Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues

Xiaoyu Sun
xiaoyu.sun@monash.edu
Monash University
Australia, Clayton, VIC

Xiao Chen
Xiao.chen@monash.edu
Monash University
Australia, Clayton, VIC

Yanjie Zhao
Yanjie.Zhao@monash.edu
Monash University
Australia, Clayton, VIC

Pei Liu
Pei.Liu@monash.edu
Monash University
Australia, Clayton, VIC

John Grundy
john.grundy@monash.edu
Monash University
Australia, Clayton, VIC

Li Li*
li.li@monash.edu
Monash University
Australia, Clayton, VIC

ABSTRACT

Despite being one of the largest and most popular projects, the official Android framework has only provided test cases for less than 30% of its APIs. Such a poor test case coverage rate has led to many compatibility issues that can cause apps to crash at runtime on specific Android devices, resulting in poor user experiences for both apps and the Android ecosystem. To mitigate this impact, various approaches have been proposed to automatically detect such compatibility issues. Unfortunately, these approaches have only focused on detecting signature-induced compatibility issues (i.e., a certain API does not exist in certain Android versions), leaving other equally important types of compatibility issues unresolved. In this work, we propose a novel prototype tool, JUnitTestGen, to fill this gap by mining existing Android API usage to generate unit test cases. After locating Android API usage in given real-world Android apps, JUnitTestGen performs inter-procedural backward data-flow analysis to generate a minimal executable code snippet (i.e., test case). Experimental results on thousands of real-world Android apps show that JUnitTestGen is effective in generating valid unit test cases for Android APIs. We show that these generated test cases are indeed helpful for pinpointing compatibility issues, including ones involving semantic code changes.

ACM Reference Format:
Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. 2022. Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues . In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3561151

1 INTRODUCTION

Unit testing is a type of software testing aiming at testing the effectiveness of a software's units, such as functions or methods. It is often the first level of testing conducted by software developers themselves (hence a white box testing technique) to ensure that their code is correctly implemented. Unit testing has many advantages. First, it helps developers fix bugs early in the development cycle which subsequently saves costs in the end. Indeed, it is known that the cost of a bug increases exponentially with time in the software development workflow. Second, it makes it possible to achieve regression testing. For example, if developers refactor their code later, it allows them to make sure that the refactored code still works correctly. Third, unit tests provide an effective means for helping developers understand the unit under testing, i.e., what functions are provided by the unit and how to use them? Because of the aforementioned advantages, it is recommended to always write unit tests when developing software, and the unit tests should cover as many units as possible.

The Android framework, as one of the largest software projects (with over 500,000 commits), is no exception. The Android framework provides thousands of public APIs that are heavily leveraged by app developers to facilitate their development of Android apps. Ideally, each such public API should be provided with a set of unit tests to ensure that the API is correctly implemented and the continuous evolution of the framework will not change its semantics. Unfortunately, based on our preliminary investigation, less than 30% of APIs, are provided with unit test cases, leaving the majority of APIs uncovered. This is unacceptable considering that the Android framework nowadays has become one of the most popular projects (with millions of devices running it).

This poor test coverage of Android APIs has led to serious compatibility issues in the Android ecosystem, as recently shown [10, 11, 22, 31, 47, 53]. For example, Li et al. [22] demonstrate that various Android APIs suffer from compatibility issues as the evolution of the Android framework will regularly remove APIs from or add APIs into the framework. Such API removal or addition can result in no such class or method runtime exceptions when the corresponding app is running on certain framework versions. Liu et al. [26] further present an approach to detect silently-evolved Android APIs, which could cause another type of compatibility issue as their semantics are altered (while not explicitly documented) due to the evolution of the Android framework. Moreover, Wei et al. [48] experimentally show that some Android APIs could even be customized by smartphone manufacturers, leading to another type of compatibility issue that causes Android apps to crash on

*Li Li is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3561151

certain devices while behaving normally on others. The authors further propose a prototype tool called PIVOT to automatically learn device-specific compatibility issues from existing Android apps. Their experiments on a set of top-ranked Google Play apps have discovered 17 device-specific compatibility issues.

To the best of our knowledge, the state-of-the-art works targeting compatibility issue detection leverage static analysis techniques to achieve their purpose. However, as known to the community, the static analysis will likely yield false positive results as it has to make some trade-offs when handling complicated cases (e.g., object-sensitive vs. object-insensitive). In addition, the static analysis will also likely suffer from soundness issues because some complicated features (e.g., reflection, obfuscation, and hardening) are difficult to be handled [38, 44]. Furthermore, except for syntactic changes, compatibility issues could also be triggered by semantic changes, which are non-trivial to be handled statically. Indeed, as demonstrated by Liu et al. [26], there are various semantic change-induced compatibility issues in the Android ecosystem that remain undetected after various static compatibility issue detection approaches are proposed to the community.

Moreover, static app analysis can only be leveraged to perform post-momentum analysis (i.e., after the compatibility issues are introduced to the community). They cannot stop the problems from being distributed into the community – many Android apps, including very popular ones, still suffer from compatibility issues. To mitigate this problem, we argue that incompatible Android APIs should be addressed as early as possible, i.e., ideally, at the time when they are introduced to the framework. This could be achieved by providing unit tests for every API introduced to the framework and regressively testing the APIs against Android devices with different manufacturers and different framework versions. However, it is time-consuming to manually write and maintain unit tests for each Android API (which probably explains why there is only a small set of APIs covered by unit tests at the moment). There is hence a need to automatically generate compatibility unit tests for Android APIs.

In this work, we present a prototype tool, JUnitTestGen, that attempts to automatically generate test cases for Android APIs based on their practical usage in real-world apps. Specifically, after locating existing API usages in real-world Android apps, JUnitTestGen performs field-aware, inter-procedural backward data-flow analysis to infer the API caller instance and its parameter values. JUnitTestGen then leverages the inferred information to reconstruct a minimal executable code snippet for the API under testing. Experimental results on thousands of Android apps show that JUnitTestGen is effective in generating test cases for Android APIs. It achieves an 80.4% of success rate in generating valid test cases. These test cases subsequently allow our approach to pinpoint various types of compatibility issues, outperforming a state-of-the-art generic test generation tool named EvoSuite, which can only generate test cases to reveal a small subset of compatibility issues. Furthermore, we demonstrate the usefulness of JUnitTestGen by comparing it against a state-of-the-art static analysis-based compatibility issue detector called CiD. JUnitTestGen is able to mitigate CiD's false-positive results and go beyond CiD's capability (i.e., detecting compatibility issues induced by APIs' signature changes) to detect compatibility issues induced by APIs' semantic changes.

Overall, we make the following main contributions in this work:

- We have designed and implemented a prototype tool JUnitTestGen that leverages a novel approach to automatically generate unit test cases for APIs based on their existing usages.
- We have set up a reusable testing framework for pinpointing API-induced compatibility issues by automatically executing a large set of unit test cases on multiple Android devices.
- We have demonstrated the effectiveness of JUnitTestGen by i) generating valid test cases for Android APIs and pinpointing problematic APIs that could induce compatibility issues if accessed by Android apps, ii) outperforming state-of-the-art tools on real-world apps in detecting a wider range of compatibility issues.

The source code[1] and experimental results are all made publicly available in our artifact package.[2].

2 MOTIVATION

To overcome the fragmentation problem, our fellow researchers have proposed various approaches to mitigate the usage of compatibility issues in Android apps [47, 48, 50]. These approaches mainly leverage static analysis to achieve their purpose. Unfortunately, static analysis is known to likely generate false-positive and false-negative results and is yet hard to handle such issues that involve semantics changes in Android APIs [26]. Therefore, we argue that there is a need also to invent dynamic testing approaches to complement existing works in handling app compatibility issues.

