Precisely and Scalably Vetting JavaScript Bridge In Android Hybrid Apps

Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu

SUCCESS LAB
Texas A&M University
{ygl,abmendoza}@tamu.edu, {jialong,guofei}@cse.tamu.edu

Abstract. In this paper, we propose a novel system, named BridgeScope, for precise and scalable vetting of JavaScript Bridge security issues in Android hybrid apps. BridgeScope is flexible and can be leveraged to analyze a diverse set of WebView implementations, such as Android's default WebView, and Mozilla's Rhino-based WebView. Furthermore, BridgeScope can automatically generate test exploit code to further confirm any discovered JavaScript Bridge vulnerability.
We evaluated BridgeScope to demonstrate that it is precise and effective in finding JavaScript Bridge vulnerabilities. On average, it can vet an app within seven seconds with a low false positive rate. A large scale evaluation identified hundreds of potentially vulnerable real-world popular apps that could lead to critical exploitation. Furthermore, we also demonstrate that BridgeScope can discover malicious functionalities that leverage JavaScript Bridge in real-world malicious apps, even when the associated malicious severs were unavailable.

Keywords: Android Security, WebView Security, JavaScript Bridge

1 Introduction

Android apps (i.e., hybrid apps) increasingly integrate the embedded web browser component, "WebView", to render web pages and run JavaScript code within the app for seamless user experience. App developers can select from a variety of WebView implementations, such as Android's default WebView[1], Mozilla's rhino-based WebView[2], Intel's XWalkView[3], and Chromeview[4].

The power of WebView extends beyond the basic browser-like functionality by enabling rich interactions between web (e.g., JavaScript) and native (e.g., Java for Android) code within an app through a special interface known as a *"JavaScript Bridge"* [8, 14, 22, 23, 26, 27, 31, 32]. The JavaScript Bridge feature eases the development of hybrid apps. However, it also introduces critical security risks, such as sensitive information leakage, and local resource access (Section

[1] https://developer.android.com/reference/android/webkit/WebView.html
[2] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[3] https://crosswalk-project.org/
[4] https://github.com/pwnall/chromeview

2.2). Recent research work [8, 14, 22, 23] has highlighted the problems rooted in the use of JavaScript Bridge. However, an automated and fine-grained solution that can precisely and scalably detect JavaScript Bridge security issues is still missing.

In this paper, we present a precise and scalable static detection framework named *"BridgeScope"*. BridgeScope can automatically vet JavaScript Bridge usage in Android hybrid apps, and generate test exploit code to validate problematic JavaScript Bridge usage. Our approach is four-fold. First, BridgeScope fills the semantic gap between different core WebView implementations using a generalized WebView model. Second, using the generalized code, BridgeScope is able to precisely discover all available WebView components and bridges in an app. Third, BridgeScope reconstructs the semantic information of all JavaScript Bridges and identifies the *sensitive bridges* that contain data flows to sensitive API invocations (such as *getLastLoction()*). Finally, BridgeScope generates test exploit code using the analysis results (such as the UI event sequences to trigger WebView components and data flow inside sensitive bridges).

To achieve high precision and scalability, BridgeScope applies fine-grained type, taint, and value analysis, which is implemented based using a novel *"shadowbox"* data structure. We refer to our analysis technique as "shadowbox analysis". Compared with state-of-the-art static approaches such as data flow tracking [4,33], shadowbox analysis is path- and value-sensitive, while preserving precision and scalability. We evaluated our *shadowbox* analysis technique using a generic benchmark (DroidBench[5]), and found that it achieved 94% precision.

Finally, we evaluated BridgeScope with 13,000 of the most popular free Android apps, gathered from Google Play across 26 categories. BridgeScope found a total of 913 potentially vulnerable apps that may enable various types of attacks such as stealing sensitive information, gaining privileged access by bypassing security checks (such as Same Origin Policy[6] in the web context), and other serious attacks that may result in monetary loss to device users. Furthermore, our evaluation on real-world malware apps also demonstrated that BridgeScope could identify malicious functionalities hidden in sensitive JavaScript Bridges, even when the associated malicious servers were unavailable.

In summary, we highlight our key contributions:

– We conduct a systematic study on how WebView and JavaScript Bridge are used by both benign apps and malware with diverse WebView implementations.
– We design a precise and scalable static detection system to automatically detect vulnerabilities caused by JavaScript Bridge.
– We evaluate our detection system BridgeScope with real-world popular apps and find 913 potentially vulnerable apps that could be exploited by attackers. On average, our system can vet an app within 7 seconds with a low false positive rate.

[5] https://github.com/secure-software-engineering/DroidBench
[6] https://en.wikipedia.org/wiki/Same-origin_policy

2 Problem Statement

2.1 Background: WebView and JavaScript Bridge

To understand the fundamental components of WebView, irrespective of any specific implementation, we devise a model, shown in Figure 1, based on Android's default WebView which we find to be representative of most key properties that are important for our JavaScript Bridge analysis.

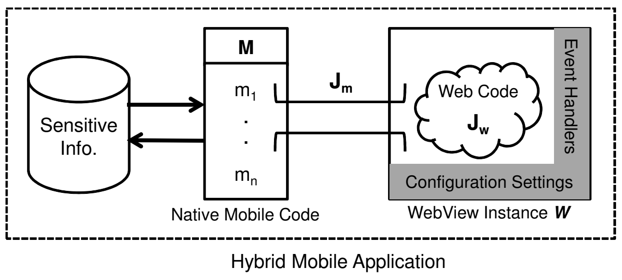

Fig. 1. Major modules in Android default WebView. In the example, Bridge J_m enables interaction between web code J_w and native code M.

JavaScript Bridge. The bridge J_m, shown in Fig. 1, allows interactions between the embedded web content J_w and the mobile native code implemented in M (the Bridge Object). Through its access to M, the web code in J_w inherits access to the local resources and sensitive information in the mobile framework.

To enable bridges in WebView, all bridges must be registered by the API *addJavascriptInterface(BridgeObject, BridgeName)* in advance, where *BridgeObject* is a native object (i.e., an instance of a Java class such as M in Fig. 1) that is being imported into the WebView instance W, and *BridgeName* is the object's reference name that can be used to directly access *BridgeObject* in the web context through J_w.

Annotation. In Android 4.2+, an annotation mechanism is introduced to restrict bridge access. In *BridgeObject*, only the methods that are explicitly annotated by '@JavaScriptInterface' can be invoked by JavaScript code.

Configuration Settings. Developers can configure a WebView component through its setting property. For instance, developers can enable/disable JavaScript in WebView. JavaScript is generally disabled by default requiring explicit activation by developers.

Event Handler. This mechanism allows developers to handle different events after WebView runs, which can be further utilized to provide additional security checks. For instance, the two event handlers *shouldOverrideUrlLoading*() and *shouldInterceptRequest*(), which are designed to handle URL and resources loading events, can be further used to restrict new web content loaded in WebView.

