Dual-Force: Understanding WebView Malware via Cross-language Forced Execution

Zhenhao Tang[†], Juan Zhai[†‡], Minxue Pan[†], Yousra Aafer[‡], Shiqing Ma[‡], Xiangyu Zhang[‡], Jianhua Zhao[†]
[†] Nanjing University, China
[‡] Purdue University, USA
tangzhnju@gmail.com, {zhaijuan, mxp}@nju.edu.cn, {yaafer,shiqingma}@purdue.edu,
xyzhang@cs.purdue.edu, zhaojh@nju.edu.cn

ABSTRACT

Modern Android malwares tend to use advanced techniques to cover their malicious behaviors. They usually feature multi-staged, condition-guarded and environment-specific payloads. An increasing number of them utilize WebView, particularly the two-way communications between Java and JavaScript, to evade detection and analysis of existing techniques. We propose Dual-Force, a forced execution technique which simultaneously forces both Java and JavaScript code of WebView applications to execute along different paths without requiring any environment setup or providing any inputs manually. As such, the hidden payloads of WebView malwares are forcefully exposed. The technique features a novel execution model that allows forced execution to suppress exceptions and continue execution. Experimental results show that Dual-Force precisely exposes malicious payload in 119 out of 150 WebView malwares. Compared to the state-of-the-art, Dual-Force can expose 23% more malicious behaviors.

CCS CONCEPTS

• Security and privacy → Malware and its mitigation; **• Theory of computation → Program analysis**;

KEYWORDS

WebView malware, forced execution

1 INTRODUCTION

Nowadays, Android malware normally uses advanced techniques to cover its malicious behaviors [22]. It usually features multi-staged, condition-guarded and environment-specific payloads. For instance, a potentially harmful Android application only exhibits its malicious payload when it passes integrity check and is executed in a real device, with a targeting application running, during some period of time, in specific countries. Besides, some applications are controlled by remote command and control (C&C) servers, through which hackers can control what kinds of attacks can be launched, and when and where these attacks are going to happen. To make things worse, an app may contain malicious payloads that are not activated in the current version, but could be enabled in newer versions. In this case, an arbitrary dynamic execution of a potentially harmful application is highly likely to be benign.

In addition, an increasing number of Android malware samples utilize the WebView technique [1] to evade detection. WebView allows Android applications to display web contents within an app, which is particularly useful when the data and layouts of the contents are frequently updated from servers. The power of WebView is magnified by enabling Java and JavaScript interoperability. However, WebView makes the behaviors of potentially harmful apps more difficult to understand and reason. For example, some Android malware uses WebView to overlay phishing pages on other popular apps like banking apps to lure users to enter credentials like passwords and then exfiltrate them to remote servers. This is a severe threat to privacy and property security. What's more, a malicious app can deliberately hide the malicious payload deeply in the two-way communications between Java and JavaScript.

Current Android malware detection techniques have limitations in systematically dealing with malwares implemented using WebView. Static analysis [7, 9, 10] cannot deal with dynamic features, but dynamic DEX loading on Android is very common and most JavaScript that runs on WebView is only known at runtime. Dynamic analysis [6, 12, 14, 24, 26, 28, 30], usually has dismal performance when encountering apps with carefully designed and hidden malicious payloads, and the existence of WebView makes it more difficult to analyse these malwares. Symbolic and concolic analysis need to effectively model various aspects of Android and the apps, such as intents and cross-language features, which is highly challenging. It also suffers from potential efficiency problems for large real-world applications.

In this paper, we propose *Dual-Force*, a forced execution technique which forces Java and JavaScript code to execute along paths of interest for Android applications without any environment setup to expose malicious behaviors. Dual-Force monitors the execution of an application from both the Android runtime perspective and the JavaScript engine perspective. It systematically forces a small set of instructions that could affect the execution path to have specific values, based on certain exploration strategies. When inputs are required for apps to run, we feed random values or values collected through additional analysis. Forcing execution paths and providing random values likely cause exceptions. Dual-Force features a novel runtime that can suppress various kinds of exceptions and allow the execution to proceed to expose behaviors hidden deep in the state space.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09...$15.00
https://doi.org/10.1145/3238147.3238221

We evaluated Dual-Force on 150 Android malware samples randomly obtained from VirusTotal, Koodous and Contagio mobile mini dump. The results demonstrate that Dual-Force is capable of exposing potentially harmful behaviors for 119 malicious apps. Dual-Force identifies 23% additional malicious behaviors that cannot be found by the state-of-the-art techniques. The malicious behaviors detected by Dual-Force include retrieving private personal information, targeting reputed apps for credentials, intercepting SMS messages and incoming calls.

Our main contributions are summarized as follows:

- We propose Dual-Force, a forced execution technique that simultaneously forces Java and JavaScript code to execute along various paths of interest to better understand behaviors of WebView applications.
- We develop a crash-free forced execution model that can recover from exceptions properly for WebView applications.
- We identify technical challenges of dealing with WebView applications and propose techniques to address them, e.g., supplying appropriate values to the executions on-demand.
- We have applied this technique to 150 WebView malwares. The results show that Dual-Force can expose malicious behaviors for 119 samples, many of which utilize WebView to hide or complicate their malicious behaviors, and 23% of the exposed behaviors cannot be found by exisiting techniques.

2 BACKGROUND

2.1 Entry Points

Each application has entry points for the system or a user to enter. The entry points can be divided into two categories. The first category refers to app components, which are building blocks of an Android app. To be launched by the Android system, a component must have itself registered in the Android manifest file *AndroidManifest.xml*. There are four different types of app components: activities, services, broadcast receivers and content providers. Activities, services, and broadcast receivers are activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime. The second category refers to Java methods annotated with *@JavascriptInterface* that can be invoked by JavaScript running in WebView.

2.2 WebView

WebView is a fully functional browser that can be integrated into Android applications. Android applications utilize WebView to display web contents within the app. This technique offers great flexibility for developers as well as simplifies user interface (UI) design. Developers can put rich contents in web pages without using widgets provided by the Android system. Views are basically web pages and the app just needs to render it using WebView without considering the details such as layouts. Many Android apps make heavy use of this technique. For instance, Amazon and eBay use WebView to display home pages with ongoing and upcoming promotions, and the details of newly added items.