We hence start by conducting a preliminary study investigating the test case coverage in the Android framework. Specifically, we downloaded the source code of AOSP from API level 21 to 30 and then calculated the number of public APIs[3] and their corresponding unit test cases provided by Google. Our result reveals that on average **less than 30% of Android framework APIs have provided test cases in each API level**, indicating the Android framework has a poor test case coverage. When more APIs are provided with unit test cases, more compatibility issues of APIs will likely be identified during regression testing. This will enable them to be fixed at an earlier stage to avoid the introduction of compatibility issues in the first place. To this end, we propose to effectively and efficiently detect compatibility issues through a dynamic testing approach that fulfills its objective by automatically generating valid test cases by mining API usages from real-world Android apps.

Why Dynamic Testing. We now present a concrete example to motivate why there is a need to generate more and better unit test cases for Android APIs to pinpoint compatibility issues. There is an Android API called *getNotificationPolicy()*, located in class *NotificationManager*. At the moment, there are no unit tests provided for this API. The lack of solid testing for this API has unfortunately led to various problems, as demonstrated by the various discussions on StackOverflow [41], one of the most widely used question and answer websites.

Figure 1 presents the brief evolution of the source code of *getNotificationPolicy()*. This API was introduced to the Android framework at API level 23. The apps that accessed this API would crash on devices powered by Android with API level 22 or earlier, resulting

[1] https://github.com/SMAT-Lab/JUnitTestGen
[2] https://doi.org/10.5281/zenodo.6507579
[3] The APIs in `platform/frameworks/base` path.

API 23 (getNotificationPolicy Added at this version)

```
/**
* Gets the current notification policy.
*
* <p>
* Only available if policy access is granted to this package.
* See {@link #isNotificationPolicyAccessGranted}.
*/
public Policy getNotificationPolicy() {
    INotificationManager service = getService();
    try {
        return
service.getNotificationPolicy(mContext.getOpPackageName());
    } catch (RemoteException e) {
    }
    return null;
}
```

API 24-27

```
/**
* Gets the current notification policy.
*
* <p>
* Only available if policy access is granted to this package.
* See {@link #isNotificationPolicyAccessGranted}.
*/
public Policy getNotificationPolicy() {
    INotificationManager service = getService();
    try {
        return
service.getNotificationPolicy(mContext.getOpPackageName());
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

API 28 latest

```
/**
* Gets the current user-specified default notification policy.
*
* <p>
*/
public Policy getNotificationPolicy() {
    INotificationManager service = getService();
    try {
        return
service.getNotificationPolicy(mContext.getOpPackageName());
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

Figure 1: The source code of Android API *getNotificationPolicy()*.

in forwarding compatibility issues. During the evolution of the Android framework, the implementation of *getNotificationPolicy()* is quickly changed at API level 24 (i.e., no longer returns null). Nevertheless, at this point, the comments are not changed, suggesting potential compatibility issues because of changed semantics. At API level 28, this API is further changed. This time, only the comments are changed, i.e., the implementation of the API is kept the same. This change further suggests that this API may be involved with compatibility issues as the source code of the API and its comments are inconsistent at a certain period (over six API levels).

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3  super.onCreate(savedInstanceState);
4  setContentView(R.layout.activity_main);
5
6  NotificationManager mng = (NotificationManager)
       this.getSystemService("notification");
7  NotificationManager.Policy policy =
       mng.getNotificationPolicy();
8  int priorityCallSenders = policy.priorityCallSenders;
9 }
```

Listing 1: Code example of invoking API *getNotificationPolicy()*.

The actual implementation of *getNotificationPolicy()* is through the complicated inter-process communication mechanism (e.g., the API is defined via Android Interface Definition Language (AIDL). Thus, it is non-trivial to confirm if there is a compatibility issue after API level 23 by only (statically) looking at the Java code of the framework. It is still a known challenge to statically analyze Java and C code at the same time.

We resort to a dynamic approach to check if *getNotificationPolicy()* suffers from compatibility issues. Specifically, we implement a simple Android app with minimal lines of code to invoke the API (i.e., lines 6-7 in Listing 1) and also the usage of the return value of this API(i.e., lines 8 in Listing 1). The minimal and targeted SDKs of this app are set to be 21 and 30, respectively. We then launch this app on ten emulators covering Android API levels from 21 to 30. As expected, the app throws *NoSuchMethodError* as the API is not yet introduced at API level 21 and 22. From API level 23 to 27, the app throws *SecurityException* with the message "Notification policy access denied", due to a lack of declaration of Android permissions. Even though the permission was granted, the API can still introduce compatibility issues at API level 23 because the return value can be null, causing *NullPointerException* later on (e.g., lines 8 in Listing 1)). Surprisingly, since API level 28, the app does not throw any exception, even though no permissions are declared as well. We then go one step further to track the detailed implementation of *service.getNotificationPolicy(String)* and found that the enforcement of policy access (i.e., line 3 in Listing 2) is removed in the API at API level 28, which explains why the app no longer crashes since API level 28.

```
1  @Override
2  public Policy getNotificationPolicy(String pkg) {
3 -  enforcePolicyAccess(pkg, "getNotificationPolicy");
4   final long identity = Binder.clearCallingIdentity();
5   try {
6    return mZenModeHelper.getNotificationPolicy();
7   } finally {
8    Binder.restoreCallingIdentity(identity);
9  }}
```

Listing 2: Code changes of method *getNotificationPolicy(String)*, which includes the underline implementation of Android API *getNotificationPolicy()*.

These observed runtime behaviours strongly indicate that the direct invocation of *getNotificationPolicy()* will very likely result in two compatibility issues: (1) a method is not yet defined and (2) method semantics have been altered. Ideally, such issues – especially the latter case – should not be introduced into the Android framework. However, due to the lack of API unit tests, such issues are non-trivial to identify and avoid. It is hence essential to provide more and better unit test cases for all Android APIs. Since it is time-consuming to achieve this manually, we argue that there is a strong need to provide automated approaches to automatically generate unit test cases for Android APIs to identify compatibility issues as early as possible. In this work, we propose to generate unit test cases for Android APIs by learning from existing Android API usages.

Why mining API usage. In addition, generic test generation tools [1, 7, 35] mainly targeting on satisfying coverage criteria for classes, while compatibility issues are mainly caused by the fast-evolving of APIs [22], which makes them insufficient in detecting compatibility issues. Specifically, such generic test case generation approaches (such as EvoSuite) are tailored to generate tests based on the source code of classes, lacking API usage knowledge [15], including both API calling context and API dependency knowledge. This information is crucial to setting up the environment for successfully calling Android APIs properly to detect compatibility issues. In addition, all of these tools completely ignore semantic-level behaviours at the API level, leading to many compatibility issues undetected. Especially in Android, APIs often come with usage caveats, such as constraints on call order [37]. Thus, it is essential to capture API dependencies [51] involved in the calling context before invoking the target API. However, it is challenging for traditional test generation tools to achieve this since they generate test suites only based on source code, which lacks API dependency knowledge. Thus, the insufficiency of the generic coverage-based test case generation approach motivates us to mine API usage to

generate much more effective test cases in detecting compatibility issues.

3 OUR APPROACH

The main goal of this work is to automatically generate unit tests for the Android framework to provide better coverage at the unit test stage of as many Android APIs as possible. Fig. 2 outlines the process of JUnitTestGen, which is made up of two modules involving a total of nine steps. We first locate target API invocations after disassembling the APK bytecode. We then apply inter-procedural data-flow analysis to identify the API usage, including API caller instance inference and API parameter value inference. We then execute these generated test cases on Android devices with different Android versions (i.e., API levels). The following elaborates on the detailed process of each module.

The first module, *Automated API Unit Test Generation*, includes five sub-processes to automatically generate unit test cases for Android APIs. This module takes as input an Android API (or an API list) for which we want to generate unit tests and an Android app that invokes the API and outputs a minimum executable code snippet involving the given API. This code snippet is the API's unit test case.

The second module, *App Compatibility Testing*, includes two sub-processes. This module takes as input the previously generated unit test cases to build an Android app, allowing direct executions of the test cases on Android devices running different framework versions (i.e., API levels). The output of this second module is the execution results of the test cases concerning different execution environments. Using this, JUnitTestGen can then determine all the Android APIs suffering from potential compatibility issues. We now detail these two modules of JUnitTestGen below.

3.1 Automated API Unit Test Generation

As shown in Figure 2, the first module of JUnitTestGen, namely Automated API Unit Test Generation, is made up of five steps.

I.1 Android Bytecode Disassembling. JUnitTestGen takes Android APKs as input. Given an Android app, the first step of JUnitTestGen is to disassemble the Dalvik bytecode to an intermediate representation. In this work, we leverage Soot [45], which provides precise features for code analysis. In particular, it supports 3-address code intermediate representation Jimple and accurate call-graph analysis framework Spark[17]. On top of Soot, JUnitTestGen is able to convert Android APK's bytecode into Jimple precisely.

I.2 Android API Locating. The second step towards locating the practical usages of Android APIs is quite straightforward. JUnitTestGen visits every method of each application in its Jimple representation, i.e., statement by statement, to check if any of the APIs in the input list is invoked. If so, we record the location and mark the usage of the API as located.

I.3 API Caller Instance Inference. Android APIs are invoked as either *static methods* (also known as *class methods*) or *instance methods*. A static method can be called without needing an object of the class, while an instance method requires an object of its class to be created before it can be called. Listing 3 shows an example for each method type in the Jimple representation. The static method (*boolean equals()*) can be called directly as long as the API signature (*<android.text.TextUtils: boolean equals(java.lang.CharSequence, java.lang.CharSequence)>*) is acquired, which has already been done in step I.2. However, to invoke the instance method (*void setLayoutParams()*), its calling object *$r5* needs to be identified. Specifically, a backward data-flow analysis is needed to locate both the definition statement of *$r5* and any intermediate APIs (or methods) on the call trace that may change the behaviour of the caller instance.