Same Origin Policy (SOP). In WebView, SOP is enabled to enforce access control on local data in the web layer between mutually distrusting parties.

However, SOP is not extended to resources in the native layer, such as users' contact list.

2.2 Security Issues Caused By JavaScript Bridge And Their Impacts

To illustrate the general problem with JavaScript Bridges, consider an Android app that exposes several methods $\{m_1...m_n\} \in M$ through a bridge J_m in an embedded WebView W, as shown in Figure 1. Consider that m_1 provides privileged access to sensitive APIs and/or functionality in the mobile framework. The web platform dictates that any code J_w that executes in the context of the embedded WebView W will also have access to the exposed interface J_m since $J_m \in J_w$. In other words, all JavaScript code J_w executed in the context of the WebView, even in embedded iFrames, can invoke all methods exposed by the app in M.

We consider two general approaches attackers may use to exploit JavaScript Bridge's:

- *Direct Access To Sensitive APIs*: Attackers who can inject code into W can then directly invoke sensitive functionality exposed through J_m. Attackers can also combine the use of multiple methods in M for more stealthy attacks that may use one method to read data, and another method to write data to a remote server. This is a variant of the classic confused deputy access control problem [16]. In this scenario, the WebView W, as the deputy, will diligently allow access to both exposed methods m_1 and m_2, allowing an attacker to first invoke the request for sensitive information through m_1, and then append the returned data to another request to the communication-enabled exposed interface m_2. Additionally, even if M does not include a method such as m_2, if the app has INTERNET permissions, then data from m_1 can still be leaked by J_w through a JavaScript HTTP method.
- *Cross-Origin DOM Manipulation*: A more interesting attack scenario emerges when m_n exposes an API that allows manipulation of the DOM in W, such as using $loadURL()$ or $loadDataWithBaseURL()$. As a result, an embedded iFrame in W can inject cross origin JavaScript code to effectively circumvent the same origin policy (SOP) and execute cross-site-scripting-like attacks in W's web origin. This is a violation of the same origin policy assumption, and can result in client-side XSS attacks using JavaScript Bridges. The root cause here is that the origin information is lost when JavaScript causes content to be loaded via a Bridge Object.

2.3 Sensitive APIs

We consider three type of *'sensitive'* system APIs, which we categorize as *source* (i.e., reading data from Android), *sink* (i.e., sending data out of mobile devices), and *danger* (i.e., dangerous operations) APIs. Specifically, we define "source API" and "sink API" using a comprehensive categorization developed in a previous work [25]. Additionally, we treat any API that can access local hardware (such as camera), and cause charges on the user's device (e.g. SMS, phone calls), as a "danger API".

2.4 Threat Model

We focus on hybrid apps that enable JavaScript and *JavaScript Bridge*. We assume that the code written in C/C++ and implicit data flow inside apps have minimal influence for our analysis. Generally, we consider attack scenarios in the context of benign and malicious apps:
Benign Apps. In this scenario, we assume that HTML/Javascript code loaded in WebView of benign apps is untrusted. We also assume that web attackers cannot directly access the native context, but can inject malicious HTML/JavaScript code to WebView through code injection attacks. We consider two ways for attackers to launch such attacks. Attackers can either compromise third-party websites, or inject/hijack network traffic (e.g., MITM attack) [3], such as the HTTP communication within WebView or third party Java libraries (e.g., ad libs [26]).

A much stronger assumption is that attackers may also hijack HTTPS traffic. Although this type of attack is difficult, it is still feasible, particularly considering how poorly/insecurely HTTPS is implemented/used in mobile apps [11, 13].
Malicious Apps. We assume that an attacker writes a malicious app using WebView and JavaScript Bridge, and submits it to app marketplaces, such as Android official market 'Google Play'. To evade security vetting systems in app marketplaces, such as Google Bouncer[7], the app is designed in such a way that 1) WebView loads a remote web page, whose content is controlled by the attacker; 2) the malware's sensitive behaviors are conducted in JavaScript Bridge, while its command & control (CC) logic is implemented by JavaScript code in WebView; 3) initially, the CC code is not injected into the loaded web page, and it only becomes available at a specific time, such as after the app bypasses the security checks and is published.

3 Shadowbox Analysis

In this section, we present details about our shadowbox analysis technique. First, we highlight the advantages of our approach, compared with other state-of-the-art approaches. Then, we present definitions and concepts related to shadowbox. We also discuss more details about type, taint and value analysis respectively. Finally, we show how to apply shadowbox analysis to solve different challenges, such as the problem caused by common data structures.

3.1 Challenges

Type, taint, and value/string analysis are frequently used program analysis techniques [4, 7, 15, 33]. However, the state-of-the-art approaches fall short of 1) precisely handling common data structures, such as list, hashmap, Android

[7] http://googlemobile.blogspot.com/2012/02/android-and-security.html

Bundle[8], Parcel[9], etc.; 2) maintaining path- and value-sensitivity while also remaining precise and scalable. These shortcomings may cause false negatives and false positives in analysis results.

Path- And Value-Sensitivity. To achieve high precision, it is critical to maintain path- and value-sensitivity. However, state-of-the-art work (such as Flowdroid [4] and Amandroid [33]) do not thoroughly maintain these properties. For instance, Listing 1.1 shows a snippet of a test case (from DroidBench) designed to test false positives of alias analysis. In this test case, sensitive information saved in '*deviceId*' is transferred to a field of an instance of the class '*A*' (Line 14), and then a sink API is called (Line 15), which merely sends out a constant string rather than the sensitive information. However, existing approaches, such as Flowdroid [4] and Amandroid [33], erroneously find a path from source to sink in this scenario due to path-insensitivity.

```
1  class A{ public String b = ”Y”;}
2  class B{ public A attr;}
3  ...
4  A b, q, y; B a, p, x;
5  a = new B(); p = new B();
6  b = new A(); q = new A();
7  if (Math.random() < 0.5) {x = a; y = b;}
8  else {x = p; y = q;}
9  x.attr = y;
10 q.b = deviceId; // source
11 sms.sendTextMessage(”+49 1234”, null, a.attr.b, null, null);//sink
```

Listing 1.1. A snippet of a test case for alias analysis in DroidBench

Common Data Structures. When a common data structure (e.g., list, hash map) is temporarily used to store tainted data (e.g., sensitive information), it may raise challenges to precisely track the sensitive data flow inside these data structures, since the position of taint data is difficult to determine. Most existing work (e.g., [4]) simply overtaints the entire data structure, which inevitably introduced false positives. Consider, for example, an array where only a single entry should be tainted (code shown in Listing 1.2). When line 4 is executed, only array[1] should be tainted. If, instead, the entire array is tainted, false positives are inevitably caused.

```
1 ArrayList<String> array = new ArrayList<String>();
2 String s = source();
3 array.add(s);              // array: [souce]
4 array.add(0, ”element0”);  // array: ["element0", source()]
```

Listing 1.2. An Example abstracted from real apps

BridgeScope solves this problem by performing fine-grained type, taint and value analysis using a '*shadowbox*' data structure as discussed in the following sections.

