Effective Unit Test Generation for Android Apps

Guojun Ma*, Yu Pei†§, Liushan Chen*§, Chenqing Gan*, Hao Zhang‡, Hao Liang*, Tian Zhang‡
*Douyin Co., Ltd., Shenzhen, China, {maguojun,chenliushan,ganchenqing,lianghao.roger}@bytedance.com
†The Hong Kong Polytechnic University, Hong Kong, China, yupei@polyu.edu.hk
‡Nanjing University, China, mf21330110@smail.nju.edu.cn, ztluck@nju.edu.cn

Abstract—**While the received wisdom says that testing at levels like classes and methods is necessary for detecting bugs in programs, the application of unit testing to Android development in practice is limited so far due to the lack of sufficient technical and tool support. This paper proposes the EvoDroid approach to the automated unit test suite generation for Android code. EvoDroid is inspired by EvoObj, a SOTA test generation technique for object-oriented Java programs based on EvoSuite. EvoObj generates unit test suites for Java methods and constructs object construction graphs to guide the synthesis of complex objects as test inputs. In contrast to that, EvoDroid generates test suites for Java classes, and its object synthesis is driven by input structure maps which are comparably effective but much less expensive to construct. EvoDroid also integrates the Robolectric framework to support running Android unit tests on regular Java virtual machines. Experimental evaluation results show that EvoDroid is both effective and efficient in generating unit test suites for Android.**

Index Terms—**search-based unit test generation, static analysis, Android**

I. Introduction

Android is now the dominant operating system on mobile devices, and the number of Android apps on the market is growing rapidly [1]. Given that our work and life rely more and more on the correct functioning of Android apps, considerable effort has been dedicated in the past years to detect bugs in Android apps via testing effectively. Since most Android apps have a graphical user interface (GUI) for better usability, GUI testing has been the most popular form of testing performed on Android apps. Particularly, various techniques and tools have been developed to automatically generate GUI tests [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], to augment existing GUI tests [15], [16], [17], [18], and to repair obsolete GUI tests after the apps have evolved [19], [20], [21], [22]. While GUI testing is essential for checking whether an Android app behaves as expected at the system level, the received wisdom says that testing at levels of smaller code units like classes and methods is necessary since system-level testing, like GUI testing, alone is insufficient for detecting many bugs in the app. In fact, unit testing for Android apps is not only necessary but also feasible. Although most code in Android apps depends on the Android environment, a significant portion of such code encodes only the internal processing logic of the corresponding apps and is suitable for unit testing. For example, given a class `Bitmap` from Android.jar that abstracts bitmap images, it is quite reasonable to unit test a client class `BitmapUtils` that relies on class `Bitmap` and provides utility methods to manipulate bitmap images.

However, the application of unit testing to Android development in practice is limited so far. One major reason is that many unit tests for Android code can only be executed on Android platforms like Android emulators and devices, making the unit testing process quite heavy. More concretely, many classes in Android apps are *Android specific*, i.e., they directly or indirectly depend on the Android environment (including support libraries like Android.jar and kotlin-stdlib.jar), while full-fledged Android environments are only available on Android platforms. Therefore, given a unit test that exercises an Android-specific class, executing the unit test typically involves the following steps: Packing the test with the corresponding app code into an APK file, deploying the file to an Android platform, and launching the app and running the test.[1] Note that although Android IDEs like Android Studio and IntelliJ IDEA enable programmers to develop Android apps on desktop computers, they only support the compilation, but not the execution, of Android-specific code since they merely mock the APIs of the Android environment.

The restriction that unit tests for Android-specific code must be executed on Android platforms not only significantly increases the already huge effort required for Android unit testing but also limits the tool support for generating, analyzing, and enhancing Android unit tests. On the one hand, existing Java testing techniques and tools are ineffective on Android apps since they do not run on Android platforms and can only handle code that is not Android-specific. On the other hand, it is challenging to migrate techniques like dynamic analysis to the Android platform because they rely on the platform APIs and must meet the platform restrictions. For example, when generating GUI tests for an Android app, besides utilizing an emulator to run the app, an automated tool also needs to gather information about the current status of the app GUI, decide which event is to be triggered on which GUI widget next based on the gathered information, and actually trigger the selected event. The process is only feasible because the emulator supports the corresponding operations and makes them readily accessible via clearly defined APIs. The Android platform, however, provides little, if any, support for gathering information about the execution of Android apps at the method

§ Yu Pei and Liushan Chen are the corresponding authors.