```
1 //Calling a Static API
2 <android.text.TextUtils: boolean
      equals(java.lang.CharSequence,java.lang.CharSequence)>
3
4 //Calling an Instance API
5 $r5.<android.widget.ListView: void
      setLayoutParams(android.view.ViewGroup$LayoutParams)>
```

Listing 3: Examples of calling static and instance APIs.

Identifying the calling object and constructing the call trace is a non-trivial task. As the calling object's instantiation could depend on multiple method calls (e.g., can be defined in other methods and passed on via callee's returned values), the caller instance inference process needs to be inter-procedural. Furthermore, each of the involved methods in the instantiation process may further require specific parameter values, which need to be properly prepared in order to successfully create the calling object. Hence, the caller instance inference process requires backtracing not only the direct caller instance but also many other variables leveraged by the app to instantiate the calling object.

Algorithm 1 API Caller Instance Inference.

Require: api_t: the target API
Ensure: M: a list of method invocations towards api_t
function CALCULATEMINIMUMEXECUTABLECONTEXT(api_t)
  $M = \emptyset$
  $stmt_t \leftarrow$ statement contains api_t
  if api_t is a static API **then**
    $M \leftarrow$ The API signature of api_t
  end if
  if api_t is an instance API **then**
    $assignStmt \leftarrow$ GETDEFINITIONSTMT($stmt_t$)
    $M \leftarrow$ CONSTRUCTCALLTRACE($assignStmt, \emptyset$)
  end if
  return M
end function

Algorithm 2 Construct Call Trace.

Require: $stmt_{co}$: A definition statement of the calling object
  $visitedStmts$: The set of statements visited by the analysis.
Ensure: $callTrace$: a list of method invocations towards $stmt_{co}$
function CONSTRUCTCALLTRACE($stmt_{co}, visitedStmts$)
  $callTrace \leftarrow \emptyset$
  if ($stmt_{co}$ in $visitedStmts$) OR ($stmt_{co}$ is Constant) OR ($stmt_{co}$ is Syetem API) **then**
    return $callTrace$
  end if
  $visitedStmts \leftarrow visitedStmts \cup stmt_{co}$
  if $stmt_{co}$ is ParameterRef **then**
    $stmt_{inv} \leftarrow getInvolvingStmtsfromCallGraph(stmt_{co})$
    for *each* $s \in stmt_{inv}$ **do**
      $definitionStmt_{param} \leftarrow$ GETDEFINITIONSTMT(s)
      $callTrace \leftarrow$ CONSTRUCTCALLTRACE($definitionStmt_{param}, visitedStmts$)
    end for
  end if
  if $stmt_{co}$ is InvokeExpr **then**
    $stmt_{inv} \leftarrow getInvolvingStmtsfromCallGraph(stmt_{co})$
    for *each* $s \in stmt_{inv}$ **do**
      $method_{host} \leftarrow$ the host method of s
      $returnStmt \leftarrow getReturnStmtFromMethod(method_{host})$
      $definitionStmt_{returnStmt} \leftarrow$ GETDEFINITIONSTMT($returnStmt$)
      $callTrace \leftarrow$ CONSTRUCTCALLTRACE($definitionStmt_{returnStmt}, visitedStmts$)
    end for
  end if
end function

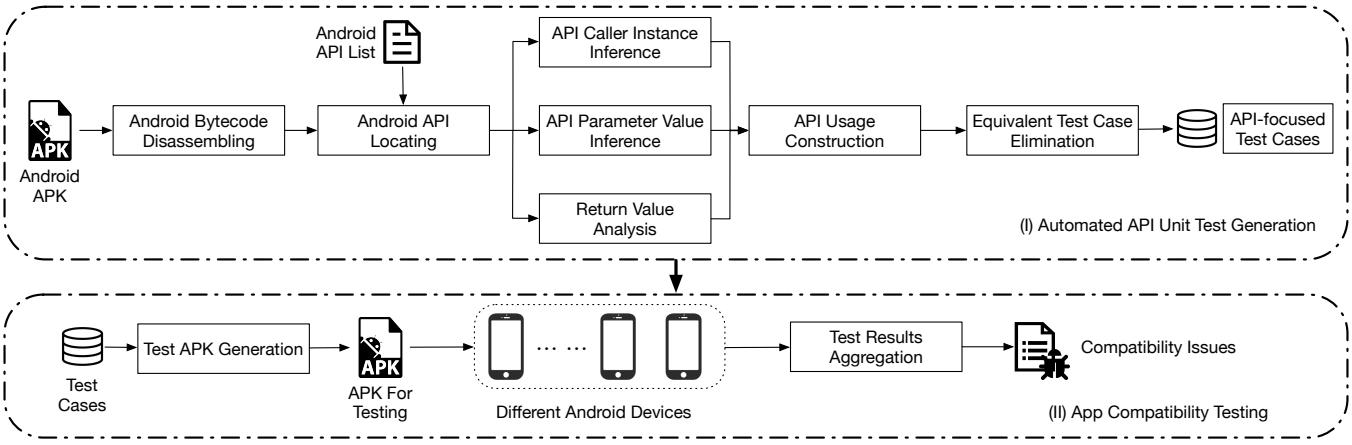

Figure 2: The working process of our approach.

Algorithm 1 gives the details of the approach. Given a target API as input, we apply a backward data flow analysis to identify the minimum executable context of the target API (lines 1~12). As shown in Algorithm 1, we describe the API caller inference process for both static methods (lines 4~6) and instance methods (lines 7~10). For static methods, we return its corresponding API signature, which has been obtained in step I.2. For instance methods, we first locate the definition statement of the calling object through invoking method *getDefinitionStmt* (line 8), which returns the definition statement for a local variable. This method walks the inter-procedural control flow graph from the target API statement in reverse order, aiming to look for the nearest assignment statement defining the API's calling object. After that, with the help of the function *constructCallTrace*(i.e., defined in Algorithm 2) (line 9), we can extract the call trace corresponding to the calling object. As shown in Algorithm 2, we handle parameter callers (lines 7~13) and method callers (lines 14~22), respectively. If the definition statement of the calling object comes from a parameter reference, we first retrieve all the statements at which the invocation occurs, and then for each of the statements, we recursively construct its call trace by calling the method *constructCallTrace*. A similar process has been applied to handle method callers (lines 14~22), which recursively construct the call trace involving statements of the calling object. The recursive process will not terminate until any of the conditions have been satisfied in line 3, i.e., either the statement is a constant, or it has been visited before, or it is an Android system API.

We elaborate on this process with a Jimple code example presented in Listing 4. In this example, the target Android API to test (i.e., *queryDetailsForUid(int,String,long,long,int)*) is invoked in line 26 by the calling object *$r3*, where *$r3* is a returned value of a self-defined method *getNetworkStatsManager(Context)* (line 22). We then step into the definition of the method *getNetworkStatsManager(Context)* (lines 1~7), and further backtrace the variables *$r2* and *$r1* along the call chain. *$r1* retrieves the network stats service from the application context *$r0* (line 4), and finally, the backtrace terminates at *$r0* (line 3), where all unknown variables are resolved.