3.2 Concepts Related to Shadowbox

We define a *shadowbox* as the representation of an object (e.g. WebView). Generally, only tainted 'primitive variables' (e.g., integers), whose data type is primitive,

[8] https://developer.android.com/reference/android/os/Bundle.html

[9] https://developer.android.com/reference/android/os/Parcel.html

and all 'non-primitive variables' (e.g., string and array) are boxed. The relevant concepts are defined as follows: (note that v and s represent a variable and a shadowbox, respectively)

- ***A variable** v**'s representation*** $\langle scope_v, name_v \rangle$ *:* Generally, $scope_v$ is the full name of a function (for local variables), an object (for regular fields), or a class name (for static fields), while $name_v$ is v's name, which is usually a register name.
  Furthermore, to support inter-component communication (ICC) [33], the global but temporary representation `<global, intent>` is created to represent an intent message. To record a function f's return value, the representation `<f, return>` is used.
- ***Points-to relationship*** *:* If a variable v points to an object o, whose shadowbox is s, v and o have points-to relationship, which is represented by $v \rightarrow s$.
- ***Alias relationship*** *:* If two variables v_1 and v_2, and their shadowboxes s_1 and s_2 stasify the following statement: $v_1 \rightarrow s_1 \wedge v_2 \rightarrow s_2 \wedge ID^{10}(s_1) = ID(s_2)$, v_1 and v_2 are alias. Such relationship is represented by $v_1 = v_2$.
- ***Shadowbox dependency graph (SDG)*** *:* A collection of points-to relationships : $\{(v,s)^* \mid v \rightarrow s\}$. For convenience, we use $SDG(v)$, SDG_v, or $SDG(\langle scope_v, name_v \rangle)$ to represent the shadowbox pointed by v.
- ***Fields information in shadowbox (FDG)*** *:* This is a variant of SDG : $\{(v,s)^* \mid v \rightarrow s \wedge v \in$ 'non-static fields in s'$\}$. Since FDG is always bound with a shadowbox s, we use FDG_s to indicate such relationship.

3.3 Type and Taint Analysis

$v_1 = v_2$ op v_3	$\Rightarrow SDG(v_1)_{taint} = SDG(v_2)_{taint} \mid SDG(v_3)_{taint}$
$v =$ new C	$\Rightarrow s =$ a new shadowbox; $s_{data\_type} = C$; $v \rightarrow s$
$v_1 = v_2$	$\Rightarrow v_1 \rightarrow SDG(v_2)$
$v \in C$	$\Rightarrow SDG(v)_{data\_type} = SDG(v)_{data\_type} \wedge C$
function $f(...)\{...; return\ r;\}$	$\Rightarrow \langle f, return \rangle \rightarrow SDG(r)$
	$for\ v \in SDG_{vertexes},\ delete\ v\ if\ v.scope == f$
$r = f(p_0, p_1, ...)$	$\Rightarrow \langle f, return \rangle \rightarrow null$
function $f(p'_0, p'_1, ...)$ $\{...\}$	$\langle f, p'_0 \rangle \rightarrow SDG(p_0)$; $\langle f, p'_1 \rangle \rightarrow SDG(p_1); ...;$
	$r \rightarrow SDG(\langle f, return \rangle)$
$v = o.e$	$\Rightarrow SDGv \rightarrow FDG_{SDG(o)}(e)$
$v = C.e$	$\Rightarrow SDG(v) \rightarrow \langle C, e \rangle$
$o.e = v$	$\Rightarrow FDG_{SDG(o)}(e) \rightarrow SDGv$
$C.e = v$	$\Rightarrow \langle C, e \rangle \rightarrow SDG(v)$
$a[i] = v$	$\Rightarrow$ Section 3.5
$v = a[i]$	$\Rightarrow$ Section 3.5

Table 1. Analysis Rules

Driven by the shadowbox concept, we define the analysis rules that implement type and taint analysis. The analysis rules work directly on Dalvik opcode[11]. We use lower case letters to represent variables, with the exception of e and f, which represent fields and functions, respectively. We use upper case letters for classes

[10] ID stands for the shadowbox's memory location in our static analysis.
[11] https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

or data types. In the rules, operations on array are solved with the help of value analysis, as shown in Section 3.5.

3.4 Value and String Analysis

Given a target variable v and its instruction i, v's value is calculated by performing backward programming analysis along the analyzed path to update its "expression tree". The expression tree is subtly different with the regular binary expression tree [1]. The expression tree's leaf nodes correspond to system APIs (e.g., $getDeviceId()$), constants, JavaScript Bridge function input, and variables whose values is to be calculated (i.e., *variable leaf*), and the internal nodes correspond to functions (e.g., $string.replace()$) and operators. Initially, the root node of the expression tree is v. Starting from i, all variable leaves in the expression tree are kept being updated. If it is found that a variable v_1 is dependent on another variable v_2 or an expression e_1, v_1's leaf node is replaced by v_2 or e_1. The analysis is continued till there are no variable leaves. To handle string operations, the associated functions are modelled. For example, the function $StringBuilder.append()$ itself is treated as an internal node, and its function parameters are added as its children nodes.

Then, the target variable' value can be retrieved by resolving the expression tree. For this purpose, the expression tree is first converted to a regular expression using in-order traversal. During the conversion, functions in internal nodes are converted to proper operators. For example, $StringBuilder.append()$ is replaced by $+$, and linked with the function's parameters (i.e., the children nodes). Then, we apply a lightweight solver to compute the expression's value, which is built on top of the Python function '$eval()$'.

3.5 Application Of Shadowbox Analysis

Path-Sensitivity. We use the code shown in Listing 1.1 as the illustrative example. Before utilizing shadowbox analysis on the test case, SDG is first created by initializing shadowboxes of '*this*' and the corresponding function's parameters with their associated data types. Then, the analysis is applied on each instruction based on the rules defined in Section 3.3. When a conditional statement c (Line 8) is encountered, the depth-first strategy is used and each path is analyzed sequentially. To keep the independence of each path, SDG is cloned and saved so that when a path is done, SDG is quickly restored for another path. Finally, when the sink API '$sendTextMessage()$' is encountered, the third parameter's shadowbox is read from SDG and checked to determine whether the parameter contains sensitive information.

SDG's content (when the branch statement is true) is partially shown in Figure 2.[12] By checking the shadowbox referenced by '$a.attr.b$' (the box with red line), we can learn that the third parameter is not tainted.

[12] Since most variable scopes are the same, scope information in variable representations is hidden to make SDG more concise.