[1] AndroidJUnitRunner [23] streamlines the process, but an Android emulator or device is still needed.

or statement level.

Recently, new techniques have been proposed to facilitate the unit testing of Android-specific code, which includes, e.g., ROBOLECTRIC, an industry-standard unit testing framework for Android [24]. The ROBOLECTRIC framework supports the execution of Android-specific code on a regular Java virtual machine (JVM) by simulating an Android environment atop the JVM. By avoiding the aforementioned overhead, ROBOLECTRIC makes unit testing for Android-specific code considerably more affordable to Android developers.

To further reduce the burden of manually preparing those unit tests, we propose the EVODROID approach to support effective automated unit test generation for Android-specific code in this paper. EVODROID is inspired by the EVOOBJ [25] approach. EVOOBJ extends the EVOSUITE [26] evolutionary test generation technique to generate unit test suites for Java methods effectively, and it constructs object construction graphs to guide the synthesis of complex objects as test inputs. While EVOOBJ produced better results than EVOSUITE on open-source Java code in the reported experimental evaluation [25], in our experience, its efficiency degrades significantly and often becomes unacceptable on more complex code. Compared with EVOOBJ, EVODROID is also built atop EVOSUITE, but it aims to generate unit test suites for Java classes, and object synthesis in EVODROID is driven by *input structure maps*, which are comparably effective but less expensive to construct. EVODROID also incorporates the ROBOLECTRIC framework to streamline the execution of Android unit tests on regular Java virtual machines. We summarize the advantages of class-level unit test generation based on input structure maps over method-level unit test generation based on object construction graph in Section III.

The EVODROID approach has been implemented into a tool with the same name. We have experimentally compared EVODROID with the STOAT tool in test generation for open-source Android apps and with EVOOBJ in unit test generation for Java code, and we have applied EVODROID to facilitate the unit and regression testing of production-level Android code. The experimental results clearly show that EVODROID complements STOAT in test generation for Android, outperforms EVOOBJ in unit test generation for Java, and is effective in testing production-level Android code.

This paper makes the following contributions:

1) We propose the EVODROID approach to effective unit test generation for Android; To the best of our knowledge, EVODROID is the first *class-level unit test generation technique for Android code*.
2) We implement the EVODROID approach into a tool with the same name to facilitate the easy application of the approach by developers;
3) We evaluate the effectiveness and efficiency of the approach based on the results it produces on real-world Android and Java code.

II. BACKGROUND

A. Android Environment and Platform

Every Android app directly or indirectly depends on the libraries that come with the Android SDK and invokes methods in those libraries to provide its functionalities at runtime. For example, Android framework classes are usually contained in a library named android.jar. Correspondingly, full-fledged implementations of those libraries can be viewed as *Android environments* necessary for Android apps to run successfully. While all Android devices and emulators, which we refer to as *Android platforms*, are equipped with full Android environments, no such *native* environments exist on popular desktop platforms like Windows[2], MacOS, and other Linux-based operating systems, and therefore Android apps cannot directly execute on those platforms. Note that although Android IDEs like Android Studio and IntelliJ IDEA enable programmers to develop Android apps on desktop computers, libraries like Android.jar that come with the IDEs merely mock the APIs of the Android environment and support only the compilation, but not the execution, of Android apps.

B. ROBOLECTRIC

ROBOLECTRIC is a testing framework that enables unit tests of Android-specific code to execute on regular JVMs. It achieves so by providing a native implementation of the Android environment and redirecting calls to Android APIs in the test code to that implementation. In particular, ROBOLECTRIC introduces a home-brewed class `SandboxClassLoader`, which wraps around the JVM's default class loader. During class loading, `SandboxClassLoader` first replaces classes from the Android libraries with their matching native implementations, then modifies the other classes' bytecode so that invocations to Android APIs are redirected to the corresponding methods in the native implementation, and finally delegates the loading of the replaced/modified classes to the JVM's default class loader. The sandbox environment supports access to Android resources in a similar way.

To be able to run an Android unit test method in the ROBOLECTRIC environment, developers just need to annotate the method with class `RobolectricRunner`, which is a ROBOLECTRIC executor defined by the framework. Upon running the unit test with ROBOLECTRIC, a singleton instance of `RobolectricRunner` will first be instantiated and utilized to initialize the ROBOLECTRIC sandbox environment (including, e.g., `SandboxClassLoader`), and then the test method will be executed in the sandbox.