```
1 public static android.app.usage.NetworkStatsManager
        getNetworkStatsManager(android.content.Context)
2 {
3   $r0 := @parameter0: android.content.Context;
4   $r1 = $r0. getSystemService("netstats");
5   $r2 = (android.app.usage.NetworkStatsManager) $r1;
6   return $r2;
7 }
8
9 public static int getUid(android.content.Context) throws
        android.content.pm.PackageManager$NameNotFoundException
10 {
11   $r0 := @parameter0: android.content.Context;
12   $r1 = $r0.getPackageManager();
13   $r2 = $r0.getPackageName();
14   $r3 = $r1.getApplicationInfo($r2, 1);
15   i0 = $r3.<android.content.pm.ApplicationInfo: int uid>;
16   return i0;
17 }
18
19 public static float getCurAppFlow(android.content.Context)
        throws
        android.content.pm.PackageManager$NameNotFoundException
20 {
21   $r0 := @parameter0: android.content.Context;
22   $r3 = DeviceInfoUtil.getNetworkStatsManager($r0);
23   $l1 = System.currentTimeMillis();
24   $i0 = DeviceInfoUtil.getUid($r0);
25   //Target API
26   $r5 = virtualinvoke
        $r3.<android.app.usage.NetworkStatsManager:
        android.app.usage.NetworkStats
        queryDetailsForUid(int,java.lang.String,long,long,int)>(0,
        "", 0L, $l1, $i0);
27 }
```

Listing 4: Code example demonstrating the usage of API *queryDetailsForUid*. The code snippet is extracted from app *com.eyoung.myutils*.

In addition to inter-procedural data-flow analysis, JUnitTestGen also needs to be field-aware. When performing backward data-flow analysis, the access of fields may break the original flow and hence may lead to unexpected results if not properly handled.

To mitigate this, we transform a field involved in the call trace of the API under testing (in the analyzed app) into a local variable in the generated test case. The local variable will be initiated following the same method as it is assigned in the original app. We then search the whole class to check how the field's value is assigned and subsequently apply the same method to initialize the local variable. If we cannot find the field's assignment or the assigned value is complicated to be reconstructed, we will use a dummy object to mock the required value.

In addition, JUnitTestGen needs to handle branches in the backward dataflow analysis. Specifically, we leverage the Inter-procedural Control Flow Graph (ICFG [3]), which is a combination of call-graph(CG) and control flow graph(CFG), to identify the minimum

executable context of the target API. Here, CG is a graph representing the calls between methods over the entire program, while CFG is a graph that represents the control flows in a single method. ICFG treats each statement (or a set of sequential statements) as a node, including branch statements that enable path-sensitive analyses, i.e., the propagation of different information along different branches. With ICFG, we are able to implement branch analysis by analyzing the structure of the graph. To this end, for those methods involving multiple branches, JUnitTestGen will separate each branch to form a different test case.

I.4 API Parameter Value Inference. To support API compatibility testing, e.g., to ensure that the API will, in any case, be reached once the test case is executed, we propose to directly assign possible values to the API's parameters inside the test case, i.e., the test case per se will not contain any parameter. To achieve this, JUnitTestGen also infers possible values for each parameter of the API to be tested. This step follows the same strategy i.e., the approach adopted to infer API caller instances, to infer the API's parameter values. This is done by performing inter-procedural backward data-flow analysis. We apply the same algorithms described in Algorithm 1 and Algorithm 2 on each parameter object to figure out the exact value.

Unfortunately, some Android APIs' parameter values may involve sophisticated operations when building their run-time values that are non-trivial to correctly retrieve statically. Specifically, if the analysis process does not end up at a constant/Android system API statement, the value of a parameter is regarded as being undiscovered. To mitigate this, we introduce a set of pre-defined rules to generate dummy values for such APIs that have their parameter values hard to retrieve practically. Some of the representative rules for generating dummy values for such hard-to-retrieve parameters are:

- For the eight primitive data types in Java (such as int, double, etc.) or their wrapper data types (such as Integer, Double, etc.) – we provide random values for each of them that conform to their types.
- For the String data type in Java – we generate a random alphanumeric string.
- For the Array parameter whose basic type is the eight primitive data types (or their wrapper data types) in Java (such as int, Integer, etc.) – we generate an Array variable with random primitive values.
- For Android system-related objects (or the intermediate objects in the calling object's instantiation process), we use a heuristic approach to obtain the corresponding constructors to create their instances. If an object has multiple constructors, we select the simplest one (with the least number of parameters) to achieve the highest possibility of constructing a valid object.

I.5 Return Value Analysis. To support detecting compatibility issues caused by return values (e.g., a given API may return A at API level X and B at API level Y), we propose to output the return value of the target API at the end of the test case. To achieve this, JUnitTestGen adds a statement at the end of the test case to further record the API's return value.

I.6 API Usage Construction. After obtaining the caller instance, the invocation statements along the call trace and the return object, we can now recover the call sequence from program entry to the target Android API by reversing the retrieved statements step by step. Based on the results of API caller instance inference along, JUnitTestGen will generate a test case containing the same number and type of parameters as the API to be tested. For example, as shown in Listing 5 at line 3, the generated test case contains five parameters, in the same type and order of the API under testing.