To make this technique more powerful, Android also allows WebView to run JavaScript. Java code and JavaScript code can invoke each other at runtime: 1) Java calls JavaScript through the invocation to the Android API, such as *WebView.loadUrl*. The arguments of *loadUrl* can be a web page containing JavaScript scripts or a dynamically generated JavaScript string literal starting with "javascript:"; 2) JavaScript code invokes Java methods annotated with *@JavascriptInterface*. Note that Java methods can be dynamically attached with this annotation by invoking the API method *WebView.addJavascriptInterface*. With WebView, Android apps can achieve cross-language interoperability between Java and JavaScript.

2.3 Execution

An Android app starts its execution by initiating one of its components. Typically, clicking the app icon in the top-level application launcher triggers its main activity. Besides, a component can be invoked programmatically by other components from the same app and other apps through inter-component communications (ICC) [16].

WebView starts as a demon service when the Android system boots up. It starts working when an app calls the WebView API at some point. JavaScript code is eventually passed to the back-end JavaScript engine, in which it is interpreted and executed asynchronously. JavaScript may call Java methods in turn in a synchronous way.

3 MOTIVATING EXAMPLE

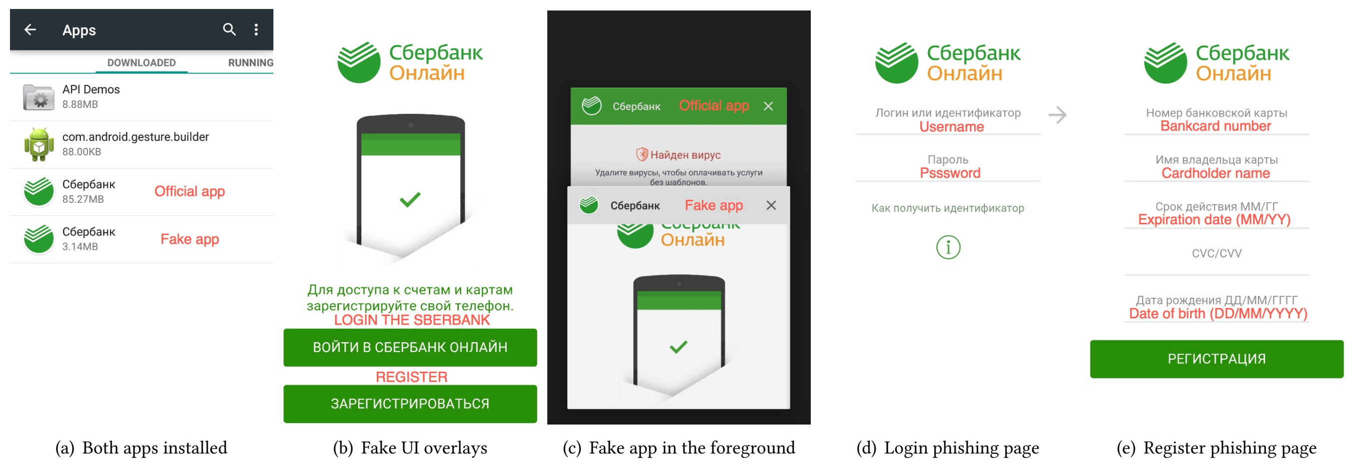
In this section, we use a WebView malware sample to illustrate the challenges of exposing its payload effectively and how we address the challenges using our forced execution approach.

In this example, the malicious app pretends to be the banking app of Sberbank, a reputable Russian bank. From Fig. 1(a) we can see that the fake app looks exactly the same as the official one. Among all its malicious behaviors, the most dangerous feature of the malware is the capability of targeting other apps via overlay WebView pages, which lures users to enter their private information and sends it to the remote servers. Naturally, the official app is one of the targets, as specified in a configurable preference file named *interceptor.xml* in Fig. 2.

The malicious app attacks the official banking app whose package name is "ru.sberbankmobile" by starting the phishing pages that are stored locally. When the official app is launched, the phishing pages are displayed right over the official banking app, as shown in Fig. 1(b). We can confirm this by checking the task manager in Fig. 1(c), which shows that it is the fake app instead of the official one that is running in the foreground. When the user clicks the buttons on the phishing pages, another two phishing pages are displayed asking the user to enter his/her user name, password and bank account information, as shown in Fig. 1(d) and Fig. 1(e). When the user enters his/her credentials and tries to login/register, the private information is uploaded to a remote C&C server. This leak is a direct threat to privacy security and property safety.

To show the logic of this attack, we reverse-engineered the code from the malicious app and simplify it by removing irrelevant statements and exception handlers, as well as renaming variable names for readability. The simplified code is shown in Fig. 3.

The attack is initially launched by the method *doInBackground* of the class *MasterInterceptor*, which repeatedly reads the shared preference file *interceptor.xml* (line 5). It parses the file and stores



(a) Both apps installed (b) Fake UI overlays (c) Fake app in the foreground (d) Login phishing page (e) Register phishing page