C. EVOSUITE *and* EVOOBJ

EVOSUITE is an automated, search-based unit test generation tool for Java. It employs an evolutionary algorithm [27] to construct a suite of unit test cases that maximizes the code coverage with respect to a given criterion. Each generated test case contains a sequence of statements that set up input objects and invoke methods on those objects. During its execution,

[2] Essentially, the Windows Subsystem for Android functions as an emulator.

```
public class GenericMarkupMenuItemRenderer extends ... {
  private String text;     private HashMap attributes;
  public void setAttributes(HashMap attr) { attributes = attr; }
  public void setText(String text) { this.text = text; }
  public void startRender(IMarkupWriter writer, IMenuItem item) {
    writer.begin(element);
    if (attributes != null && attributes.keySet() != null) {
      Iterator attrNames = attributes.keySet().iterator();
      while(attrNames.hasNext()) {
        String attrName = (String)attrNames.next();
        String attrValue = (String)attributes.get(attrName);
        writer.attribute(attrName, attrValue);
      }
    }
    ...
  }
  /* other details omitted */
}
```

Listing 1: Method `GenericMarkupMenuItemRenderer.startRender` from the BluePenguinMail web-based Email application.

```
GenericMarkupMenuItemRenderer renderer0 = new
      GenericMarkupMenuItemRenderer();
HashMap<Object, Object> hashMap0 = new HashMap<Object, Object>();
Object object0 = new Object();
hashMap0.put("framework", (Object) null);
renderer0.setAttributes(hashMap0);
NullWriter nullWriter0 = new NullWriter();
DefaultMenuItem defaultMenuItem0 = new DefaultMenuItem(object0);
renderer0.startRender(nullWriter0, defaultMenuItem0);
```

Listing 2: One unit test generated for method `writeByteSized`.

EvoSuite first randomly generates an initial generation of test suites and then iteratively evolves those suites. In each iteration, EvoSuite dynamically analyzes the execution of the current generation of test suites, calculates for each suite a fitness value that summarizes its usefulness for fulfilling its overall goals, and derives from the test suites with the highest fitness values the next generation of test suites via two operations, namely mutation and crossover. Here, the mutation operation involves adding new test cases to a test suite and modifying existing test cases from a test suite, which in turn involves adding, removing, or changing statements from the test cases; The crossover operation involves exchanging test cases from two test suites.

One important aspect in EvoSuite's implementation is that it installs a customized class loader named `InstrumentClassLoader` to replace the default one of the JVM, and the instrumentation performed by `InstrumentClassLoader` is essential for the overall effectiveness and efficiency of the tool [28]. For instance, `InstrumentClassLoader` heavily instruments all app classes loaded during the execution of a test both to log their traces at runtime and to reduce the overhead induced by mutation testing.

The evolutionary algorithm implemented by EvoSuite is only effective when its search space is continuous and monotonic with respect to the fitness function used. However, that is seldom the case when testing object-oriented programs, which often involves generating objects with complex structures as test inputs. In view of that, Lin et al. [25] proposed the EvoObj approach that extends the EvoSuite technique with a mechanism to effectively generate unit tests for hard-to-cover branches in programs under testing. Given a target branch within a method to cover, EvoObj first builds an object construction graph to capture the fields that a test covering the target branch may access and the dependence relations among those fields. Then, it synthesizes unit tests with instructions to instantiate the method's input objects, change the object states by assigning different values to the relevant fields based on the object construction graph, and invoke the method using the input objects. Finally, it reuses the evolutionary algorithm from EvoSuite to search for appropriate values to be used for setting the object fields. By breaking complex objects into collections of fields, EvoObj makes the input space of unit tests easier to navigate with an evolutionary algorithm.

III. EvoDroid In Action

EvoDroid has two major components as its cornerstones. One is the integration of Robolectric to enable automated unit test generation for Android, whose value can be confirmed once the tool is able to generate and run useful unit tests for Android apps. The other is the effective unit test suite generation technique for Java classes. In this section, we use class `GenericMarkupMenuItemRenderer` (shown in Listing 1) from the SF100 benchmark [29] (See Section V-A for more information about the benchmark) as an example to demonstrate EvoDroid's effectiveness and efficiency in class-level unit test suite generation.