```
1  //for supporting generic testing
2  @Test
3  public void testQueryDetailsForUid(int var1, String var2,long
       var3, long var4, int var5) throws Exception {
4   Context var6 = InstrumentationRegistry.getTargetContext();
5   Object var7 = var6.getSystemService("netstats");
6   NetworkStatsManager var8 = (NetworkStatsManager) var7;
7   var8.queryDetailsForUid(var1, var2, var3, var4, var5);
8  }
9
10 //for supporting compatibility testing
11 @Test
12 public void testQueryDetailsForUid() throws Exception {
13  long var1 = System.currentTimeMillis();
14
15  Context var2 = InstrumentationRegistry.getTargetContext();
16  PackageManager var3 = var2.getPackageManager();
17  String var4 = var2.getPackageName();
18  ApplicationInfo var5 = var3.getApplicationInfo(var4, 1);
19  int var6 = var5.uid;
20
21  NetworkStats var7 = testQueryDetailsForUid(0, "", 0L, var1,
       var6);
22 // Output return value
23  out(var7);
24 }
```

Listing 5: Examples of the generated test cases for API *queryDetailsForUid (int networkType, String subscriberId, long startTime, long endTime, int uid).*

Taking the results of the API parameter value inference step, JUnitTestGen will generate another test case containing no parameters (line 10 in Listing 5). This test case will directly call the former test case with prepared parameter values. This test case is specifically designed to support API compatibility testing. The former test case, on the contrary, is designed to serve a more general purpose. With the help of fuzzing testing approaches (to generate possible parameter values for the test case), we expect the former test case could be leveraged to discover not only compatibility issues but also design defects such as bugs and security issues. This trade-off allows JUnitTestGen to generate test cases that are at least suitable for identifying signature-based compatibility issues (e.g., a given API is no longer available in a certain framework), although it may not be effective enough to help identify semantic change involved compatibility issues.

Please note that there are several special classes, such as *InstrumentationRegistry*, involved in the generated unit test cases. These classes are part of the Android Testing Support Library provided by Google for supporting instrumented unit tests. Compared to traditional unit tests, also known as local unit tests which can run on the JVM, instrumented unit tests require the Android system to run (e.g., through physical Android devices or emulators). Since this requires us to generate actual Android apps to run on Android devices or emulators, instrumented tests are much slower than local unit tests.

Nevertheless, we still choose to use instrumented tests to examine the compatibility of Android APIs. This is because instrumented tests provide more fidelity than local unit tests, which we have found is essential to reveal potential compatibility issues, especially those that involve device-specific issues.

I.7 Equivalent Test Case Elimination. Considering that the constructed test cases could be equivalent (i.e., duplicated), it is necessary to filter them out to save subsequent testing time and resources. Here, based on the concept of semantic equivalence defined in operational semantics [12], we consider that two test cases are equivalent if they share the same API invocation sequence. To this end, we first obtain the API invocation sequence for each test case and then examine the discrepancy between any two of them to check if $O_a = O_b$, where O_a and O_b are the lists of API invocations in two different test cases. Based on this rule, we are able to eliminate equivalent tests (i.e., the first test case is retained). After this step, for the sake of simplicity, if there are still multiple test cases retained for a given API, we will select the small-scale one (with the least number of method invocations) for supporting follow-up analyses.

3.2 App Compatibility Testing

Using the unit test cases generated by the first module, the second module of JUnitTestGen leverages them to check if the corresponding Android APIs will likely induce app compatibility issues. It first assembles all the test cases into an Android app and then aggregates their execution results against different devices running different Android frameworks. We now briefly detail these two steps below.

II.1 Test APK Generation. As discussed earlier, we have to resort to instrumented unit tests to examine Android APIs' incompatibilities. This process essentially requires us to generate an Android app (or APK) to be installed and executed on Android systems. Fortunately, Google has provided such a mechanism to achieve this purpose, i.e., supporting instrumented tests for a limited number of Android APIs. In this work, we directly reuse this mechanism to generate the test APK for all the unit tests automatically generated by JUnitTestGen.

II.2 Test Results Aggregation. After the test APK is generated, we can distribute it for execution on multiple devices. Since the test APK contains only known test cases, it is quite straightforward to execute it fully. The only challenge that lies in this step is to select the right set of devices on which to execute the test cases to reveal as many incompatible APIs as possible. Crowdsourced app testing could be an approach to achieve this purpose.

After installing and executing the generated test APK on multiple devices, the last step is in aggregating the test results to highlight potential compatibility issues in the app. Inspired by the experimental setup of the work proposed by Cai et al. [4], we consider an API as a potential incompatible case if 1) its corresponding test case can successfully run on a nonempty set of devices while failing on others; 2) its corresponding test case returns different values when running on different SDKs.

4 EVALUATION

Our JUnitTestGen aims to generate unit test cases covering as many Android APIs as possible, so as to allow the discovery of more API-induced compatibility issues in apps. To evaluate if this goal has been fulfilled, we propose to answer the following three research questions.

RQ1 To what extent can JUnitTestGen generate executable unit test cases for Android APIs?

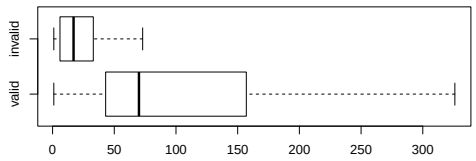

Figure 3: Distribution of the number of valid and invalid test cases per APK.

RQ2 How effective is JUnitTestGen in discovering API-induced Compatibility Issues?

RQ3 How does JUnitTestGen compare with existing tools in detecting compatibility issues?

4.1 Experimental Setup

To investigate the success rate of JUnitTestGen in producing valid test cases, we randomly select 1,000 Android apps for each target SDK version between 21 (i.e., Android 5.0) and 30 (i.e., Android 11.0[4]) from AndroZoo to prepare the experimental dataset. Here, we select 1,000 apps for each target SDK version because compatibility issues mainly lie in the evolution of APIs on different Android SDK versions [22]. Here, the criteria for app selection are based on the targetSdkVersion, which is the most appropriate API level on which the app is designed to run. Hence, the overall dataset for the experiment contains 10,000 Android apps whose target API versions are distributed equally across ten API levels. Our experiment runs on a Linux server with Intel(R) Core(TM) i9-9920X CPU @ 3.50GHz and 128GB RAM. The timeout setting for analyzing each app with JUnitTestGen is 20 minutes. In this experiment, we generate the test cases for all Android APIs that have been invoked in the apps. However, users of JUnitTestGen could provide a customized list of APIs to only generate test cases based on their interests.

4.2 RQ1-Effectiveness

Our first research question concerns the effectiveness of JUnitTestGen in mining API usages from existing Android apps to generate valid unit tests for Android APIs. In this work, we consider a test case to be valid if (1) the generated code snippet can be successfully compiled on all API levels and (2) the test case does not throw an exception before the execution point of the API on all API levels. The first condition ensures that the test case is syntactically correct. The second condition makes sure that the API's execution environment is properly set up. In other words, the second condition ensures that the exceptions we collected from valid test cases are exceptions thrown by the API under testing, which is essential for examining if the API will induce compatibility issues.

Result. Among the 10,000 randomly selected apps, JUnitTestGen generates in total 1,032,182 test cases. After eliminating the equivalent test cases, 66,499 of them are retained as distinct test cases, w.r.t. 28,367 distinct Android APIs. For the sake of simplicity, we select the small-scale test case (with the least number of invocation sequences) for each API (i.e., 28,367 test cases) for further study. By compiling and executing these 28,367 test cases, we further confirm that 5,562 of them are invalid (i.e., 22,805 of them are valid), giving a success rate of 80.4% in generating valid test cases. In addition,

[4]The latest version at the time when we conducted this study.

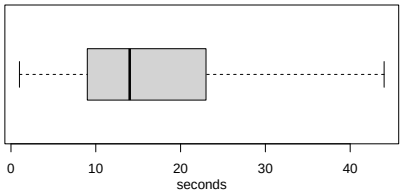

Figure 4: Distribution of time spent by JUnitTestGen to generate test cases per APK.

our manual analysis on 100[5] randomly selected test cases confirm that these test cases generated by JUnitTestGen are indeed valid.

Figure 3 summarizes the distribution of the number of valid and invalid test cases generated per app. The median number of valid and invalid test cases generated per app are 70 and 17, respectively, while their average are 106.29 and 25.37, respectively. Like most other state-of-the-art approaches, even though our static analysis approach has limitations so that it cannot generate valid test cases for every API, especially the complex ones, **our approach is still capable of generating more valid test cases than invalid ones.** This demonstrates the effectiveness of our approach in mining Android API usages to generate API unit test cases.

We note that the success rate of generating valid test cases – 80.4% at the moment – is important but not crucial to our work. Theoretically, as long as we increase the number of Android apps considered for learning, we would likely be able to generate valid test cases for the given API under testing. For the test cases that are regarded as invalid, we further manually look into their root causes. Our in-depth analysis reveals that the invalid cases are mainly caused by the lack of prerequisites (e.g., resource files), especially in UI-related APIs. For example, UI objects can hardly be programmatically initialized without certain resource files.