Figure 1: How the attack is launched and the deceiving phishing pages.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <map>
3   <string name="ru.sberbankmobile">
4     javascript:
          MeSetting.startPage("http://android_asset/2/index.html");
5   </string>
6 </map>
```

Figure 2: An instance of *interceptor.xml* that would trigger the overlay payloads of the malicious app.

the entries in a map, where each entry is a mapping between a package name and a string. If the map is empty, which means there is no targeted app, the loop breaks and starts over (lines 6-7). Otherwise, it tries to get the currently running app in the foreground (line 9). After that, the map is iterated to check if it contains a key that equals to the name of the running app (lines 10-13). If so, it creates an intent of the class *GlobeCode* and puts extra staff including the string it reads from the preference file into the intent (lines 14-16). Then the intent is started as a service at line 18. When the intent is started, the method *onStartCommand* of the class *GlobeCode* begins to run. It first retrieves the string from the "content" field (line 26) and makes all the methods in class *MeSetting* callable from JavaScript (lines 27-28). It then checks if the string starts with "javascript:". If so, the JavaScript code is then called in lines 31-32.

The JavaScript code shown in Fig. 3 is embedded in the login phishing web pages. It first checks if the user name and password fields are correctly filled in (lines 47-48). If either field has a length less than five, then the HTML form cannot be submitted (line 51). It then registers a callback function for the submit button for submitting the form (lines 54-60). Finally, it changes the default *action* of the form to a uniform resource identifier (URI) at lines 61-62. The URI is constructed by string concatenation, where the domain name is retrieved by calling a Java method *MeSetting.getDomain*, which returns the domain name that is stored in the Android manifest file (lines 37-43). Note that all methods in the class *MeSetting* are designed to be callable from JavaScript, as shown in lines 27-28, where the method *addJavasriptInterface* is called for an instance of class *MeSetting*.

This example poses challenges for traditional and the state-of-the-art analyses. Static analysis cannot expose the complete logic behind the malicious behavior since it is unaware of the existence of the JavaScript code (lines 46-62). Dynamic analysis is unable to find the overlay payload unless the target app specified by the preference file is actually running in the foreground. Symbolic/-concolic analysis needs to model files (line 5), intents (line 16) and cross-language communications (lines 31, 32 and 61), which is highly challenging. The state-of-the-art work [21], which proposes a targeted fuzzing framework that combines an extensive number of hybrid techniques, is unlikely to generate an environment that triggers the payload because it has to make the targeted app run in the foreground. It also needs to fill in the login information with two strings whose lengths are bigger than five and then click the submit button.

Dual-Force deals with this example by force-executing the app on both Android and WebView. The basic idea is to forcefully switch the outcomes of a small number of branch predicates. Choosing which branch outcomes depends on specific exploration strategies. For example, if a branch condition is evaluated to be the same value for consecutive five times, then we switch its outcome. Because of space limitation, here we omit the branch switchings that do not produce useful information. Suppose that we are analyzing the behavior of the malicious app. Then it is probably always running in the foreground. Consequently, the branch outcome at line 13 is always *false* for the first five executions, because the package name of the running app does not match "ru.sberbankmobile". Then Dual-Force switches the branch outcome to *true*, which makes the app run into the method *GlobeCode.onStartCommand* through inter-component communications. In this method, the overlay phishing pages are displayed by WebView (lines 31-32), where the JavaScript code is going to be executed. The user name and the password fields on the phishing pages are initially empty and we do not fill them in manually. As a result, the branch condition at lines 47 to 48 always resolves to be *false* for the first five executions, making the

```
// Java code
1 class MasterInterceptor {
2  protected Object doInBackground(Object... paramVarArgs) {
3    for (;;) {
4      Thread.sleep(500L);
5      Map localMap = getSharedPreferences("interceptor", 0).getAll();
6      if (localMap.size() <= 0)
7        break;
8      String runningApp;
9      runningApp = getActivePackagesCompat();
10     Iterator iter = localMap.keySet().iterator();
11     while (iter.hasNext()) {
12       String str = iter.next();
13       if (str.equals(runningApp)) 1 {
14         Intent localIntent = new Intent(getApplicationContext(),
15                                         GlobalCode.class);
16         localIntent.putExtra("content", localMap.get(str));
17         ...
18         MasterInterceptor.this.startService(localIntent);
19       }}}}
20 }
21 class GlobeCode {
22  public int onStartCommand(Intent paramIntent, /*...*/) {
23    if (paramIntent != null) {
24      Object localObject = paramIntent.getExtras();
25      if (localObject != null) {
26        String content = getexstras((Bundle)localObject, "content");
27        localWebView.addJavascriptInterface(
28            new MeSetting(getApplicationContext()), "MeSetting");
29        if (content.substring(0, 11).contains("javascript:")) {
30                                   2
31            localWebView.loadData("<script>" + paramIntent.substring(11)
32              + "</script>", "text/html; charset=UTF-8", null);
33        }}}}
34 }
```

```
// JavaScript interfaces: Java methods callable from JavaScript
35 public class MeSetting {
36  ...
37  public String getDomain(){
38    String str = "";
39    ...
40    str = localApplicationInfo.metaData.getString("domain");
41    ...
42    return str;
43  }
44  ...
45 }
```