The class is part of the BluePenguinMail web-based Email application and is responsible for rendering the generic markup menu items on the application's GUI. In particular, the class stores the attribute name and value pairs to be used in rendering menu items in a HashMap field named `attributes`. Taking an `IMarkupWriter` and an `IMenuItem` as the parameter, its method `startRender` applies all the attributes stored in field `arributes` when constructing the markup node for the menu item and sends the constructed markup element to the writer. The loop in the method body iterates through all the name and value pairs stored in field `attributes` and sends them to the writer. For a unit test to enter the loop, the render's field `attributes` should not be empty, and all the names and values stored in it should be convertible to strings. It, however, is none trivial for a test generation tool to construct such a non-empty hashmap and assign it to field `attributes`.

When applied to generate unit tests for method `startRender`, EvoObj always produced three unit tests that cover 51% of the method's instructions, with the average generation time for each test being around 0.6 minutes. Although EvoObj successfully identified `attributes` as a relevant field for a test to cover the method's **`while`** loop, and it also included invocations to method `setAttributes` in the generated tests to diversify the renderer's states, the hashmaps used in those tests were either empty or populated with references to `IMarkupWriter` and `GenericMarkupMenuItemRenderer` instances as the keys and values. In the former case, the **`if`** condition in the method body

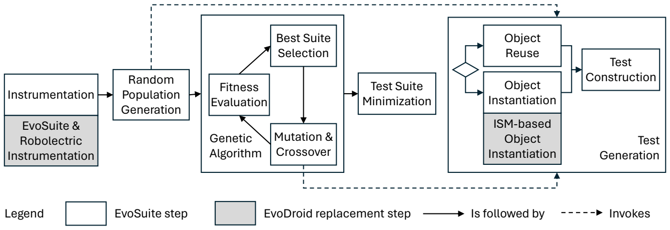

Figure 1: The main workflow of EvoDroid and EvoSuite.

shall evaluate to `false`, causing the execution to skip the `while` loop; In the latter case, the explicit type conversions in the loop body shall fail, causing the test to terminate prematurely. As a result, tests generated by EvoObj were not able to properly cover the loop body.

In contrast to that, when applied to generate unit tests for class `GenericMarkupMenuItemRenderer`, EvoDroid always produced four unit tests for method `startRender` that cover 85% of the method's instructions, with the average generation time for each test being around 0.3 minutes. For instance, Listing 2 gives one generated unit test that successfully exercised the whole `while` loop because it properly populated the renderer's field `attributes` with key and value references that are convertible to strings.

EvoDroid was considerably more effective and efficient than EvoObj for two main reasons. First, EvoObj and EvoDroid were designed to effectively generate high-quality unit test suites for methods and classes, respectively. On the one hand, class-level unit test suite generation makes it easier for EvoDroid to come across objects in more diversified and hopefully useful states, which increases the chance of generated tests exercising more program behaviors. On the other hand, class-level unit test suite generation also makes EvoDroid more sensitive to the differences among test input object states because the differences are more likely to cause distinct behaviors if the objects are used to test more methods, which increases the chance of EvoDroid retaining the useful objects during the genetic evolution of the unit tests. For example, it is inevitable for EvoDroid to construct `String` objects when testing the other methods (e.g., `setText`) in class `GenericMarkupMenuItemRenderer`, and such `String` objects can be easily reused by EvoDroid when populating the `attributes` hashmap. In contrast, since EvoObj has a narrow focus on fields accessed in method `startRender` and by the `while` loop, the chance it stands of constructing a `String` object is almost negligible. Second, although the analysis that EvoObj performs to construct the input objects is more sophisticated, it is rather expensive and has to be repeated on each target branch to be covered, which adds huge overhead to the unit test generation process. In contrast, EvoDroid conducts only a lightweight analysis, once on each method, for the same purpose. The analysis runs much faster and is sufficient to support the effective generation of unit test suites with high coverage.

IV. The EvoDroid Approach

Figure 1 shows the major steps involved in test generation with EvoSuite and the corresponding replacement steps EvoDroid introduces for program instrumentation and object instantiation. Correspondingly, we explain in this section how EvoDroid generates tests for methods expecting complex objects as the input and how the Robolectric framework is integrated into EvoDroid to support running Android-specific code on JVMs.