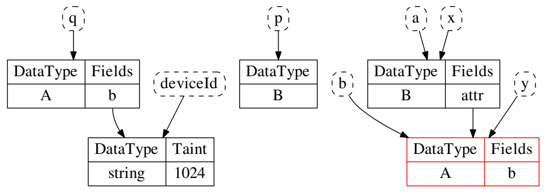

Fig. 2. SDG's partial content when sending text message, where cycles with dashed line are variable representations, and boxes with solid line represent corresponding shadowboxes.

HashMap and Linear Data Structures. The propagation problem caused by common data structures is due to their lack of regular field information, which makes it difficult to locate target variables. To mitigate this problem, we model common data structures using 'shadowbox', and augment common data structures by adding explicit fields to them that enable us to apply our analysis rules and handle them similar to regular data structures.

We use keys in a hashmap as the explicit fields, since keys are always unique. We leverage value analysis to retrieve the keys' values, which are then treated as fields. Thus the instructions '`value = H.get(key)`' and '`H.get(key) = value`' can be converted to assignment statements '`value =` $FDG_H(key)$' and '$FDG_H(key)$ `= value`', where H is an instance of hashmap.

We select the element position in linear data structures (such as list, array, Android Parcel, etc.) as the explicit fields. Thus the instructions '`value = array[index]`' and '`array[index] = value`' can be converted to assignment statements '`value =` $FDG_{array}(index)$' and '$FDG_{array}(index)$ `= value`'.

Most cases can be handled using the above intuition by computing $index$'s value in advance (Section 3.4), and converting it to a regular field. However, since an operation's $index$ value is changeable, such as injecting a new element in the middle of a list, or merging two lists, we maintain the data structures' internal state (which is represented by FDG) during updates. For example, consider if an element e is inserted into a list L at the position i through the API '$L.add(i, e)$'. FDG_L can be updated to

$$\begin{aligned} FDG'_L = &\{(v, FDG_L(v)) \mid v \in FDG_L.fields \wedge v < i\} \\ \cup\ &\{(i, e) \mid i \rightarrow e\} \\ \cup\ &\{(v+1, FDG_L(v)) \mid v \in FDG_L.fields \wedge v >= i\}. \end{aligned}$$

Similarly, operations in Android Bundle and Parcel are also supported.

4 BridgeScope

In this section, we present the design and implementation details of BridgeScope, and we explore the major challenges we encountered in detecting JavaScript Bridge problems and how BridgeScope intuitively solves these challenges.

4.1 Challenges and Solutions

Semantic gap between web and native code. This adds complexity to the analysis, especially when the suspicious command and control web code is not available, which is true for most cases.

To solve the problem, we assume that the code O loaded in WebView is ***omnipotent***, which means O has the capability to do anything through the JavaScript Bridge. Under this assumption, it is only necessary to focus on the analysis of JavaScript Bridge, which lowers the complexity and scope of our analysis.

However, actual code R loaded in WebView has the following relationship with O: $R \subset O$, which means our initial assumption introduces false positives to our analysis, as it may be not feasible for attackers to launch code injection attacks in some cases. For instance, if a benign app's communications with remote servers are always properly protected, then even when there is a sensitive bridge found in the app, it is still hard to exploit.

To reduce false positives, we consider the complexity for attackers to launch attacks (i.e., *attack complexity*). We discuss more details in Section 5.

Semantic gap between different WebView implementations. As discussed in Section 2.1, there are multiple WebView implementations in the Android platform. The major challenge is to eliminate the semantic gap between different implementations to enable a generic analysis. For example, the default Android WebView uses `addJavascriptInterface(BridgeObject,BridgeName)` to enable a JavaScript Bridge, while rhino-based WebView uses `putProperty(scope, BridgeName, BridgeObject)`. Similarly, the default WebView in Android 4.2 and above requires the annotation '@JavascriptInterface', while the default WebView in older Android versions and Rhino does not use any annotation feature.

Rather than specifically hard-coding and addressing each different implementation and their syntax differences, we observe that all implementations have key common elements that follow the model shown in Section 2.1. Armed with that observation, we address this challenge by translating different implementations to an intermediate representation. This gives us an abstraction that lowers the semantic gap and eliminates the diversity to allow analysis that is agnostic of the specific implementation.

Difficulty in identifying all JavaScript Bridges. A quick but naive solution to identify Android's default WebView in 4.2 and above, as well as Crosswalk and Chromeview, is to directly search for annotations. However, this approach may introduce false negatives because it is not generic, different WebView implementations do not use the same annotation syntax, and annotated functions may only be known at runtime. While our generic WebView model supports the annotation mechanism, it is still not possible to apply a simple search approach. Specifically, due to the well-known program analysis *points-to* problem [30], *BridgeObject* cannot be easily identified, meaning that functions which are annotated are only identifiable at runtime. Additionally, it is error-prone due to annotation inheritance.

To address this challenge, we leverage a shadowbox dependency graph (see Section 3), which we use to first identify all possible WebView implementations, and further identify JavaScript Bridges for each WebView according to the semantics of WebView.

During analysis, a key consideration is to maintain the status of variables, especially WebView, so that critical information can be quickly extracted, such as the pair $\langle BridgeObject, BridgeName\rangle$. Then, all JavaScript Bridges can be extracted using the 'shadowbox' data structure and its dependency graph (see Section 3).

Unknown semantics of functions in JavaScript Bridge. Generally, the native end of the JavaScript Bridge is a black box, since its source code is not always readily available. It is challenging to reconstruct the semantics of each function in a bridge (i.e., bridge function), but it is a critical step in undersanding the functionality to decide which bridge is sensitive. To solve the problem, we use fine-grained data flow analysis on all functions of JavaScript Bridges by tracking their parameters and system sensitive information.

Unknown input format of JavaScript Bridge. Even when a sensitive bridge is found, it is still challenging to validate it since appropriately formatted input data is required. We mitigate the problem by applying several heuristics information gathered from our analysis results, such as the data flow information, key API semantic, etc.

4.2 System Overview

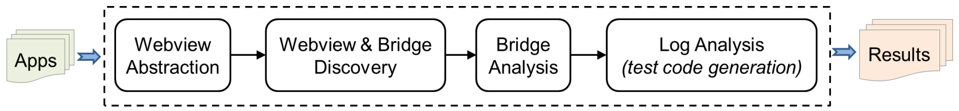

Fig. 3. Overview of BridgeScope

As shown in Figure 3, our static analysis approach BridgeScope consists of four main components: WebView abstraction, WebView and bridge discovery, bridge analysis, and log analysis. Given an app, the WebView abstraction module firstly disassembles it to the Dalvik bytecode[13] and then abstracts all WebView implementations in the app by translating the different implementations of WebView to an 'intermediate representation'.

Next, starting from entry points of activities [4, 21], type and value/string analysis based on shadowbox is performed to extract control flow graph (CFG), where type analysis is critical to resolve virtual function calls and solve points-to problem, and value/string analysis is useful to resolve Java reflection. Compared with existing approaches to generate CFG, our approach is fine-grained and complete.