We further look at the efficiency of JUnitTestGen by reporting its time cost when applied to generate test cases. Figure 4 summarizes the distribution of time consumed by JUnitTestGen per app. On average, it takes 17.45 seconds to process an app. The median time cost is 14 seconds. The fact that the time spent by JUnitTestGen to process an Android app is quite small suggests that it is practical to apply JUnitTestGen to analyze large-scale Android apps.

Answers to RQ 1. *By learning from existing API usages, our approach can automatically generate API unit test cases. Despite various challenges posed by advanced Android programming features, our static analysis approach can still achieve over 80% of the success rate in generating valid test cases.*

4.3 RQ2-Performance on Real-World Applications

The ultimate goal of JUnitTestGen is to help better identify API compatibility issues that occur during Android system evolution. To this end, in this research question, we evaluate, based on the generated valid unit test cases, to what extent can JUnitTestGen help in identifying API-induced compatibility issues.

[5]The sample size is determined based on a confidence level at 95% and a confidence interval at 10(https://www.surveysystem.com/sscalc.htm).

We consider an Android API has a compatibility issue if the execution results on different Android SDK versions are inconsistent. More specifically, an API is considered to have a compatibility issue if any of the following happens: (1) the corresponding test case runs successfully on some Android SDK versions but fails (e.g., throws errors or exceptions) on others; or (2) the corresponding test case throws different errors or exceptions on different Android SDK versions (e.g., throws *NoSuchMethodError* on some versions, while throws *IllegalArgumentException* on others); or (3) the return values of a target API are non-identical on different SDK versions.

Result. Recall that JUnitTestGen successfully generates 22,805 distinct valid test cases covering 22,805 unique Android APIs, based on the randomly selected 10,000 apps. In this work, these 22,805 test cases are respectively executed on ten Android emulators running API levels from 21 to 30.

By comparing the experimental results against the aforementioned three rules, **we are able to locate 3,488 compatibility issues covering 2,695 Android APIs.** Note that some APIs may involve more than one compatibility issue. To confirm whether the APIs identified by JUnitTestGen indeed have compatibility issues, we manually examined 100 randomly selected APIs reported to have compatibility issues, 100 of them are confirmed to have compatibility issues (i.e., true positive results). Here, we remind the readers that all of the compatibility issues are actually identified through dynamic analysis, which is expected to be highly accurate. In the manual validation process, we manually examine the APIs' implementation in Android framework source code across different SDK versions and compare it with the release notes in the official API documentation.

According to the root causes of the compatibility issues, we further categorize them into the following types.

- **Type 1: Signature-based compatibility issue.** This type refers to the incompatibility caused by adding new APIs, deprecating existing APIs, or changing existing APIs' signatures, such as changing parameters or return types.
- **Type 2: Semantics-based compatibility issue.** This type refers to the incompatibility caused by the same API (i.e., the signature is not changed) behaving differently on different Android API levels. Based on the APIs' behaviours, semantic-based compatibility issues are further breakdown into the following two sub-types.
  1) Type 2.1 Semantics-based compatibility issue: Different Errors: The APIs categorized into this type will throw exceptions or errors (including app crashes) on devices running some SDK versions but will not (either running successfully or throw different exceptions/errors) when running on devices with other SDK versions.
  2) Type 2.2 Semantics-based compatibility issue: Different Return Values: APIs of this type will not directly cause compatibility issues. Under the same experimental setting, these APIs will return different values when running on different SDKs. However, if the different return values are not specifically distinguished, the subsequent code that uses these return values may behave differently on different devices, leading to also compatibility issues.

Among the 3,488 identified compatibility issues, 438 of them suffer from signature-based issues, while 3,050 are caused by semantic issues. Within the semantic-based compatibility issues, 946 of them

are caused by different errors (Type 2.1), while the remaining 2,104 cases observe different return values (Type 2.2). Table 1 summarizes the possible errors/exceptions that cause the signature/semantic compatibility issues.

As expected, the most common error is no such method error, which can be caused by (1) the API being deprecated and removed from the framework, or (2) the API not yet introduced. Both reasons are introduced by the evolution of the framework. As also revealed by Li et al. [22], the fast evolution of the Android framework has indeed introduced a lot of APIs that will likely induce compatibility issues. Except for signature-based issues, which are relatively easy to be statically identified (for example, by comparing the framework codebase of different versions), our approach has also found 2,974 issues (over five times the number of Type 1 issues) involving semantic changes of APIs, which are non-trivial to be identified statically [26].

Table 1: Categories and statistics of the observed error/exception types associated with compatibility issues.

Issue Type	Errors/Exceptions	Count
Signature	NoSuchMethodError	270
	NoClassDefFoundError	163
	NoSuchFieldError	5
Semantic	SecurityException	196
	NullPointerException	189
	RuntimeException	139
	Resources$NotFoundException	113
	IllegalArgumentException	67
	NoSuchElementException	42
	Crash	41
	IllegalStateException	24
	AndroidRuntimeException	23
	IOException	15
	ArrayIndexOutOfBoundsException	15
	FileNotFoundException	12
	PackageManager$NameNotFoundException	10
	IndexOutOfBoundsException	10
	ClassCastException	9
	IllegalAccessError	9
	ActivityNotFoundException	8
	StringIndexOutOfBoundsException	5
	UnsupportedOperationException	5
	BadTokenException	3
	CanceledException	3
	ErrnoException	2
	KeyChainException	2
	SQLiteCantOpenDatabaseException	1
	ParseException	1
	StackOverflowError	1
	NumberFormatException	1

Below, we elaborate on real-world compatibility issues for each type of case study.

Case Study 1: Signature-based Compatibility Issue. The API *android.content.pm.LauncherApps#hasShortcutHostPermission* has been reported to contain a signature compatibility issue. The corresponding test case*(whose API usage is extracted from app ch.deletescape.lawnchair.plah*[6]*)* throws *NoSuchMethodError* on Android SDK version 21 to 23 but can be successfully executed on Android SDK version 24 to 30. This result suggests that the API was introduced to the Android system since API level 24; therefore, it would cause an error if the containing app runs on devices with Android SDK version earlier than 24. However, according to the official Android API documentation, this API was added in API level 25 [9], which is imprecise according to our result. We further checked the source code of Android SDK 24 and confirmed its existence.

Table 2: The comparison results between JUnitTestGen and Evosuite, CiD.

Tool	# Type 1	# Type 2.1	# Type 2.2
JUnitTestGen	438	946	2,104
Evosuite	36	0	0
CiD	864	-	-

Case Study 2: Semantics-based compatibility issue caused by different Exceptions. The API *android.app.NotificationManager #notify* has been reported to contain a semantic compatibility issue. The corresponding test case*(whose API usage is extracted from app com.ag.dropit*[7]*)* can be successfully executed on Android SDK version 21 to 22 but throws *IllegalArgumentException* on Android SDK version 23 to 30. We manually looked into its source code in the Android codebase and found that the actual implementation of this API has been changed since API level 23, which added a sanity check of object *mSmallIcon*. This explains why it throws *IllegalArgumentException* when there is no valid small icon from the API level after 23. Unfortunately, the official Android API documentation does not reflect this change, which is misleading.

Case Study 3: Semantics-based compatibility issue caused by different return values. The API *(extracted from app cleaner.clean.booster*[8]*) <android.text.format.Formatter: String formatShortFileSize(Context, long)>* has a return value-induced compatibility issue. The format of the return values vary on different API levels: given the 1L(The long data type of value 1) as the second parameter, the return value on API level 21 to 22 is *"1.0B"*, the return value on API level 23 is *"1.0 B"* (with additional whitespace between 1.0 and B), while the return value on API level 24 to 30 is *"1 B"*. The difference in return values can introduce potential issues if app developers rely on the return value to implement future functions without checking the running API level. For example, if app developers cast the return value from String to Byte afterwards, it may throw an exception if users ignore the value discrepancy on different API levels.

Answers to RQ 2. *Our approach is useful in automatically pinpointing API-induced compatibility issues. It also goes beyond the state-of-the-art to be capable of detecting not only signature-based compatibility issues but also more significant semantics-based compatibility issues, i.e., the corresponding APIs' signatures are kept the same, while their semantics are altered.*

4.4 RQ3-Comparison with State-of-the-art

Considering the main purpose of our work is generating test cases for detecting compatibility issues, both generic test case generation approaches, such as EvoSuite [8] and compatibility issues detection tools, such as CiD [22], are selected as the baselines to evaluate our approach. We evaluate the performance of JUnitTestGen, EvoSuite [8] and CiD [22] in detecting compatibility issues. Overall, table 2 lists the number of compatibility issues found by JUnitTestGen, EvoSuite and CiD. We then break down the comparative results as follows:

[6]SHA-256:bf4e6e7fb594cd9db4b168a68f70157ad9c1fea0192e0bd5d9a39d1c38802639

[7]SHA-256:30f7f72cebeffd7c6e26489198ee5ad244bd44b076dd9cb59865d8b0e82a86af

[8]SHA-256:def5db37b3a68de62a0472e872700092060bdec3e875d4f476fcda52795bceb2