A. Construction of Complex Objects as Test Inputs

EvoDroid adopts EvoSuite's overall unit test generation process, but its object synthesis algorithm is inspired by EvoObj. There are two important differences between unit test suite generation in the two approaches. First, EvoObj generates test unit suites for methods, while EvoDroid does that for classes, which aligns better with EvoSuite's underlying idea of *whole test suite optimization* [26]. Second, EvoObj employs object construction graphs to guide the generation of test input objects, while EvoDroid constructs input structure maps for the same purpose. Compared with object construction graphs, input structure maps are much easier and less expensive to build but still contain sufficient information to steer the generation of useful unit tests.

In general, three groups of input variables are accessible in a method, namely the explicit parameters, if any; the implicit `this` parameter, if the method is an instance method; and the static fields defined in the containing and other classes. These input variables and the variables they directly or indirectly refer to via their fields determine the behaviors (including the output, if any) of a method. Therefore, to construct complex objects so as to drive a method under testing to exercise more behaviors, EvoDroid first statically analyzes the method's definition to gather information about which objects are used as the input to the method and which fields of the objects may influence the behaviors of the method. Note that the analysis reasonably ignores local variables defined and initialized inside the method since their values can always be derived from the aforementioned three groups of input variables.

EvoObj builds an object construction tree to abstract the input object states in terms of class fields that are relevant to a given target branch to be covered. In contrast to that, EvoDroid constructs for *each method* under testing an *input structure map*, where each key corresponds to a class whose instance(s) may be accessed during the execution of the method, while each value contains two collections of accessed fields of the class, one for instance fields and the other for static fields, respectively. In other words, each input structure map contains the structural information about the input objects for one method under testing, and the fields directly or indirectly accessible via these variables are grouped and recorded by their defining classes. For example, Figure 2 shows part of the input structure map for method `startRender` shown in Listing 1.

Key	Value
GenericMarkupMenuItemRenderer	({text: String, attributes: HashMap, …}, {…})
HashMap	({entrySet: Set<Map.Entry<K,V>>, size: int, …}, {…})
IMarkupWriter	({…}, {…})

Figure 2: Part of the input structure map for method `startRender` in Listing 1.

1) Construction of Input Structure Map

Given a method `m` from a class `C` to be processed, EvoDroid traverses the instructions in the method to gather information about which fields are accessed during the execution of the method. The traversal focuses on the objects directly or indirectly reachable from the input objects and the exact fields of those objects accessed by the method. Upon encountering an access to a field `f` of a reference `o`, e.g., in the form of `o.f`, EvoDroid will first resolve `o`'s class `co` and `f`'s type `tf`. Afterward, EvoDroid incorporates the structural information reflected by the access into the input structure map of the current method by first making sure a key of class `co` is contained in the map and then adding the field `f` with its type `tf` to the corresponding collection, depending on whether `f` is static or not. Note that all fields of class `C`, either defined in the class or inherited from its super-classes/interfaces, are always contained in the input structure maps for all methods from the same class. This is to ensure that every object created for class `C` in the generated tests has all its fields set up, which helps to diversify the `C` objects used in the generated tests.

To find out which objects and fields are accessed in other methods invoked by method `m`, EvoDroid *conservatively* follows all the possible invocation relationships and tracks the usage of those input objects across method boundaries, using the same strategy as described above. In particular, if a method invocation `o'.m'()` in method `m` can be dispatched to different implementations of method `m'`, EvoDroid conservatively processes all the overriding versions of method `m'` when constructing the input structure map for method `m`. In this way, the input structure map constructed for a method captures both the classes and the fields of all objects possibly accessed during the method's execution, whose values essentially determine the behaviors of the method. Afterward, generating proper input objects to exercise the method in different ways boils down to assigning appropriate values to those fields.

2) Test Generation Based on Input Structure Map

After constructing the input structure map for method `m`, EvoDroid generates unit tests for the method under the guidance of the input structure map. Each test contains statements both to prepare the input objects and to invoke the method, and the search algorithm adopted from EvoSuite will evolve the tests to achieve the overall testing goals.

When preparing the input objects, if an object is of a primitive type, EvoDroid reuses the process from EvoSuite to determine a value of the proper type for the object [26]. For an input object of a reference type, EvoDroid either constructs a new object (with a probability of 0.1 to reduce overhead) or reuses an existing object of a compatible type (with a probability of 0.9 to promote object reuse). To construct a new object of a reference type, EvoDroid first collects all methods that may return an object of the type, including the corresponding class's constructors, then picks one method `f'` from them, and finally recursively builds the inputs required by `f'` and invokes the method. Note that, in case that method `m` is non-static, EvoDroid considers the receiver of the method invocation as an (implicit) input object for the method.

