dynamic execution by hooking the underlying implementation of WebView. In the same example, 'SMS' and 'listSMS' are recorded, and thus the methods in JS and Java can be matched to associate the returned value of the corresponding method with the parameters.

### B. Comparison With Play Protect

We compare BRIDGEINSPECTOR with Play Protect, which is Google's latest comprehensive security service for Android. We install the Google Play Store App on a test phone (Nexus 5). The app provides two ways to perform Play Protect to detect malicious behaviors. One way is to run safety check

at the time of app installation. It requires setting up the app and activating its Play Protect protection mechanism. After setting up, when the mobile phone installs a new app, Play Protect will automatically scan the app and perform security detection. When there is a security risk in the newly installed app, Play Protect will prompt and alert user of data theft risks and mark it as a harmful application in the app installation stage. Another way is to scan the app after installation, which can detect all installed apps in the mobile phone. We perform the two kinds of security tests on app, and find that the test results are consistent.

For ease of comparison, we use Play Protect to detect the 117 testing apps listed in Table V. The results show that Play Protect find only 5 apps (marked with + in Table V) as privacy theft but it does not indicate what data the apps steal. In contrast, BRIDGEINSPECTOR finds all 38 suspected applications, including the 5 found by Play Protect. Besides, we also use Play Protect to scan all test cases of BridgeBench and find that Google Play cannot detect any security issues. Therefore, BRIDGEINSPECTOR can detect data paths of potential privacy theft more accurately and provides more details of leaked data than Play Protect.

## VII. RELATED WORK

Android security researches has been a hot topic in the field of mobile security, and granted many attentions by the members of both academic and industry community. Tan *et al.* [32] conducted a comprehensive survey on vulnerabilities, security mechanisms and detection systems of Android, and addressed the limitations of existing works and current challenges. Reaves *et al.* [30] performed a systematical technical analysis and comparison on static and dynamic analysis tools of Android native apps. In this paper, we mainly analyze and summarize the present researches and the existing tools of taint analysis for native apps and hybrid apps.

Researchers have implemented numerous analysis systems to detect the security issues in native Android apps over the recent 5 years. Various tools, e.g., AppsPlayground [29], TaintDroid [13], DroidScope [40], and VetDroid [42], use dynamic taint tracking techniques to detect the data leaks in Java code of apps. Among these tools, DroidScope even extend the analysis scope to native C code. In addition, static taint analysis is also a common method for discovering privacy exposure and security vulnerabilities. The typical data tracking systems include FlowDroid [6], AmanDroid [35] and the tools in the literature [9], [14], [16], [21], [24], [37], [41], as well as intent injection [25] and permission escalation [12] detection system.

However, the above-mentioned tools can only analyze the security problems in the native apps, and cannot apply for detecting the cross-language security threats caused by the interoperation between JS and Java code. Therefore, researchers have recently begun to explore new solutions for hybrid mobile apps, including taint tracking mechanisms and analysis methods for bridge calls in multiple programming languages.

Jin *et al.* [19] was the first to propose a static taint tracking system for hybrid apps. They found that HTML5-based hybrid apps developed using Cordova, are vulnerable to code injection attacks. Attackers can inject malicious code into Web pages via native internal and external channels, such as Bluetooth and Wi-Fi. They proposed a method to manually generate JS stub code to model the semantic of Cordova plug-ins, and use static taint analysis to detect whether the JS code in hybrid apps are prone to the attacks or not. Compared with their works, we present a complete cross-language taint tracking method having the following characteristics. First, in the process of native-to-web taint tracking, we propose an semantic modeling method, which construct stub code of bridge method entirely automatically according to the taint records. Moreover, the cross-language data tracking method proposed in this paper is not limited to the Cordova plug-ins, and our system can also detect the cross-language privacy leaks.

HybriDroid [22] focuses on static taint analysis for cross-language data leaks. It first constructs the control flow graph of the Web code and Java code, then builds a taint mapping between Java and JS data based on the bridge semantics, and finally performs static taint analysis on the graph to detect cross-language privacy leaks. However, it focuses on detecting Java-to-JS privacy leaks and cannot track JS-to-Java data flows. Moreover, it cannot be used to analyze hybrid apps developed with frameworks like Cordova. In contrast, our work can track data flows in bi-directions and can analyze apps developed with frameworks.

Additionally, the researchers applied the access control, static analysis, and symbolic execution in security detection of hybrid apps. Recent works [15], [20], [28], [31], [33] proposed a fine-grained access control strategy to isolate untrusted Web code in hybrid apps, thereby protect apps from code injection attacks. Hassanshahi *et al.* [17] found that hybrid apps are vulnerable to W2AI (Web-to-Application Injection) attacks, which means native Java code is vulnerable to untrusted Web code, and proposed a hybrid detection method combining dynamic and static analysis and symbol execution. The literature [8], [26], [36] used machine learning, behavior state machines, and data flow analysis to detect code injection in hybrid apps. Chen *et al.* [10] found a new channel for code injection of HTML5-based hybrid apps, and the attacker can exploit HTML5 text box to inject malicious code into apps.

## VIII. CONCLUSION

In this paper, a feasible bi-directional taint tracking method is proposed for addressing the security issues introduced by the bridge communication in Android hybrid apps. Based on the method, we develop a benchmark for bridge security test and implement a taint tracking system. We test the system with our benchmarks and real apps in Android market. As the result implicate, our method is effective to track data flows in the bridge communication and detect real vulnerabilities in hybrid apps.

## ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments to improve our paper.

## REFERENCES

[1] *Smartphones Industry: Statistics Facts*. Accessed: Jul. 5, 2017. [Online]. Available: https://www.statista.com/topics/840/smartphones/

[2] *Android APKTool: A Tool for Reverse Engineering Android APK Files*. Accessed: May 8, 2017. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[3] *Developer Survey: Insights, Trends, and Perspectives From the Worldwide Ionic Community*. Accessed: Nov. 28, 2017. [Online]. Available: https://ionicframework.com/survey/2017

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. ASE*, Sep. 2012, pp. 258–261.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015.
[6] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[7] J. Bai, W. Wang, M. Lu, H. Wang, and J. Wang, "TD-WS: A threat detection tool of WebSocket and Web storage in HTML5 websites," *Secur. Commun. Netw.*, vol. 9, no. 18, pp. 5432–5443, 2016.
[8] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation," in *Proc. ESSoS*, 2016, pp. 72–88.
[9] X. Chen and S. Zhu, "DroidJust: Automated functionality-aware privacy leakage analysis for Android applications," in *Proc. WiSec*, 2015, p. 5.
[10] Y.-L. Chen, H.-M. Lee, A. B. Jeng, and T.-E. Wei, "DroidCIA: A novel detection method of code injection attacks on HTML5-based mobile apps," in *Proc. Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 1014–1021.
[11] A. Continella *et al.*, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proc. NDSS*, 2017, pp. 1–16.
[12] X. Cui, J. Wang, L. C. K. Hui, Z. Xie, T. Zeng, and S. M. Yiu, "WeChecker: Efficient and precise detection of privilege escalation vulnerabilities in Android apps," in *Proc. WiSec*, 2015, p. 25.
[13] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2010.
[14] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. FSE*, 2014, pp. 576–587.
[15] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks," in *Proc. NDSS*, 2014, p. 1.
[16] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in droidsafe," in *Proc. NDSS*, 2015, pp. 1–16.
[17] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang, "Web-to-application injection attacks on Android: Characterization and detection," in *Proc. ESORICS*, 2015, pp. 577–598.
[18] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *Proc. AST*, 2011, pp. 77–83.
[19] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *Proc. CCS*, 2014, pp. 66–77.
[20] X. Jin, L. Wang, T. Luo, and W. Du, "Fine-grained access control for HTML5-based mobile applications in Android," in *Proc. ISC*, 2013, pp. 309–318.
[21] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proc. SOAP*, 2014, pp. 1–6.
[22] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for Android hybrid applications," in *Proc. ASE*, 2016, pp. 250–261.
[23] L. Li *et al.*, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. ICSE*, May 2015, pp. 280–291.
[24] K. Lu *et al.*, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. NDSS*, 2015, pp. 1–15.
[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. CCS*, 2012, pp. 229–240.
[26] J. Mao, R. Wang, Y. Chen, and Y. Jia, "Detecting injected behaviors in HTML5-based Android applications," *J. High Speed Netw.*, vol. 22, no. 1, pp. 15–34, 2016.
[27] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. ISSTA*, 2016, pp. 94–105.
[28] S. Pooryousef and M. Amini, "Fine-grained access control for hybrid mobile applications in Android using restricted paths," in *Proc. ISCISC*, Sep. 2016, pp. 85–90.
[29] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. CODASPY*, 2013, pp. 209–220.
[30] B. Reaves *et al.*, "*droid: Assessment and evaluation of Android application analysis tools," *ACM Comput. Surv.*, vol. 49, no. 3, 2016, Art. no. 55.
[31] M. Shehab and A. AlJarrah, "Reducing attack surface on cordova-based hybrid mobile apps," in *Proc. MobileDeLi*, 2014, pp. 1–8.
[32] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing Android: A survey, taxonomy, and challenges," *ACM Comput. Surv.*, vol. 47, no. 4, 2015, Art. no. 58.
[33] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for Web code on Android," in *Proc. CCS*, 2016, pp. 104–115.
[34] W. Wang, J. Bai, Y. Zhang, and J. Wang, "Dynamic taint tracking in javascript using revised code," *J. Tsinghua Univ. (Sci. Technol.)*, vol. 56, no. 9, pp. 956–962 and 968, 2016.
[35] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. CCS*, 2014, pp. 1329–1341.
[36] X. Xiao, R. Yan, R. Ye, Q. Li, S. Peng, and Y. Jiang, "Detection and prevention of code injection attacks on HTML5-based apps," in *Proc. CBD*, Oct./Nov. 2015, pp. 254–261.
[37] X. Xiao, N. Tillmann, M. Fahndrich, J. de Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," in *Proc. ASE*, Sep. 2012, pp. 80–89.
[38] Y. Xiao, S. Zhang, J. Wang, and C. Zhu, "A novel indoor localization algorithm for efficient mobility management in wireless networks," *Wireless Commun. Mobile Comput.*, vol. 2018, Jun. 2018, Art. no. 9517942.
[39] M. Xu *et al.*, "Toward engineering a secure Android ecosystem: A survey of existing techniques," *ACM Comput. Surv.*, vol. 49, no. 2, 2016, Art. no. 38.
[40] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Secur.*, 2012, pp. 569–584.
[41] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. CCS*, 2013, pp. 1043–1054.
[42] Y. Zhang *et al.*, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. CCS*, 2013, pp. 611–622.

Authors' photographs and biographies not available at the time of publication.