Comparison with EvoSuite. To compare JUnitTestGen with generic test case generation tools, we choose EvoSuite as the baseline because EvoSuite has been considered the state-of-the-practice test generation tool, which achieved the highest score at the SBST 2021 Tool Competition [46]. EvoSuite uses an evolutionary search approach to generate and optimize test suites toward satisfying an entire coverage criterion for Java classes. We remind the readers that the objective of EvoSuite is to generate tests for classes, not directly aiming at generating tests for APIs, which are the main target when concerning compatibility issues happening in Android apps. Since Evosuite can only generate tests based on source code, we resort to AOSP from SDK 21 to 30 for Evosuite to generate test cases. In total, Evosuite successfully generates 5,335 test cases. We then execute all of them on SDK versions from 21 to 30 and apply the same rules for determining compatibility issues as used in JUnitTestGen.

As shown in Table 2, EvoSuite only finds 36 signature-based compatibility issues, and no semantic compatibility issues are identified. We further manually check the test cases generated by EvoSuite and observe that the false negatives (compared with JUnitTestGen) are mainly caused by overlooking API dependency information. For example, some APIs can only be invoked by system services, which makes EvoSuite fail to generate valid tests without knowing the usage of such APIs. Missing API dependency information makes EvoSuite insufficient in pinpointing compatibility issues. In other words, our comparison result reveals that mining API usage from apps is beneficial for finding compatibility issues.

Comparison with CiD. To the best of our knowledge, no work has been devoted to detecting compatibility issues in Android apps dynamically. CiD [22] is the closest work to ours on detecting compatibility issues. CiD model and compare API signatures on different SDK versions to detect compatibility issues. We thus evaluate the performance of JUnitTestGen and CiD on the same dataset in RQ1, which contains 10,000 Android apps.

In total, CiD detects 864 compatibility issues, as highlighted in Table 2. Out of the 864 compatibility issues detected by CiD, JUnitTestGen successfully identified 3,050 cases that have been overlooked by CiD. We further randomly analyze 50 false negatives of CiD and find that these issues are caused by the lack of semantics analysis of the API implementation. Specifically, when analyzing the evolution of APIs, CiD only examines the change of API signatures (including name, type, and parameters); hence it is not capable of detecting APIs that modify the implementation details but retain the same signature. Also, we find other 51 false positives of CiD are caused by imprecisely extracting the usage of the APIs, due to the context-insensitive approach of CiD when building the conditional call graph (CCG). On the other hand, we find that JUnitTestGen miss 375 cases reported in CiD (i.e., false negatives in JUnitTestGen). This is mainly caused by the sophisticated usage of some APIs, which cannot easily be initialized programmatically (e.g. UI-related APIs). For example, some APIs may involve the initialization of UI objects that cannot be initialized programmatically. This makes it very challenging to automatically generate unit test cases in some circumstances. However, we argue that this limitation can be alleviated by manually adding prerequisite resources to create more valid tests. Overall, the comparison results reveal the weakness of static analysis approaches in detecting semantic compatibility issues, i.e., yields false-positive results and is hard to detect issues involving semantic changes in methods. It also demonstrates that our approach can indeed find more diverse compatibility issues and hence is promising to complement existing static approaches.

Answers to RQ 3. *JUnitTestGen outperforms the state-of-the-practice test generation tool, EvoSuite, and the state-of-the-art static compatibility detection tool, CiD, in pinpointing compatibility issues caused by the fast evolution of Android APIs. This experimental evidence shows the necessity to perform dynamic testing to pinpoint compatibility issues in Android apps, and it should take API usage dependencies into consideration when generating test cases to fulfill the dynamic testing approach.*

5 DISCUSSION

We now discuss the potential implications and limitations of this work.

5.1 Implications

Better Supplementing Compatibility Analysis: JUnitTestGen is able to automatically generate tests for Android APIs for detecting both signature-based and semantics-based compatibility issues. Previous static analysis works [22] overlooked the semantics-based ones (i.e., APIs have the same signature but different implementation), which are more challenging to detect statically. Together with the state-of-the-art static analysis-based methods, our proposed method can provide a more comprehensive overview of compatibility issues in Android APIs. Therefore, we argue that there is a need to invent hybrid approaches to take advantage of both static analysis and dynamic analysis to conquer compatibility issues.

Beyond Compatibility Testing: JUnitTestGen is not only useful in pinpointing compatibility issues in Android APIs but also could be easily adopted to automatically generate test cases for other purposes. For instance, in the cases that an API takes various parameters as input, it can work with other testing approaches such as fuzz testing to explore the API's implementation dynamically. It hence goes beyond compatibility testing and provides a more general-purpose form of API testing.

Go Beyond Android. Our approach performs static program analysis to learn and generate test cases from existing API usages, which are not strongly attached to Android apps. We believe it could be easily adapted to analyze other Java projects, e.g., to automatically generate test cases for popular Java libraries. Although our approach cannot be directly applied to analyze projects written in other programming languages than Java, we believe the idea and methodology proposed in this paper could still work. We plan to explore these research directions in our future work. We also encourage our fellow researchers to explore this direction further.

5.2 Limitations

The main limitation of JUnitTestGen lies in its backward data-flow analysis when inferring API caller instance and API parameter values. Indeed, as already known in the community, it is non-trivial to implement a sound data-flow analysis. Other researchers often accept trade-offs to obtain relatively good results, and this is the same in our case. When the variables to be backwardly retrieved are

complex, e.g., involving constructing various intermediate objects, JUnitTestGen rely on their simplest constructor to initialize the objects to generate a valid test case. Unfortunately, some of the constructors are too complex to initialize, leading to invalid test cases. Some other APIs may involve the initialization of UI objects that cannot be initialized programmatically (hence random values are leveraged to handle such cases). This makes it very challenging to automatically generate unit test cases in all circumstances.

Second, currently, our JUnitTestGen data-flow analysis is agnostic to some advanced programming features, such as reflective and native calls. This may further impact the soundness of our approach. As part of our future work, we plan to integrate approaches developed by our fellow researchers to mitigate those long-standing challenges (e.g., applying DroidRA [20] to mitigate the impact of reflective calls on our static analysis approach.).

Third, the capability of our approach is limited by the number of Android APIs leveraged by real-world apps. Our approach cannot generate unit test cases for such APIs that have never been accessed by real-world Android apps. Nevertheless, we argue that this impact is not significant as the APIs that have never been used by app developers should have a low priority to be tested than the others that are frequently accessed. Compared to the latter case, the former ones will not cause problems such as crashes to Android apps. Subsequently, it will not impact the user experience and the reputation of the app developers.

Fourth, considering the generated parameter values are inferred from real-world apps, which might not reveal all possible semantic-related compatibility issues. In other words, the capability of our approach is limited by the values of parameters leveraged by real-world apps. As for our future work, we plan to integrate fuzzing techniques [36, 42, 52] into our approach so as to trigger as many compatibility issues as possible.

Fifth, our definition of compatibility issue is based on the observation that the same test case throws different errors or exceptions on different Android SDK versions. However, this might introduce false negatives because the tests that throw the same exception across all SDK versions are ignored. Nevertheless, we argue that this type of false-negative requires further human validation and thus cannot be determined automatically. In addition, related works [4] also have not considered this situation.

Sixth, the tests generated by our approach may suffer from flaky tests. Indeed, non-deterministic return values may appear under different execution environments, leading to false positives. However, it is a non-trivial task to tackle flaky tests [56] because the root causes of flaky tests can be quite sophisticated. Nevertheless, as part of our future work, we plan to integrate other approaches developed by our fellow researchers to mitigate this long-standing challenge, e.g., by applying FlakeScanner [6] to mitigate the impact of flaky tests on our approach.

Seventh, the types of compatibility issues detected in our approach are the ones that are related to exceptions or return values. However, this will certainly not be complete to cover all possible cases. For example, the value of variables in the same API may evolve on different SDK versions. Indeed, as summarised by Liu et al. [27], apart from compatibility issues raised by API signature/semantic changes and return value differences, there exist other types of compatibility issues, such as those introduced by field evolution, callback method changes, etc. Nevertheless, as also highlighted by Liu et al. [27], the number of such compatibility issues is quite limited, suggesting that the impact of such cases on our approach may not be significant.

Eighth, the main objective of the generated test cases in this work is to pinpoint compatibility issues. The quality of these test cases (e.g., readability, overlaps, and maintenance) has hence not been considered. We therefore commit to further investigating the quality of the generated test cases in our future work.

Last but not least, our approach relies on existing code examples to generate test cases. However, the selected code examples may contain sub-optimal or erroneous API usages. Nevertheless, we argue that this impact is not significant as we extract code examples from real-world Android apps from Google Play, for which thousands of users might have used (hence tested) them in practice.