```
// JavaScript code
   <script>
46 function SBank() {
47   if ($('#sbol-login').val().length >= 5 &&
48       $('#sbol-password').val().length >= 5) 3 {
49     $('#send-sbol').prop("disabled", false);
50   } else {
51     $('#send-sbol').prop("disabled", true);
52   }
53 };
54 $('#send-sbol').click(function() {
55   $('#myformsbol1').fadeOut(1000, function() {
56     $('#error2').fadeIn(500).delay(100, function() {
57       document.myformsbol1.submit();
58     });
59   });
60 });                                                         4
61 document.myformsbol1.action = "http://"+MeSetting.getDomain()+
62                               "/api/indata.php?type=SBankFull";
   </script>
```

Figure 3: The simplified code that shows the logic behind the overlaying attack.

form unable to be submitted (line 51). Dual-Force then switches the outcome to *true* to make the form enabled (line 49). Note that the function *SBank* is triggered by timeout events (not shown here in the code though) and the anonymous function (lines 54-60) is triggered by clicking events on a button. We force them to run after the web page is loaded. Consequently, an overlay phishing payload is successfully exposed using Dual-Force, by switching one branch outcome for Java on Android (box 1) and another for JavaScript on WebView (box 3).

Finally, Dual-Force produces an execution path annotated with the following information: 1) the switched branches (boxes 1 and 3), 2) the suppressed exceptions (none in this case), and 3) the values fed to the execution (none in this case). Also, the two-way communications between Java and JavaScript are also recorded, e.g., the invocations to *loadData* (box 2) and the JavaScript interfaces (box 4), as well as their arguments. Similar to existing techniques that expose hidden behaviors in malware analysis [8, 11], human domain knowledge is needed to determine if a specific execution is malicious. Essentially, the value of Dual-Force lies in producing a (hidden) behavior report for a (potentially malicious) app, which can hardly be generated by other approaches.

4 OVERVIEW

Fig. 4 illustrates the workflow of Dual-Force. Dual-Force takes an Android application package as the input, which is a zip file containing an Android manifest file, one or more DEX executables and other resources. To be analyzed, these contents are extracted first. Then Dual-Force performs a static analysis on the Android manifest file as well as DEX files, and instruments the DEX executables with forced execution semantics. Note that we add forced execution semantics to JavaScript by hacking Chromium, the back-end of WebView, instead of instrumention. One important reason is that JavaScript code is highly dynamic and it may only be known at runtime. After that, the static analysis results, instrumented DEX and JavaScript code are fed into the forced execution engine, which forces both Java and JavaScript code of WebView applications to execute along different paths to expose malwares.

4.1 Static Analysis

The static analysis engine aims to acquire four kinds of information for the forced execution engine, namely, entry points, control flow graphs (CFGs) and call graphs (CGs), values that can be statically determined, and locations where Java and JavaScript interact.

Entry points. We obtain all the registered entry points from *AndroidManifest.xml* where they are declared and then relate them to the corresponding classes.

CFGs and CGs. We first generate CFGs and CGs based on the extracted files from the package and then update them incrementally and iteratively with the dynamic results of the forced execution engine.

Values that can be statically determined. We also collect values that can be statically determined, such as string literals. Constant values, random values and other values of various types and formats (e.g., configuration files with an *.xml* extension and database files with *a .sqlite* extension) constitute a pool, whose values are supplied to the forced execution engine when values of specific types are required by the app to execute. The pool is also updated

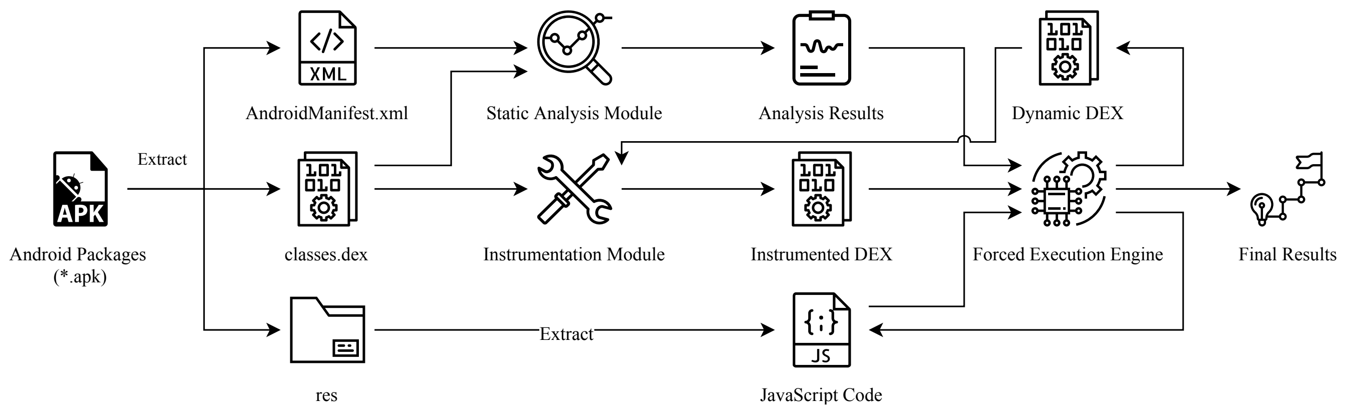


Figure 4: Overview of Dual-Force's workflow.

on-the-fly during execution, by dynamically intercepting the arguments of the invocations to a list of Java and Android APIs of interest, such as *String.equals* and *WebView.loadUrl*.

Locations where Java and JavaScript interoperate. For most cases, the values that are supplied to the engine are merely to allow the execution to proceed instead of driving the execution along different paths, except for strings. String values are special because they can be JavaScript code. If we feed arbitrary strings to WebView, we possibly cause fatal exceptions, not to mention exposing WebView related malicious payloads. And thus we use static analysis to find all the locations where an app is potentially asking for JavaScript code. Such locations include the following.

- Methods that are annotated with *@JavascriptInterface*.
- Statements that invoke methods of the class *WebView*, such as *addJavaScriptInterface*, *setJavascriptEnabled*, *loadUrl*, *loadData*, and so on.
- Statements that invoke string comparison methods whose arguments contain "http://", "file:///" or "javascript:".

These locations are identified for further use at runtime. When the forced execution engine finds that string types are required by an app to continue execution, it checks whether or not the app is asking for JavaScript code according to the analysis results on such locations. If so, we synthesize JavaScript code and feed it to the app. Otherwise, a normal string is sufficient.

4.2 Instrumentation

We instrument the DEX executables to achieve the following purposes at runtime.

Adding forced execution semantics. We add statements to log whether a branch is taken and the frequency of a branch being taken. Such information is used by the forced execution engine to decide what branches ought to be explored.

Injecting a top-level exception handler. Such a handler is used to record the exceptions raised at runtime, which helps the forced execution engine to supress them so that the execution can continue.

Monitoring dynamic class loading. Dynamic DEX executables are loaded by a set of class loaders. We instrument such loaders to obtain the dynamic DEX files and then additionally instrument new dynamic DEX files to understand their behaviors.

4.3 Forced Execution Engine

The forced execution engine shown in Fig. 4 is the key part of Dual-Force. It consists of two components:

- An execution model that forces WebView applications to execute along various paths of interest in a crash-free manner, which will be introduced in Section 5.
- A path exploration algorithm that steers forced execution to paths of interest, according to specific strategies, which will be introduced in Section 6.

The forced execution engine runs iteratively until an app is considered to be sufficiently explored by the path exploration algorithm, according to specific criteria. It forces both Java on Android and JavaScript on WebView to run. Particularly, it tries to make them actively interoperate, considering the nature of WebView malware.

5 CRASH-FREE EXECUTION MODEL

The essence of forced execution is an execution model that drives an app to execute along different paths, together with the ability to recover from exceptions and continue execution. The idea of forced execution is to make an app to execute along various paths forcefully by switching the branch outcomes. However, forced execution tends to raise exceptions as it may get into infeasible states. While pior works on forced execution have shown that such feasibility violations are in limited scale and do not incur problems in practice for malware analysis [8, 11, 13, 19], an execution model that can suppress WebView app related exceptions is critical. Dual-Force provides such a crash-free execution model from two aspects: the Android runtime and the WebView environment.

5.1 Android Runtime

The crash-free execution model on Android virtual machines mainly deals with Java unchecked exceptions which are not typically handled by the app. Checked exceptions are supposed to be dealt with by the exception handlers of the app and we let them remain what

they are. When an unchecked exception is thrown, the app is typically terminated by Android. We use a top-level exception handler to deal with the exceptions that are not caught by the app. Before the app is killed by the system, this handler attempts to collect information like the causes of the exceptions and the stack traces. When an exception is caught, Dual-Force analyzes the causes of the exception and tries to recover the execution. Note that, once the global exception handler catches an exception, the app has lost the control of the current execution. We recover the execution by restarting a new execution that is the same as the current one.

Algorithm 1 describes how we recover executions from exceptions. For a stack trace *ST* and the last switched branch *lsb*, we first get the locations where they are initially thrown at line 2, and try to suppress the exceptions based on the rules described in Table 1 (line 3). Then we restart the execution at line 4. But these actions do not guarantee that the exceptions are properly suppressed. Subsequent exceptions may follow in the following executions. We set a threshold *maxEx* to limit the max number of exceptions we deal with for one switched branch. Then we repeat the above steps until no exceptions are thrown or the threshold is reached. If more than *maxEx* exceptions are raised, we first locate the method invocation in *ST* that follows *lsb* in the same method (lines 7-11), which is the root method that causes too many exceptions. Then we replace it with a simple method which has the same return type as the original one (line 12). Table 2 shows the rules to construct such a method. Finally, the recovery is completed by restarting the execution at line 13.

Algorithm 1 Exception Recovery Algorithm

Inputs: *ST* - stack trace of the exception; *exec* - current execution; *lsb* - the last switched branch; *maxEx* - max number of exceptions we deal with for one branch switch