After an object is obtained, EvoDroid generates statements to set all the non-final fields of the object that were recorded in method `m`'s input structural map. In particular, EvoDroid attempts to set a private or protected field by invoking a method and set a public field by either invoking a method or directly assigning a value to it. Here, EvoDroid applies a light-weight static analysis technique to gather the set of public methods defined in class `co` that modify the field `f` of the objects of the class and invokes only such methods in attempts to set the values of the field. If the field itself is of a reference type, an object of a compatible type will be reused or constructed recursively and then assigned to the field. Besides, EvoDroid will stochastically invoke state-changing methods to diversify the states of the objects before they are actually used to invoke a target method.

Two more things about the generation of unit tests are worth extra attention. First, all the generated objects are put into a pool and can be reused later as the input for method invocations in other tests. Second, to control the overall length of the statement sequences produced by this essentially recursive process, we set the upper bound of the sequence length to 40 and stop adding statements to a sequence once its length reaches this limit.

B. Integration of Robolectric *and* EvoSuite

As explained in Section II, both Robolectric and EvoSuite require their own customized class loader to function correctly. However, naively combining the two tools as black-box systems will not work because of how Java class loaders work in general. According to the JVM specification, the same class loader for loading a Java class `K` will be used to load the classes on which `K` depends, and class `K` is only aware of the classes loaded by the very class loader that loaded itself.

Correspondingly, having two active class loaders, from Robolectric and EvoSuite, respectively, in EvoDroid will cause two major issues. First, since each class loader is also responsible for instrumenting the classes it loads, a class will miss out on an important type of instrumentation no matter which class loader is used to load it. Second, we may end up loading a class twice, each time using one class loader, which may cause data fragmentation and other problems. For instance, EvoSuite employs a singleton class named `ExecutionTracer` to gather the execution traces of the generated test cases, and the class will be loaded twice if Robolectric and EvoSuite are integrated in a naive way, one time by `InstrumentClassLoader` when EvoSuite is running as the class is clearly part of the EvoSuite tool, and the other time by `SandboxClassLoader` since its methods will be invoked when the generated test cases are executed

```
class InstrumentationClassLoader{
  ClassLoader loader = ...;
  void injectClassLoader(SandboxClassLoader scl){ loader = scl; }
  void loadClass(String name){ loader.loadClass(name); }
  /* other details omitted */
}
class SandboxClassLoader{
  ClassLoader loader = /*default class loader*/;
  void loadClass(String name){
    ByteCode bc = retrieveClassByteCode(name);
    bc = sandboxInstrument(bc); bc = evosuiteInstrument(bc);
    loadBytecode(bc);
  }
  /* other details omitted */
}
```

Listing 3: Implementation of the two class loaders in pseudo-code.

in Robolectric's sandbox environment. As a result, two distinct instances of class ExecutionTracer may coincide, with different knowledge about the coverage achieved by the test generation process.

For instance, EvoSuite employs a singleton class named ExecutionTracer to gather the execution traces of the generated test cases, and the class will be loaded twice if Robolectric and EvoSuite are integrated in a naive way. On the one hand, class ExecutionTracer will be loaded by InstrumentClassLoader when EvoSuite is running as the class is clearly part of the EvoSuite tool. On the other hand, the class will also be loaded by SandboxClassLoader since its methods will be invoked when the generated test cases are executed in Robolectric's sandbox environment. As a result, two distinct instances of class ExecutionTracer may coincide, with different knowledge about the coverage achieved by the test generation process.

To overcome that problem, EvoDroid combines the class loaders from the two tools in the following way. First, we modify both SandboxClassLoader and InstrumentationClassLoader so that a class is always loaded via the same process as defined in method SandboxClassLoader.loadClass, no matter which loader is the one to initiate the class loading process. In particular, we revise class InstrumentClassLoader so that 1) its default class loader is replaced with an instance of SandboxClassLoader from the Robolectric environment and 2) its method loadClass simply delegates class loading to SandboxClassLoader.loadClass. Meanwhile, we also modify method SandboxClassLoader.loadClass so that, when loading a class, the method will a) read the bytecode of the class, b) perform the intrumentation originally implemented in Robolectric (see Section II) so that the class's code will execute inside Robolectric's sandbox, c) invoke via reflection the instrumentation logic originally implemented in EvoSuite to further instrument the class code so that the execution trace will be recorded during testing, and d) load the class code with both types of instrumentation into the Java virtual machine. In the revised implementation, while two types of instrumentation are still implemented in two class loaders, we are certain that both types of instrumentation are applied to all classes under testing and that all the classes loaded in this way are aware of each other. The pseudo-code in Listing 3 shows the main steps performed by each loader when loading a class.