In addition, during the process, value/string analysis is also run specifically for two situations: 1) when JavaScript is enabled or disabled in WebView, the key parameter's value is computed; 2) When the pair '$\langle BridgeObject$, $BridgeName\rangle$' is configured, $BridgeName$'s value is also computed.

Then, all methods in $BridgeObject$ are further analyzed by means of data flow analysis to identify sensitive bridges. Finally, the log analysis module collects

[13] https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

all analysis results from other modules and further generates heuristic information for the test attack code generation purpose.

4.3 WebView Abstraction

This module fills the semantic gap between different WebViews, which is done by translating different implementations of WebView into a generic 'intermediate representation'. We devise an 'intermediate representation' scheme, including common APIs (Table 2), and a generalized annotation mechanism.

API	Description
add_bridge(BridgeObject, BridgeName)	Add JavaScript Bridge to WebView
enable_js(boolean)	Enable/disable JavaScript
set_event_handler(event handler)	Register event handler
load(URL / local file / JavaScript code)	Run WebView

Table 2. Generic WebView common APIs

To support the annotation mechanism, which identifies Bridge Objects, we define the common annotation '*@JavaScriptBridge*' and apply it to all WebView instances, overwriting any specific annotation implementation such as in Android WebView in 4.2+ and Crosswalk.

We generalize WebView using shadowbox, whose structure is shown in Table 3. Generally, WebView contains three types of fields: 1) JsFlag, which indicates whether JavaScript is enabled; 2) Event Handler, which is used to react to different events (e.g., URL redirection, errors in web pages); 3) and JavaScript Bridge, which is a handle to a Bridge Object between the native and web context.

(a)

DataType	Fields			
WebView	JsFlag (¡)	EventHandler	*Bridge*#0	*Bridge*#1..

(b)

DataType	Fields	
Bridge	BridgeObject	BridgeName(¡)

Table 3. The generic model representation of WebView (a) and JavaScript Bridge (b). Note that we use the special symbol '¡' to indicate that when the associated field is initialized or changed, it should be computed by value/string analysis immediately.

4.4 WebView And Bridge Discovery

The goal of this module is to discover all WebView components and bridges. We apply type and value/string analysis based on shadowbox on the generalized WebView code (Section 4.3). This allows us to generate a complete control flow graph (CFG), and enables discovery of most WebViews and JavaScript Bridges within an app.

The analysis starts from entry points of Android Activities, since a WebView is almost always launched within an Activity. Even if a WebView is standalone or independent (such as Xbot [24]), we can still identify it after obtaining the CFG of the target app.

During analysis, data types of key variables, such as *BridgeObject*, are also for the further analysis (Section 4.5). Additionally, values of key variables, such as $JsFlag$ and $BridgeName$ (Section 4.3), are computed on demand with the help of value and string analysis. $JsFlag$ can be used to filter out WebViews whose JavaScript is disabled (i.e., $JsFlag$ is false), while $BridgeName$ is helpful in attack code generation.

4.5 Bridge Analysis

The goal of the module is to identify sensitive bridges from all bridges in $BridgeObject$. To achieve the goal, it is critical to reconstruct the semantics and learn the functionality of all exposed functions in $BridgeObject$ (i.e., bridge function), which are annotated with '@JavaScriptBridge' (Section 4.3). In BridgeScope, we apply taint analysis (Section 3) based on shadowbox on each function by tracking data flow of function parameters (TP) and system sensitive information (TS). To distinguish these two types of information, we define different taint value ranges : $[TP_{min}, TP_{max}]$, and $[TS_{min}, TS_{max}]$. Initially, parameters of a bridge function are tainted from left to right sequentially. Specifically, the nth parameter is assigned with the taint value $TP_{min} * 2^n$. During analysis, if a sensitive native API (Section 2.3) is invoked, we take a snapshot of their parameters' states (e.g., the associated shadowboxes), which will be analyzed further.

Finally, a bridge function's semantic information is reconstructed based on its data flow analysis result. A bridge function will be flagged as **sensitive** if: 1) its return is tainted by the value t while $t \in [TS_{min}, TS_{max}]$, 2) or a sink API $s()$ is called while $s()$'s parameters are tainted, 3) or a danger API is invoked. Based on the above three scenarios, we categorize all bridge functions into ***SourceBridge***, ***SinkBridge***, and ***DangerBridge***, correlating to the API categorization as defined in Section 2.3.

As a result, an app can be flagged as **potentially vulnerable**, if a sensitive bridge function f is found by BridgeScope. We use the following reasoning: 1) if $f \in SourceBridge$, it means that sensitive information can be obtained in the web context. Then, an attacker can send out sensitive information through network related APIs in the web context (like $XMLHttpRequest()$) or a sink JavaScript Bridge if it exists; 2) if $f \in SinkBridge$, security checks in event handlers in WebView, such as $shouldOverrideUrlLoading()$, can be evaded; 3) if $f \in DangerBridge$, a danger API can be accessed through f.

4.6 Log Analysis And Exploit Code Generation

Purpose	Collected information	Which module
Triggering WebView	UI Events	WebView & Bridge Discovery
Generating test code	$Domains$ associated with WebView	
	$\langle BridgeObject, BridgeName \rangle$	
	Semantics of bridge functions	Bridge Analysis
	SourceBridge,SinkBridge,DangerBridge	

Table 4. Collected Information

BridgeScope collects a rich set of heuristics information for the app under analysis as it executes each module (Table 4). This information is useful to further analyze flagged sensitive bridges and to generate test attack code. Furthermore, inspired by SMV-Hunter [29], we retrieve required UI events for triggering target WebViews by analyzing the result of the 'WebView and bridge discovery' module.

Algorithm 1 outlines our approach that leverages the above collected information to generate test code to verify discovered vulnerabilities. In the algorithm, a function $create\_input()$ is assumed to generate appropriate inputs for each bridge function. We implement it as a smart fuzzer using the following heuristics:

Algorithm 1 Test Code Generation

```
function GENERATE_TEST_CODE
    for $f$ in $SourceBridge$ do
        $input \leftarrow create\_input(f)$;
        $fname \leftarrow replace\_bridgeobject\_with\_bridgename(P,\ f)$;
        $add\_test\_code(X,$ "var r = fname(input)"$)$   ▷ append the JavaScript code to the result X
        for $d$ in $Domains$ do   ▷ bypass security check in event handler
            $add\_test\_code(X,$ "XMLHttpRequest(http://d/r)"$)$
        end for
        for $f'$ in $SinkBridge$ do
            $input' \leftarrow create\_input(f',$"r"$)$;
            $fname' \leftarrow replace\_bridgeobject\_with\_bridgename(P,\ f')$;
            $add\_test\_code(X,$ "fname$'$(input$'$)"$)$
        end for
    end for
    for $f$ in $SinkBridge \cup DangerBridge$ do
        $input \leftarrow create\_input(f)$;
        $fname \leftarrow replace\_bridgeobject\_with\_bridgename(P,\ f)$;
        $add\_test\_code(X,$ "fname(input)"$)$
    end for
    return $X$
end function
```