```
1:  while ST ≠ ∅ ∧ maxEx > 0 do
2:      loc ← code location of ST.pop()
3:      Patch the code at loc according to the rules in Table 1
4:      ST ← restart(exec)
5:      maxEx ← maxEx−1
6:  end while
7:  while ST ≠ ∅ do
8:      m ← null
9:      trace ← ST.pop()
10:     if trace follows lsb in the same method then
11:         m ← trace.method()
12:         Patch m according to the rules in Table 2
13:         restart(exec)
14:         break
15:     end if
16: end while
```

Table 1: Unchecked Exception Handling Rules for Java

Exception Type	Action
ArithmeticException	Replace the arithmetic computation with a random number
ArrayStoreException	Replace the instance to be stored with a constructed one
ClassCastException	Replace the instance to be casted with a constructed one
IllegalArgumentException	Replace the argument with a value of specific values
IndexOutOfBoundsException	Replace the index value with a small positive integral value
NegativeArraySizeException	Replace the negative with with its absolute value
NullPointerException	Replace the reference with a constructed instance
NumberFormatException	Replace the conversion with a random number

Table 1 shows how we deal with the common unchecked exceptions for Java at the locations where they are initially thrown. For example, we suppress number related exceptions by supplying numeric values, and for reference or type related exceptions, we suppress them by constructing objects of specific types. Table 2 shows the rules of generating simple methods to replace the original ones. For primitive types and normal string values, the method chooses among all the candidates and return one of them. For JavaScript types, we generate JavaScript code that calls Java methods annotated as JavaScript interfaces.

Table 2: Method Generation Rules for Java

Return Type	Generation
Numeric types	Randomized/Collected
Normal strings	Randomized/Collected
JavaScript strings	Synthesized
Non-recognizable types	Constructed via available constructors

The synthesis of JavaScript code is based on the results produced by static analysis. It works as follows. First, we identify all the Java methods that serve as JavaScript interfaces. Then, for a code location, we generate a piece of JavaScript code that invokes all the Java methods that are potentially callable from JavaScript. These include the methods that are statically annotated, as well as the ones that are dynamically annotated by calling the method *WebView.addJavaScriptInterface*. The argument values that are provided to the calls from JavaScript to Java are randomly generated. Note that we only synthesize JavaScript code that is used to trigger Java methods with the *@JavascriptInterface* annotation.

For types that are not recognized, we create an instance of it through one of its constructors. We preferentially choose default constructors and those whose parameters are primitive types or other recognized types. The arguments that are used to create the instance are randomly fed.

5.2 WebView

WebView is indeed a fully functional browser without the UI frame, which has many features including networking, rendering and running JavaScript. We address two challenges for forced execution on WebView in terms of JavaScript. One is to deal with web page related operations, whereas the other is to handle JavaScript exceptions.

In WebView, JavaScript code can manipulate web pages, such as accessing DOM (Document Object Model) objects and registering callbacks for events. The most common scenarios are to access the (DOM) elements and check if values of certain HTML input controls have the correct formats. For example, phishing pages that lure users to enter their credentials are likely to check if the text inputs for bank account numbers are correctly filled. JavaScript code may try to access a DOM element that does not exist. To deal with cases that JavaScript tries to access missing DOM elements, we first find all the available DOM elements in the current web page and put them into different categories, such as input controls, select controls and labels. When the access to a missing DOM element happens, we identify its category by checking what operations are done on it or what fields are to be fetched. Then we randomly select one object from all the objects in the category and replace the missing DOM element with this object. If there is no object in the

category, we generate a DOM element of its category and add it to the DOM tree of the web page.

JavaScript code contained in a web page often acts as callback functions that are only executed when specific events are triggered. For example, a click on a button triggers the callback function registered on the clicking event of the button. When WebView loads a web page containing several pieces of JavaScript code, most of them will not run until specific events are triggered. We force these JavaScript functions to run by calling them after the page is loaded. To be specific, they are called in the callback function corresponding to the *window.onload* event of the web page.

Another aspect of the forced execution on WebView is the ability to recover from JavaScript exceptions. The exceptions raised in WebView come from two places: first, the forced execution on JavaScript; second, the synthesis of JavaScript code. As mentioned before, sometimes we feed synthesized JavaScript code to WebView to make executions continue. Since the synthesis only takes syntaxes rather than semantics into consideration, JavaScript code can potentially contain exceptions. A JavaScript engine will throw an exception if an error occurs. For example, exceptions occur when a script or a function attempts to read a property that does not exist. Note that, JavaScript does not distinguish between exceptions and errors explicitly. They only differ in naming convention: errors are thrown by JavaScript engines while exceptions are thrown by developers. We do not distinguish these two terms in this paper.

All the JavaScript code fed to WebView is embedded in web pages, which may contain only JavaScript code. We register a top-level exception handler to the *window.onerror*, which reports the exception message, the script source, line and column numbers, as well as the error object. We then recover from the exception and continue the execution accordingly.

We handle JavaScript exceptions according to the rules shown in Table 3. The name of *SyntaxError* is self-explanatory. We replace the JavaScript code containing syntax errors with a synthesized one that simply calls the available JavaScript interfaces. *ReferenceError* occurs when an unknown variable is referenced or a right-hand-side value is assigned. To deal with this, we collect all the references in the JavaScript function and then replace the invalid reference with a random one from all the available references. If none exists, we create one for it. *RangeError* is handled by replacing the index with a smaller positive integral value. *TypeError* is coped with by replacing the object with another one in the same scope whose prototype has certain fields. If no such object exists, we create one for it. *URIError* indicates that there is something wrong with the URI value. We prepare a set of URI values to replace the problematic ones.

Table 3: Exception Handling Rules for JavaScript

Exception Type	Action
SyntaxError	Synthesize a new piece of JavaScript code
ReferenceError	Replace the reference with a reference candidate
RangeError	Replace index with a small positive integral value
TypeError	Replace type with a type candidate
URIError	Replace the URI values with one of prepared ones

6 PATH EXPLORATION

Dual-Force needs a path exploration algorithm that directs an app to execute towards specific parts of interest and to expose hidden behaviors. In this section, we introduce the path exploration algorithm and policies of Dual-Force.

6.1 Algorithm

Algorithm 2 describes a general worklist algorithm that produces new execution paths that are to be explored according to previous executions. The worklist stores a list of switches indicating which branch outcomes should be switched for path exploration. Note that Dual-Force only forcefully changes the branch outcome of a small set of predicate instances. It lets the other predicate instances remain untouched.

Algorithm 2 Path Exploration Algorithm

Inputs: *EP* - the set of entry points of a WebView app

Definitions: *switches* - a sequence of switched predicates by a forced execution, e.g., 1·3·5 means that the 1st, 3rd, and 5th predicates are switched

Ex - a set of pairs, where the first element of the pair is an entry point and the second element is a set of *switches* that has been executed

WL - a worklist of *switches* to be executed

exec - a concrete execution, denoted by a sequence of pairs that maps predicates to branch outcomes, e.g., (1,true)·(2,false)·(3,true) means that the execution has three predicates, the 1st takes true branch, the 2nd takes false branch, and 3rd takes true branch