To make sure both class loaders are properly configured at the very beginning of the unit test generation process, during the initialization of EvoDroid, we initialize the Robolectric environment, get a handle of its SandboxClassLoader object, and store the handle in EvoSuite's InstrumentClassLoader.

C. Support for Android Specific Environments

1) Running view-related Android code in the main thread

During test generation, the generated test cases are often executed in a different thread from the tool's main thread. Such a design is necessary because the generated test cases may crash or run into other problems, and running them in separate threads enables the tool to control the damage the test threads may cause and even kill those threads when necessary. The Android platform, however, stipulates that all view-related operations must be performed in an Android main thread. Hence, Robolectric especially devises a method named runOnMainThread to encapsulate the support for that requirement. The method takes a Runnable object as the argument and executes the run method of the Runnable object in an Android main thread mimicked by the framework.

2) Constructing meaningful GUI-level Android objects

To effectively test methods requiring GUI-level Android objects (e.g., of types Application, Activity, and Button) as parameters, we must be able to construct or obtain meaningful objects of those types. Since those objects are typically highly complex, constructing them by directly calling the corresponding constructors is seldom sufficient and often produces only trivial instances of the classes with limited usefulness in driving interesting test executions. In view of that, EvoDroid exploits class RobolectricTestRunner from Robolectric to launch the application and activities of the app under testing and stores one instance of each GUI-level Android object type as a globally shared resource. Later, whenever such an object is needed during test generation, the stored one can be used.

While such objects may be inappropriate for invoking certain methods, e.g., because they violate the methods' (explicit or implicit) preconditions, they are usually more complex and realistic than the ones randomly constructed from scratch and therefore are useful for exercising the code under testing in more diversified ways. A similar idea was explored by Jaygarl et al. to test object-oriented Java code [30] and by Arcuri et al. to test code that involves networking [31].

3) Accessing shared preferences

Operations on shared preferences often involve accessing the related XML files on Android. In the sandbox environment provided by Robolectric, such access to local files, however, is not allowed and may cause the execution of the corresponding tests to hang. To work around such a limitation, EvoDroid provides a customized, map-based implementation for the SharedPreference interface and replaces the original SharedPreference implementation with it. With the new implementation, reading from and writing to a particular preference is achieved by retrieving and storing the value associated with a key, i.e., the preference text, from and to the map, respectively.

D. Implementation

We have implemented the EvoDroid technique into a tool with the same name atop the EvoSuite tool. Regarding the object synthesis process, during unit test generation, EvoObj *stochastically* selects targets from branches that have not been covered yet and invokes its customized algorithm to generate objects as test inputs. The design is suitable for EvoObj since its analysis of the code under testing is rather heavy, and the design can help reduce the overhead introduced by its analysis. The analysis that EvoDroid performs, however, is much lighter and does not introduce much overhead. Hence, EvoDroid *always* delegates the generation of new tests from scratch to the process described in Section IV-A. Test generation via mutation and cross-over in EvoDroid remains the same as in EvoSuite.

V. Evaluation

We have applied EvoDroid to generate unit tests for both Android and Java code. Based on the generation results, we address the following research questions:

RQ1: How effective and efficient is EvoDroid in Android test generation? In RQ1, we evaluate the test generation results produced by EvoDroid on open-source Android apps, and we compare the results with that achieved by Stoat [32], a state-of-the-art GUI test generation tool for Android, as all Android test generation tools that we have access to produce GUI tests[3].