- Data Types: Based on data type information of parameters of bridge functions, which is gathered from type analysis, we can generate random but valid inputs [29].
- Bridge Function Name: The bridge function name itself also provides an important clue. For example, if a BridgeScope's name is $downloadImage()$ and the input is of type String, then input is likely a URI of a picture file. In our fuzzer, we handle several keywords, such as "url", "email", "picture", "camera", "audio", "video", "SMS", "call" to provide typical input values.
- Semantics of bridge functions and key native APIs: We can also build input by utilizing the semantic information. For instance, assume there is a path in CFG from a bridge function to a sensitive API : $f(p_0 : string, p_1 : string) \rightsquigarrow sendTextMessage(v_0, null, v_2, null, null)$, where v_0 and v_2's taint values are $TP_{min} * 2$ and $TP_{min} * 4$, respectively. The data flow in the bridge function includes $p_0 \rightsquigarrow v_0$ and $p_1 \rightsquigarrow v_2$. Since in $sendTextMessage()$, v_0 is the destination address, and v_2 is the message content to be sent, p_0 and p_1 are likely a phone number and message content. Therefore, the following input can be used to test the sensitive bridge function: `f("0123456789", "test")`.

5 Evaluation of BridgeScope

In this section, we present our evaluation of BridgeScope. First, we measure the performance of the programming analysis techniques by leveraging the generic benchmark DroidBench. Then, we evaluate BridgeScope's efficacy, precision, and overhead using 13,000 popular apps, and present our findings. Finally, we present some interesting case studies to illustrate the JavaScript Bridge vulnerability.

5.1 Performance Of Shadowbox Analysis

We evaluate the precision of shadowbox analysis using the generic benchmark DroidBench 2.0. Our test results (Table 5) show BridgeScope's overall precision is 94%, compared to 80% and 91% for Flowdroid [4] and Amandroid [33], respectively, and BridgeScope's recall and F-score are also better than the others. Our use of shadowbox analysis benefits from its path- and value-sensitivity, and it is fine-grained, especially in handling common data structures.

DroidBench	**BridgeScope**	**Flowdroid**	**Amandroid**
Aliasing		◯	◯
Android specific	××	××	× × ×
Arrays and lists		◯ ◯ ◯◯	◯ ◯ ◯ ◯ ×
Callbacks	××	◯ ◯ ×	◯ ◯ ◯ ◯ × × ××
Emulator detection	××	××	××
Field/Object Sensitivity			
General java	◯ ◯ ××	◯{4} × × × ×	◯ × ×
Implicit flows	–	–	–
Interapp communication		◯ ◯ ◯	
ICC	× × ××	◯{16} × × × ×	◯ × × × ××
Lifecycle	◯×	◯ × ×	× × ××
Reflection		× × ×	
Threading	×	×	×{5}
Totally Found Paths f	100	127	101
Precision $p = f/(f + \bigcirc)$	94%	80%	90%
Recall $r = \bigcirc/(\bigcirc/ + \times)$	83%	82%	75%
F-score $2 * p * r/(p + r)$	0.89	0.81	0.82

Table 5. Testing Result on DroidBench. × represents suspicious data flows not detected, and ◯ represents benign data flows flagged as suspicious. The number in {} represents the number of errors.

5.2 Performance of BridgeScope

Dataset. We use 13,000 apps that were collected from the Google Play app market. We crawled these apps from 26 categories, and extracted the top 500 most popular free apps for each category.

Scalability. We implemented BridgeScope in 8,157 lines of Python code on the top of the disassembly tool Apktool[14]. We deployed our tool on a university server, which is allocated with 20 CPU cores and 100 GB of memory. Due to Python's poor support for multiple threads, we run single process and single thread for the analysis (i.e., starting 20 processes for 20 apps each time). Finally, with the boost of the JIT (Just-in-Time) based Python interpreter (such as pypy[15]), the average analysis time of each process is 141 seconds. Thus, the average analysis time for each app is around 7 seconds. This suggests that BridgeScope is indeed capable of scaling to the level of real-worlds app markets to provide vulnerability detection services.

Precision. Among 13,000 apps, we find that 11,913 apps have at least one WebView component and 8,146 apps declare at least one JavaScript Bridge

[14] https://ibotpeaches.github.io/Apktool/

[15] https://pypy.org/

interface. In total, 913 apps were flagged as potentially vulnerable apps by BridgeScope, while a total of 1,530 sensitive bridge functions were found, including 56 bridge functions which could suffer from SOP Violation Attacks (Section 2.2).

Measuring false positives and negatives. A false positive occurs when an app is flagged as potentially vulnerable by BridgeScope, but has no vulnerability. A false negative occurs when an app is flagged as non-vulnerable by BridgeScope, but includes a JavaScript Bridge vulnerability.

Since it is hard to directly collect ground truth for the dataset, manual verification may be necessary, which is a difficult and tedious job for such a large dataset. To reduce the workload, we first design a dynamic verification module to automatically validate the potentially vulnerable apps (thus we do not need to manually validate all data) when analyzing false positives. Additionally, we manually analyzed a small set of 20 randomly chosen apps from those not marked as potentially vulnerable, which we used as the basis of measuring the false negative rate.

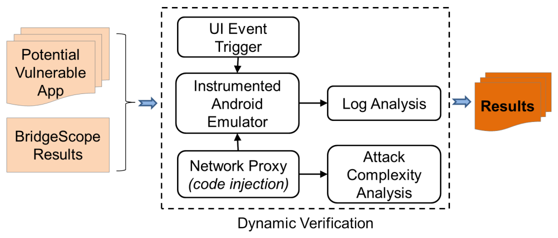

Fig. 4. Overview of the dynamic verification module

As shown in Figure 4, our dynamic verification module is built around an *instrumented Android Emulator*, where all executed functions of apps under test are outputted, sensitive information is modified to special values, and native sink APIs parameters (e.g., WebView.load) are also outputted. In the module, *UI Event Trigger* [29] is used to input UI required event sequentially to trigger target WebViews, while *Network Proxy* is used to launch MITM attacks to inject attack code, which is generated using the example algorithm mentioned earlier (Algorithm 1).

In our evaluation we also hijack all network traffic, including HTTPS, so that we can further analyze the complexity faced by attackers who launch code injection attacks (i.e., *Attack Complexity Analysis*). We mainly consider three scenarios : 1) *HTTP*: the remote server is connected over HTTP; 2) *first-party HTTPS*: the remote server belonging to developers is connected over HTTPS; 3) *third-party HTTPS*: others. In Attack Complexity Analysis, we use the URL loaded by the WebView as input, and initiate a crawler to check all accessible URLs, similar to the approach in [14].