```
1: for each entry ∈ EP do
2:     WL ← {nil}
3:     Ex.first ← entry
4:     Ex.second ← null
5:     while WL ≠ ∅ do
6:         switches ← WL.pop()
7:         Ex.second ← Ex.second ∪ switches
8:         exec ←forceExecute(entry,switches)
9:         t ← the last integer in switches
10:        exec ← remove the first t elements in exec
11:        for each (p,b) ∈ exec do
12:            if strategy(p,b) then
13:                WL ← WL ∪ switches · t
14:            end if
15:            t ← t + 1
16:        end for
17:    end while
18: end for
```

The input of this algorithm is the set of all the entry points of a WebView application. For each entry point *entry* of the app, the worklist is a singleton set with a null sequence representing an execution without forcebly switching any predicate (line 2). This means that when an entry point of an app is executed for the first time, the algorithm just allows the execution to proceed naturally. Line 8 is where forced execution is done by the function *forceExecute*. The app is forced to run from the current entry with switched branch outcomes. Then in lines 9-16, we try to determine if it would be of interest to further switch more predicate instances. Lines 9-10 compute the sequence of predicate instances eligible for switching. Note that it cannot be a predicate before the last switched predicate specified in switches as switching such a predicate may

change the control flow such that the specification in switches becomes invalid. In lines 11-16, for each eligible predicate and its current branch outcome, we query the function *strategy* which will be introduced in Subsection 6.2, to determine if we should further switch it to generate a new forced execution. If so, we add it to the worklist. Note that in each new forced execution, we essentially switch one more predicate. The algorithm is terminated there is no more paths execute, which indicates that there is no space of interest to explore.

6.2 Exploration Strategies

There exist many exploration strategies, ranging from simple ones based on instruction coverage and sophisticated ones based on evolutionary algorithms. However, each of them has specific advantages over others for different scenarios. A good and suitable exploration strategy is crucial for inspecting the target app. Here we introduce three strategies we adopted for Dual-Force.

- *Branch-coverage-based exploration.* This exploration strategy is based on the intuition that the less a branch is executed, the more likely an uncovered payload is located in it. Actually, this strategy proves to be effective in test case generations through symbolic execution [17]. In this strategy, forced execution is preferentially steered to less traveled branches.
- *Cross-language-interoperation-directed exploration.* This strategy regards the cross-language interoperability nature of WebView malware as the main factor that would lead to deeply concealed payloads. In this strategy, forced execution on Java is preferentially steered to locations that contain WebView related operations and vice versa.
- *Hybrid exploration.* This strategy takes both branch coverage and cross-language interoperation between Java and JavaScript into consideration.

7 IMPLEMENTATION AND EVALUATION

We implement the forced execution model for Android using Soot [25], FlowDroid [7] and Xposed [5]. Soot is used to instrument the DEX executables of the application to switch branch outcomes, record execution paths, add top-level exception handlers, and monitor dynamic DEX loading. FlowDroid is used to perform static analysis and compute the CGs and CFGs of the apps. Xposed is used to dynamically intercept method calls in Java. For example, we can use it to invoke a simpler method instead of a complicated one to suppress exceptions.

In addition, we enable forced execution on WebView by hacking Chromium, the back-end of WebView on Android L (5.0) and later versions, where WebView has moved to an APK so it can be updated separately to the Android platform. We modify the JavaScript engine to make it forcefully switch branch outcomes as needed. We also deal with all the problems that may be raised by forced execution in WebView, e.g. exception handling.

We evaluate the effectiveness of Dual-Force by applying it to 150 Android malware samples. Our evaluation tries to answer the following research questions.

- How effective is Dual-Force at exposing malicious payloads in WebView applications, compared with other approaches?
- How efficient is Dual-Force?

7.1 Experimental Setup

We randomly collected 150 WebView malware samples from online malware databases including VirusTotal [2], Koodous [3] and Contagio mobile mini dump [4]. We conduct the experiments on a PC with an Intel Core i7-4790 (3.6G Hz) CPU and 16 GB RAM. The PC runs an emulator on which these Android malware samples are evaluated.

7.2 Effectiveness

Table 4 shows basic results of Dual-Force. Only 21.3% of the WebView malwares show their payloads when they are launched by starting their main activities. More malwares (46.7% of them) expose their malicious payloads by starting the main activities and triggering random events. With Dual-Force, we can expose at least one harmful behavior in 119 out of 150 (79.3%) WebView malwares. This table also shows the minimum, average, and maximum results for one app. On average, 4.8 malicious behaviors are exposed taking 35.8 executions for one app. Dual-Force exposes at most 16 malicious behaviors and takes up to 98 executions to run an app. Note that human domain knowledge is needed here to determine if a specific execution is malicious. It is possible to use other approaches (e.g. rule-based filters and data mining) to help domain experts to simplify this process, which is not the focus of this work though.

Table 4: Basic Results

	Approach		
	Launch	Launch & Trigger	Dual-Force
Apps with $\geq$ 1 payload exposed	21.3%	46.7%	79.3%
Results per app (min/avg/max)			
- # of malicious behaviors	0/0.6/1	0/1.2/5	0/4.8/16
- Executions	1/3.1/7	3/13.1/22	9/35.8/98