6 RELATED WORK

Android API evolution is a critical issue in software maintenance [5, 14, 16, 18, 19, 23, 28, 29, 32, 34, 43, 55]. McDonnell et al. [30] have shown that the Android system updates 115 APIs per month on average, while app developers usually adopt the new APIs much more slower. The slow adoption of API updates may raise various issues, such as security and compatibility. An empirical study on StackOverflow conducted by Linares-Vasquez et al. [25] suggests that API updates would trigger more discussions, especially if APIs are removed from the Android system. They also revealed that users are in more favour of apps that use less fault and change-prone APIs [2, 24], as these apps would likely introduce fewer failures, crashes and other bugs.

Android developers have long been suffered from compatibility issues due to the fast-evolving and fragmented nature of the Android ecosystem [13, 21, 33, 50, 54]. Researchers have proposed several solutions for detecting compatibility issues of Android APIs. Wei et al. [47, 49] conducted an empirical study to investigate fragmentation-induced API compatibility issues and proposed a tool named FicFinder to detect such APIs. FicFinder identifies APIs with compatibility issues based on heuristic rules manually derived from a limited number of Android apps, which is expected to introduce high false negatives (i.e., missing undiscovered compatibility issues). Thereafter, several works have been proposed to leverage data-driven techniques that automatically mine compatibility issues from various sources such as Android code base and real-world apps. Li et al. proposed a tool named CiD to detect potential compatibility issues by mining the history of the Android framework source code. CiD identifies Android APIs' lifetimes and finds if an app's declared supported versions conflict with its used APIs.

Comparable to our method, several works have been proposed to mine API usage from real-world apps. Scalabrino et al. [39, 40] considered the APIs wrapped in a version check condition (e.g., *if (Build.VERSION.SDK_INT >= 21)*) to potentially have compatibility issues and developed a tool named ACRYL to extract such APIs from real-world apps. However, ACRYL can only detect APIs whose compatibility issue is already known by the developers (i.e., they are enclosed in the version check conditions by the developers), while our method is capable of detecting zero-day compatibility issues that the app developers are not yet aware of, or even Google

itself. Other generic test generation tools, such as EvoSuite and Randoop[35], are able to generate tests for Java classes. However, these tools do not directly aim at generating tests for APIs and they have been demonstrated as insufficient in pinpointing compatibility issues because of the lack of API usage knowledge.

7 CONCLUSION

In this work, we presented a novel prototype tool, JUnitTestGen, that mines existing Android API usages to generate API-focused unit test cases automatically for pinpointing potential compatibility issues caused by the fast evolution of the Android framework. Experimental results on thousands of real-world Android apps show that (1) JUnitTestGen is capable of automatically generating valid unit test cases for Android APIs with an 80.4% success rate; (2) the automatically generated test cases are useful for pinpointing API-induced compatibility issues, including not only signature-based but also semantics-based compatibility issues; and (3) JUnitTestGen outperforms the state-of-the-practice test generation tool, EvoSuite, and the state-of-the-art static compatibility detection tool, CiD, in pinpointing compatibility issues.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments on this paper. This work was partly supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP20010002.

REFERENCES

[1] 24 April 2022. Kex. https://github.com/vorpal-research/kex/tree/sbst-contest. Online; accessed 24 April 2022.
[2] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering* 41, 4 (2014), 384–407.
[3] Eric Bodden. 2012. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 3–8.
[4] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 216–227.
[5] Danny Dig and Ralph Johnson. 2005. The role of refactorings in API evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 389–398.
[6] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. Flaky test detection in Android via event order exploration. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 367–378.
[7] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*. IEEE, 31–40.
[8] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
[9] Google. 2021. LauncherApps. https://developer.android.com/reference/android/content/pm/LauncherApps#hasShortcutHostPermission(). Online; accessed 27 January 2021.
[10] Hyung Kil Ham and Young Bom Park. 2011. Mobile application compatibility test system design for android fragmentation. In *International Conference on Advanced Software Engineering and Its Applications*. Springer, 314–320.
[11] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.
[12] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 81–92.
[13] Muhammad Kamran, Junaid Rashid, and Muhammad Wasif Nisar. 2016. Android fragmentation classification, causes, problems and solutions. *International Journal of Computer Science and Information Security* 14, 9 (2016), 992.
[14] Puneet Kapur, Brad Cossette, and Robert J Walker. 2010. Refactoring references for library migration. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 726–738.
[15] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and efficient API misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 192–203.
[16] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting Android API migrations using code examples. *IEEE Transactions on Software Engineering* (2020).
[17] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.
[18] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2018. MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*.
[19] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing inaccessible android apis: An empirical study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 411–422.
[20] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 318–329.
[21] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.
[22] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.
[23] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 254–264.
[24] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 477–487.
[25] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*. 83–94.
[26] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and characterizing silently-evolved methods in the android API. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2021)*. IEEE, 308–317.
[27] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. 2022. Automatically Detecting API-induced Compatibility Issues in Android Apps: A Comparative Analysis (Replicability Study). *arXiv preprint arXiv:2205.15561* (2022).
[28] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep Learning for Android Malware Defenses: a Systematic Literature Review. *Comput. Surveys* (jun 2022). https://doi.org/10.1145/3544968
[29] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE transactions on software engineering* 43, 9 (2016), 817–847.
[30] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 70–79.
[31] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John Mitchell. 2016. Target fragmentation in Android apps. In *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 204–213.
[32] Meiyappan Nagappan and Emad Shihab. 2016. Future trends in software engineering research for mobile apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. IEEE, 21–32.
[33] Fatih Nayebi, Jean-Marc Desharnais, and Alain Abran. 2012. The state of the art of mobile application usability evaluation. In *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 1–4.
[34] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1–18.
[35] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[36] Jibesh Patra and Michael Pradel. 2016. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664* (2016).

[37] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-misuse detection driven by fine-grained API-constraint knowledge graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 461–472.

[38] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. JuCify: a step towards Android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1232–1244.

[39] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 288–298.

[40] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Valentina Piantadosi, Michele Lanza, and Rocco Oliveto. 2020. API compatibility issues in Android: Causes and effectiveness of data-driven detection techniques. *Empirical Software Engineering* 25, 6 (2020), 5006–5046.

[41] Stack OverFlow. 2021. Permissions needed for NotificationManager. https://stackoverflow.com/questions/41308512/permissions-needed-for-notificationmanager. Online; accessed 27 January 2021.

[42] Ai-Fen Sui, Wen Tang, Jian Jun Hu, and Ming Zhu Li. 2011. An effective fuzz input generation method for protocol testing. In *2011 IEEE 13th International Conference on Communication Technology*. IEEE, 728–731.

[43] Xiaoyu Sun, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. 2021. Characterizing Sensor Leaks in Android Apps. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 498–509.

[44] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2021. Taming reflection: An essential step toward whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–36.

[45] 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*. 214–224.

[46] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EVOSUITE at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 28–29.

[47] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.

[48] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning API-device correlations to facilitate Android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 878–888.

[49] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1176–1199.

[50] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 886–898.

[51] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1105–1116.

[52] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 722–733.

[53] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. 2015. Compatibility testing service for mobile applications. In *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE, 179–186.

[54] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *The 44th International Conference on Software Engineering (ICSE 2022)*.

[55] Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 266–277.

[56] Behrouz Zolfaghari, Reza M Parizi, Gautam Srivastava, and Yoseph Hailemariam. 2021. Root causing, detecting, and fixing flaky tests: state of the art and future roadmap. *Software: Practice and Experience* 51, 5 (2021), 851–867.