Finally, we check whether a potential vulnerability is successfully exploited by analyzing logs from the Android Emulator and proxy (i.e., *Log Analysis*). If a bridge function f satisfies: 1) $f \in SourceBridge$, it can be verified by checking executed functions, sink API's parameters and proxy's traffic. 2) $f \in SinkBridge \cup DangerBridge$, it can be verified by checking executed functions.

False Positives. By means of the dynamic verification module, we found that 617 potentially vulnerable apps flagged by BridgeScope are successfully exploited (i.e., they are surely not false positives). This reduces our manual verification job to only 296 non-verified potentially vulnerable apps. We then randomly selected 20 apps and manually analyzed them. We found most of them still contain vulnerable bridges that could be exploited. The reason they are missed by the dynamic verification module is because the dynamic module uses heuristics but cannot guarantee the completeness. For example, it may not always generate proper input formats of the JavaScript Bridges, such as the JSON string. There are 4 apps that use WebView to load local HTML files instead of connecting to Internet. While these 4 apps could be considered as false positives of BridgeScope (because our assumed network adversary may not be able to inject attack code in this case), we argue that they could still be vulnerable/exploited in an extended threat model in which external HTML files are not trusted (which could be also reasonable considering that these files could be manipulated by malicious apps in the phone).
False Negatives. To evaluate the false negatives of BridgeScope, we randomly selected 20 apps from those non-potentially-vulnerable apps that had at least one WebView. Thorough manual review and testing (almost 1 hour per app) of how the WebViews are used in those 20 apps, showed that none were potentially vulnerable, suggesting that indeed our false negative rate is relatively low.

5.3 Overall Findings

Diverse WebView implementations. Based on our static analysis result, we found that WebView implementations are indeed diverse. Table 6 shows the distribution of different WebView implementations in our dataset.

Android Default WebView	Mozilla Rhino Based WebView	Chromeview	XWalkView	Total
11,823	526	20	0	11,913

Table 6. Diverse WebView Implementations

Evadable Security Checks in WebView event handlers. As shown in Section 2, event handlers perform security checks on the URL to be connected. However, in our evaluation we found that the customized event handler did not properly protect sensitive information leakage. Once sensitive information is successfully obtained in the web context, it can always be directly sent out through a JavaScript API or by dynamically creating DOM elements [9].
Attacking capability. To further understand the attack capability on those potentially vulnerable apps, we analyze the different sinks and sensitive APIs of those confirmed potentially vulnerable apps and summarize the attack capabilities shown in Table 7. The most common attack enabled is to steal private information from content providers. This is due to the fact that a large number of potentially vulnerable apps use sensitive JavaScript Bridges to load authentication tokens from content providers. We also observe that attackers can launch diverse attacks including some critical attacks such as sending text messages, sending emails, and playing videos.

Attack Capability	App Number	Attack Capability	App Number
Leaking Content Provider Content	241	Sending text message by intent	57
Leaking the Device ID	42	Sending email by intent	51
Leaking phone numbers	14	Playing video by intent	61
Directly sending text message	2	Create Calendar by intent	171
Downloading/Saving Picture	344	SOP Violation Attack	41

Table 7. Attacking Capability Distribution

Network Channel	**HTTP**	**Third-Party HTTPS**	**HTTPS**
Difficulty	Easy	Medium	Hard
Number	224	103	290

Table 8. Difficulty to exploit vulnerabilities

Attack complexity. To reduce the false positive caused by our analysis assumption (Section 4.1) and further understand the relative difficulty of launching attacks on vulnerable apps, we define three attack complexity levels:

– *Hard :* The content in a vulnerable WebView is loaded over first-party HTTPS. In this case, those vulnerable JavaScript Bridges could be intentional bridges to the trusted JavaScript in the first-party content. However, it could still be attacked by hijacking HTTPS traffic [3], especially considering that HTTPS can be very poorly/insecurely implemented or used in mobile apps [11,13].
– *Medium :* The vulnerable WebView loads third-party content over HTTPS. It faces similar risks as above [3,11,13]. In addition, attackers could compromise third-party content (such as through a Content Delivery Network [20]) to inject the malicious JavaScript.
– *Easy :* The vulnerable WebView loads web content through HTTP. In this case, attackers can easily inject the malicious JavaScript into HTTP traffic.

Based on the above definitions, Table 8 shows the results of attacking complexity analysis of our automatically verified vulnerable apps. We can see that the majority of vulnerable apps are hard to attack, but we also note that most apps that fall into this category contain JavaScript Bridges that explicitly allow trusted JavaScript to access sensitive information from users. In other words, as long as the transport protocol is compromised, attacker capabilities are enhanced. Recent disclosures of the fragility of HTTPS [5,6] makes this scenario more trivial than not.

We also observe that there exists a large number of vulnerable apps using the HTTP protocol, which can be obviously easily attacked through code injection since communication is in clear text.

5.4 Case Studies

We present two interesting case studies of vulnerable apps here. In the interest of responsible disclosure, we avoid naming the specific apps at this time while we notify developers and coordinate possible fixes.

Case 1 : Advertisement library vulnerability. In this case, the vulnerable app loads an advertisement library, which is a common practice in app development. However, this ad library contains a vulnerable WebView, which is used to communicate with the advertiser's website to retrieve advertising content over the HTTP protocol. BridgeScope detects only one vulnerable JavaScript Bridge imported into this vulnerable WebView. However, 56 methods are available in this vulnerable JavaScript Bridge. Among them, 19 are critical methods, which can be invoked by attackers to steal sensitive information (such as device ID, WIFI status, network operator name, and user's internal data) and download or delete files and folders in the device.

We found 12 apps in our dataset that used this vulnerable advertisement library, making all of them equally vulnerable.

Case 2 : Vulnerable browser apps. Developers often extend WebView to quickly create and specify their own customized browser apps. Many specialized 'browsers' on the app market use this model. We crawled 100 popular browser apps from Google Play in January 2016. 74 of them are merely extensions of the standard WebView. BridgeScope successfully detected 6 vulnerable browser apps that can be exploited to leak sensitive information such as device ID (5 apps), phone number (1 app), serial number (1 app).

We also found one popular browsers app, downloaded by more than 100,000 users, which suffers from SOP Violation Attacks. The app is designed to provide an ad-free user experience by filtering out ads using a blacklist. A bridge function, named '$applyAdsRules(String\ url)$', checks whether the url is related to an advertisement website. If the url is 'safe', it will be sent to the app's main Activity to render it using the key API $WebView.loadUrl(url)$. This fits the pattern of the SOP violation attack, giving an attacker the ability to load content that he knows not to be blacklisted by the app's filter function to launch client-side XSS attacks.