As Fig. 5 illustrates, among all the 150 WebView malwares, 121 of them call JavaScript from Java and 78 of them invoke Java from JavaScript. 66 samples uses the two-way communications between Java and JavaScript. 133 samples of them use at least one way communications. Only 17 of them use WebView to display pure HTML pages without JavaScript.

Table 5 shows that the JavaScript interfaces callable from JavaScript code are typically sensitive APIs. Many of the Android system functionalities, such as sending SMS messages and getting device information are exposed to JavaScript code, making sensitive information easy to be leaked. This also makes it more difficult to understand how malicious payloads in WebView malware work. Of all the JavaScript interfaces, only 31.3% are called by other Java methods, and the rest 68.7% are only called by JavaScript code. It indicates that if we merely look at the call graphs of Java code, there are no ingoing edges to the methods that are only called by JavaScript. As such, static analysis is very likely to miss a lot of malicious behaviors.

We also compare the analysis results of Dual-Force with two malware databases: VirusTotal and Koodous, both of which are

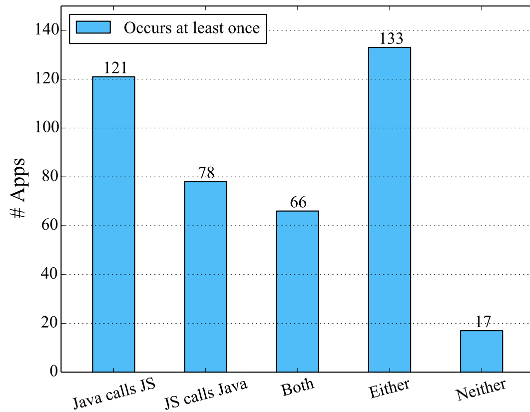


Figure 5: Cross-language interoperation of the samples.

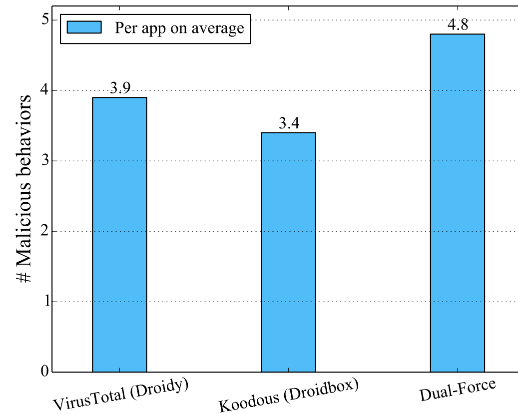


Figure 6: Overall comparison with Droidy and Droidbox.

Table 5: Top 10 Java functionalities called by JavaScript

Functionality	Occurrence
Sending SMS messages	70.7%
Getting IMEI	68.7%
Getting device name	59.3%
Getting phone number	56.0%
Intercepting SMS messages	51.3%
Reading private files	50.0%
Getting SDK versions	46.7%
Obtaining installed packages	45.3%
Running JavaScript	44.0%
Judging if running on an emulator	41.3%

equipped with multiple detection engines and behavior analysis functionalities: *Droidy* for VirusTotal and *Droidbox* for Koodous.

The overall comparison is shown in Fig. 6. On average, VirusTotal, Koodous, and Dual-Force expose 3.9, 3.3, and 4.8 malicious payloads respectively. Dual-Force exposes roughly 23% more malicious payloads than VirusTotal and 41% more than Koodous. Note that, this result is calculated on 150 WebView samples instead of 119 ones that contains at least one malicious behavior. The results shows clearly that Dual-Force has prominent advantages at exposing behaviors of WebView malwares.

Among all the undiscovered maliciousness by Droidy and Droidbox, most of them are WebView related. They are complicated by the interaction between Android and WebView, two totally different running environments. Existing approaches can hardly trigger carefully designed and deeply concealed actions. Even such an action is triggered, these approaches cannot obtain a complete execution trace for the action because it cannot handle the interoperation between Java and JavaScript. Consequently, they usually fail to find the attacks and expose the logic behind the attacks, thus making such payloads evade analysis and detection.

Fig. 7 shows the effectiveness of three different exploration strategies introduced in Subsection 6.2. We can see that branch-coverage-based strategy is overall better the cross-language-interoperation-directed one. However, the latter is better at discovering WebView-related payloads, which is intuitive because it is designed to be so. Hybrid exploration strategy that takes both into consideration is

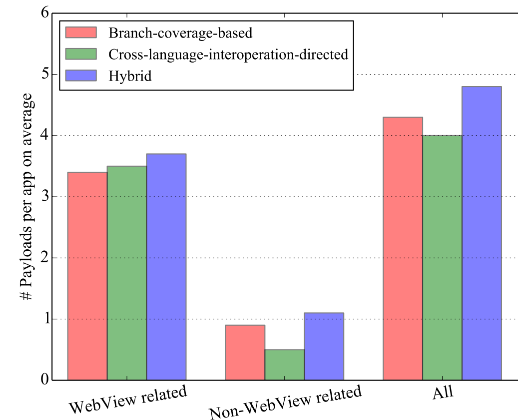


Figure 7: Comparison among three exploration strategies.

a optimal choice here in terms of exposing payloads in Android WebView malware.

In Table 6, we list the top 15 WebView malware samples with the most malicious behaviors exposed by Dual-Force and we sort them in descending order. The columns named *Detections* represent the number of engines that detect at least one malicious behavior in the app. The column *Droidy* shows the number of malicious behaviors of an app exposed by VirusTotal via Droidy. The column *Droidbox* represents the number of malicious behaviors of an app exposed by Koodous via Droidbox. Dual-Force exposes at most 16 malicious behaviors in an app, while Droidy and Droidbox expose at most 7 and 8 respectively. Note that we have four false positives for these 15 samples together. For all the 150 WebView samples, the average false positive rate is 5.1%, which is marginal.

Table 6: Comparison with VirusTotal and Koodous

MD5	Package Name	VirusTotal		Koodous		Dual-Force	
		Detections	**Droidy**	Detections	**Droidbox**	**Exposure**	F/P
cbd506003ce1a4f8cc656f6614baf775	com.vivchar.TheBookofLoveLWP	15	7	1	4	16	0
3a1c2626158acc4a55d06246a669d1e5	candy.crush.saga.unlimitedf32f	33	7	2	6	16	0
4dc7e82047a92403a23c2e6c3c3eb4bd	com.aio.downloader	21	7	5	8	15	2
abda3e50bc31f5eb16e39a72bfcc9886	com.androidsky.app.tusiji	25	0	1	6	14	0
b0c41093dc33dc81674aeb92140ad923	com.ptcc.app	19	5	6	5	14	0
85506f0b70ea01eb3b7a9a42a183375c	com.udhay.indianrecipes	15	7	1	0	13	0
f52b9985233b9c1825ef13ad60d89298	lunar.horror.view	15	7	1	2	13	1
22b097f7dfedf75e0a1f5f0e148adbed	com.mobile.shuangjielong2	25	0	2	6	12	0
02e231f85558f37da6802142440736f6	krep.itmtd.ywtjexf	40	6	0	4	12	0
a2cf71cf18e860584429a5d84365e2a9	air.TheModifuckrs.ersite.ru	22	6	0	0	11	0

7.3 Efficiency

We show efficiency related data of Dual-Force in Table 7. It takes an averge of 198.3 seconds for Dual-Force to force-execute a WebView app, while the minimum and maximum numbers are 30.2 seconds and 544.9 seconds respectively. During the execution, Dual-Force switches 4.2 and 1.2 branch outcomes on average for Java and JavaScript respectively. It switches more predicates for Java because Java code is usually more complicated than JavaScript code. The exceptions suppressed (2.6 and 0.7 on average respectively) and the values fed to the apps (4.1 and 1.1 on average respectively) are *almost* linear to numbers of predicates that are switched, as we can see in this table.

Table 7: Statistics on Dual-Force's efficiency

Statistics	min	avg	max
Predicates switched	1	5.4	15
- Java	1	4.2	11
- JavaScript	0	1.2	5
Exceptions suppressed	1	3.3	6
- Java	1	2.6	5
- JavaScript	0	0.7	2