Different from other apps, these browser apps have much larger attack surfaces since the website (e.g, third-party) to be accessed and the protocol used in communications (e.g, HTTP or HTTPS) are specified by users, making them relatively easy to attack by simply redirecting a user to an attacker-controlled website.

5.5 Results on Real-world Malware

In addition to finding potential vulnerabilities in benign apps, we also test our tool on real-world malware that uses JavaScript Bridge techniques. By searching reports from Honeynet [2] and Palo Alto Networks [24], we collected 23 malicious apps that were reported to employ JavaScript Bridge techniques.

By running BridgeScope on these malicious apps, we found a total of 68 sensitive bridges. Although the malicious servers were already down, BridgeScope still successfully identified malicious behaviors hidden in JavaScript Bridges, including leaking of sensitive information, sending text messages, and prompting fake notifications, which are the same as the report descriptions about these malware by Honeynet [2] and Palo Alto Networks [24].

6 Discussion

Limitation in Static Analysis. Similar to other existing static analysis tools [4, 33], our work does not handle implicit data flow, or low level libraries written in C/C++, which may lead to false negatives. However, C/C++ library could be mitigated by modeling their functions, such as $system.arraycopy()$. We leave implicit data flow tracking as our future work.

More comments on HTTPS. In this paper, some of detected vulnerable apps require hijacking HTTPS in order to exploit them. We consider that while HTTPS may pose a higher level of complexity and difficulty for exploiting JavaScript Bridge vulnerabilities, it is still a realistic threat vector because HTTPS is widely implemented insecurely/poorly in mobile apps [11, 13] and several recent high profile works also showed the inherent issues of HTTPS [5, 6, 20]. Therefore, once attackers can successfully hijack HTTPS, they can exploit our reported vulnerabilities to launch diverse critical attacks (shown in Table 7).

7 Related Work

WebView Security. Luo et al. [22] exposed attack vectors in WebView, and demonstrated the JavaScript Bridge vulnerability. Chin et al. [8] analyzed WebView vulnerabilities that result in excess authorization and file-based cross-zone scripting attacks. Mutchler et al. [23] did a generic large scale study on security issues (such as unsafe navigation and unsafe content retrieval) in Android apps equipped with WebView. Wu et al. [34] discussed file leakage problems caused by file:// and content:// schemes in WebView. Georgiev et. al. [14] did a study on a popular third-party hybrid middleware frameworks. Hassanshahi et. al. [17] studied the security issues caused by intent hyperlinks.

The JavaScript Bridge vulnerability is rooted in the conflict between security models of the native and web context [14], and the lack of privilege isolation [19]. The approach *NoFrak* proposed by [14] partially solves the conflict by extending the web's same original policy (SOP) to the local resources. Other works such as MobileIFC [28] also propose a similar concept of extending SOP to mediate access control between the mobile and web context within a hybrid app. Jin et. al. [19] proposed a defense solution for JavaScript Bridge vulnerabilities in hybrid apps, with focus on privilege separation based on iFrame instances within the WebView. In [31], the authors proposed Draco, a uniform and fine-grained access control framework for web code running in Android default WebView.

Privacy Detection And Protection. Taint analysis is an effective approach for detecting privacy leakage. On Android, systems such as TaintDroid [10] and FlowDroid [4] are among some of the most well-known taint-based systems. Existing Android analysis tools [4, 7, 12, 15, 33] may be useful for detection of vulnerabilities. However, existing work either performed coarse-grained analysis, or imposed high performance overhead [7, 18]. Furthermore, existing work could not handle the semantics of JavaScript Bridge and diverse WebView implementations.

8 Conclusion

The integration of mobile and web through the use of WebView requires compromises to be made in the security of both platforms. Subsequently, we find that the current design and practices in the implementation of WebView causes a class of generic vulnerabilities that can be exploited by attackers to cause serious problems on mobile devices. We implement an analysis framework, BridgeScope, which can automatically discover vulnerabilities in a hybrid mobile app and generate test attack code that is then automatically verified as a feasible exploit. Our system is implemented in Android, and we provide evaluation that shows our system is a feasible approach to automatically and precisely discover vulnerabilities at large scale.

Acknowledgments

This material is based upon work supported in part by the the National Science Foundation (NSF) under Grant no. 0954096 and 1314823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

1. Binary Expression Tree. `https://en.wikipedia.org/wiki/Binary_expression_tree`.
2. Is android malware served in theatres more sophisticated? `http://www.honeynet.org/node/1081`.
3. D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF)*, 2010.
4. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.
5. N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. Drown: Breaking tls using sslv2. In *USENIX Security*, 2016.
6. B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *IEEE Symposium on Security and Privacy*, 2015.
7. S. Calzavara, I. Grishchenko, and M. Maffei. Horndroid: Practical and sound static analysis of android applications by SMT solving. In *IEEE European Symposium on Security and Privacy, EuroS&P*, 2016.
8. E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications*, pages 138–159. Springer, 2014.
9. S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *NDSS*, 2016.
10. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.
11. S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *ACM CCS*, 2012.
12. A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland*, 2009.
13. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *ACM CCS*, 2012.

14. M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS*, volume 2014, 2014.
15. M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. In *NDSS*, 2015.
16. N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.
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 *ESORICS*, volume 9327, pages 577–598. Springer, 2015.
18. W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. *ISSTA*, pages 106–117, 2015.
19. X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in android. In *Information Security*, pages 309–318. Springer, 2015.
20. J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu. When https meets cdn: A case of authentication in delegated service. In *IEEE Symposium on Security and Privacy*, 2014.
21. L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM CCS*, 2012.
22. T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ASCAC*, 2011.
23. P. Mutchler, A. Doupe, J. Mitchell, C. Kruegel, G. Vigna, A. Doup, J. Mitchell, C. Kruegel, and G. Vigna. A Large-Scale Study of Mobile Web App Security. *MoST*, 2015.
24. P. A. Networks. New Android Trojan "Xbot" Phishes Credit Cards and Bank Accounts, Encrypts Devices for Ransom. `http://researchcenter.paloaltonetworks.com/2016/02/new-android-trojan-xbot-phishes-credit-cards-and-bank-accounts-encrypts-devices-for-ransom/`.
25. S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. *NDSS*, pages 23–26, 2014.
26. V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces. *NDSS*, 2016.
27. S. Sedol and R. Johari. Survey of Cross-site Scripting Attack in Android Apps. *International Journal of Information & Computation Technology*, 4(11):1079–1084, 2014.
28. K. Singh. Practical context-aware permission control for hybrid mobile applications. In *Research in Attacks, Intrusions, and Defenses*, pages 307–327. Springer, 2013.
29. D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. *NDSS*, 2014.
30. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, New York, NY, USA, 1996.
31. 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 *ACM CCS*, 2016.
32. R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM CCS*, 2013.
33. F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM CCS*, 2014.
34. D. Wu and R. K. C. Chang. Indirect File Leaks in Mobile Applications. *MoST*, 2015.