Values fed	1	5.2	10
- Java	1	4.1	9
- JavaScript	0	1.1	2
Time (s)	30.2	198.3	544.9

8 DISCUSSION AND RELATED WORK

Forced execution was first proposed in X-Force [19], which was originally designed for dynamic binary analysis. iRiS [8] adopted the technique to iteratively compute the call graphs and control flow graphs of iOS apps to discover private API abuse which is forbidden by Apple. J-Force [13] and JSForce [11] are two forced execution engines that work on JavaScript.

There exist dynamic analysis techniques developed to expose malicious payloads of Android apps. Grodddroid [6] uses an algorithm that automatically identifies potentially malicious code and stimulates the GUI of an application and forces the execution of some branching conditions if needed. It is similar to our work in terms of forced execution, but our work do not need to identify potentially malicious code first. Malton [28] conducts multi-layer monitoring and information flow tracking to provide a comprehensive view of malicious behaviors of Andriod apps. CooperDroid [24] monitors malware behaviors mainly through the trace of system calls. FuzzDroid [21] proposes a targeted fuzzing framework that uses multiple analyses to generate environments that trigger specific behaviors. Harvester [20] collects runtime values that can enhance other dynamic analysis. IntelliDroid [26] is conceptually similar to FuzzDroid except the fact that it does not use multiple analyses. DroidTrace [30] monitor selected system calls of the target process which is running the dynamic payloads, and classifies the payloads behaviors through the system call sequence. Droidbox [14] is an android application sandbox for dynamic analysis. AppsPalyground [23] is a framework for automated dynamic security analysis of Android applications. DroidScope [29] is a virtualization-based malware analysis. EvoDroid [18] uses evolutionary testing for Android apps. GoldenEye [27] switches the analysis environment at runtime through a specially designed speculative execution engine. HybriDroid [15] is static analysis that deals with Android hybrid applications.

Dual-Force is different from existing dynamic analyses in two ways. First, from the technical perspective, Dual-Force develops a novel crash-free forced execution model in terms of exposing maliciousness in Android apps. Second, it targets a unique and yet increasingly prominent category of malwares, i.e. Android WebView apps. Such malwares can hardly be handled effectively by existing approaches considering their unique natures, such as the two-way communications between Java and JavaScript.

9 CONCLUSION

We propose in this paper a forced execution technique called Dual-Force to expose malicious payloads for Android WebView malware. We develop a crash-free forced execution model that can recover from exceptions properly for WebView apps.The experimental results demonstrate that Dual-Force can expose potentially harmful behaviors for 119 out of 150 malicious apps. Compared to the state-of-the-art, Dual-Force can expose 23% more malicious behaviors per app on average.

ACKNOWLEDGMENT

This work is partially supported by National Natural Science Foundation (Grant No. 61502228) of China.

REFERENCES

[1] [n. d.]. https://developer.android.com/reference/android/webkit/WebView.
[2] [n. d.]. https://www.virustotal.com.
[3] [n. d.]. https://koodous.com.
[4] [n. d.]. http://contagiominidump.blogspot.com.
[5] [n. d.]. Xposed. http://xposed.info.
[6] Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, J-F Lalande, and V Viet Triem Tong. 2015. GroddDroid: a gorilla for triggering malicious

behaviors. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*. IEEE, 119–127.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[8] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 44–56.

[9] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*. Springer, 291–307.

[10] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe.. In *NDSS*. Citeseer.

[11] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2017. JSForce: A Forced Execution Engine for Malicious JavaScript Detection. *arXiv preprint arXiv:1701.07860* (2017).

[12] Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou. 2013. Behavioral analysis of android applications using automated instrumentation. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 182–187.

[13] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 897–906.

[14] P Lantz, A Desnos, and K Yang. 2012. DroidBox: Android application sandbox.

[15] S. Lee, J. Dolby, and S. Ryu. 2016. HybriDroid: Static analysis framework for Android hybrid applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 250–261.

[16] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 280–291.

[17] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths, See [17], 19–32. https://doi.org/10.1145/2509136.2509553

[18] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.

[19] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications.. In *USENIX Security Symposium*. 829–844.

[20] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.

[21] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 300–311.

[22] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. 2015. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*. Springer, 187–202.

[23] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 209–220.

[24] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *NDSS*.

[25] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[26] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

[27] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2014. Goldeneye: efficiently and effectively unveiling malware's targeted environment. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 22–45.

[28] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *In 26th USENIX Security Symposium (USENIX Security 17). ACM.*

[29] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *USENIX security symposium*. 569–584.

[30] Min Zheng, Mingshen Sun, and John CS Lui. 2014. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*. IEEE, 128–133.