RQ2: How does EvoDroid compare to state-of-the-art tools for Java unit test suite generation? Recall that EvoDroid has two major components as its cornerstones. One is the integration of Robolectric to enable automated unit test generation for Android, and the other is the effective unit test suite generation at the class level. In RQ2, we investigate the usefulness of the second component by comparing it to its counterpart algorithm implemented in EvoObj. Previous work has shown that EvoObj outperforms EvoSuite in generating unit tests for Java code [25]. Here, we did not compare EvoDroid with EvoObj in terms of their bug-finding capability mainly because, due to the lack of high-quality, automatic test oracle, it is challenging to determine whether a failing unit test reveals a bug or is caused by invalid input.

RQ3: How useful is EvoDroid in unit testing production-level Android code? Considering that the open-source Android apps may be smaller in size and less complex, we investigate in RQ3 whether EvoDroid can generate unit tests to exercise a significant percentage of production-level Android code.

RQ4: To what extent the generated unit tests can help developers detect regression bugs in real-world Android apps? Previous work has shown that automatically generated unit tests can effectively detect regression bugs in programs [26]. Because of that, we study in RQ4 the usefulness of the unit test suites generated by EvoDroid for detecting regression bugs in real-world Android apps.

A. Subjects

To answer RQ1, we used as subjects all the 25 open source Android apps with Stoat test generation results from the Prefest benchmark of Android apps [34]. We determined to use these apps as subjects in our experiments for two major reasons. First, the Prefest benchmark contains a highly diversified collection of open-source Android apps, which were initially gathered from multiple research studies on Android apps, came from a wide range of categories, and vary significantly in size. Therefore, experimental results produced on those apps are more likely to represent how EvoDroid performs in different situations when applied in practice. Second, Stoat has been applied to generate GUI tests for these apps, and the corresponding experimental results are readily available for the apps, which provides us with a natural benchmark for assessing the effectiveness and efficiency of EvoDroid in testing Android apps. Given that EvoDroid failed to run on one of the 25 apps because of the conflicts between different versions of Gradle and other libraries required by the project itself and EvoDroid, we use the test generation results on the remaining 24 apps to address research question RQ1. Table I lists, for each app, a unique id (ID), the name (App), the version, and the total numbers of classes (#C) and lines of code (#L).

The comparison between EvoDroid and EvoObj to answer RQ2 was based on the SF100 dataset. The SF100 dataset [29] is a standard benchmark consisting of 100 open-source Java projects, and it has been used to compare EvoObj and EvoSuite experimentally [25].

Since EvoObj generates unit tests at the method level, while EvoDroid does that at the class level, to make sure we compare the two tools in a fair way, we applied both tools to generate unit tests for selected methods and classes from the SF100 dataset and compared their generation results separately. At the method level, since EvoObj has been applied in a previous experiment to generate unit tests for 2500 randomly selected methods from the SF100 dataset [25], we applied EvoDroid to generate unit tests for the same 2500 methods. At the class level, we refrained from using all the classes in this study for time reasons. Instead, we randomly selected a collection of 750 classes, each with at least two public methods (including public constructors), from the SF100 dataset and used both tools to generate tests for *all* methods in each class. The 750 classes had in total 10577 public methods.

[3] TCTATCG [33] only generates unit tests for Android apps at the component level and is not available for download.

TABLE I: Subject apps and the test case generation results achieved by EvoDroid and Stoat on the apps.

ID	App	Version	#C	#L	EvoDroid				Stoat		
					#Test	#L$_c$	#L$_u$	T (min)	#L$_c$	#L$_u$	T (min)
A01	a2dpvolume-master	2.13.0.4	427	3756	39	1534 (40.8%)	590 (15.7%)	21.9	1480	0	120.0
A02	AlwaysOnDisplayAmoled	0.9.5.2	506	3163	64	1522 (48.1%)	630 (19.9%)	15.8	1391	0	120.0
A03	AmazeFileManager	3.3.0-rc1	2441	17067	453	4256 (24.9%)	1375 (8.1%)	462.7	3511	0	120.0
A04	android-money-manager-ex	2018.11.15.987	4117	21041	373	6813 (32.4%)	3055 (14.5%)	210.0	5617	0	120.0
A05	AntennaPod	1.7.1	1561	9463	88	2520 (26.6%)	0 (0.0%)	222.1	2739	859	120.0
A06	APhotoManager	0.7.2.181027	1647	9683	263	4596 (47.5%)	1616 (16.7%)	172.6	4326	0	120.0
A07	apps-android-commons-master	2.9	2235	10001	249	2721 (27.2%)	51 (0.5%)	56.9	2700	0	120.0
A08	forecastie	1.6.2	256	1761	31	1172 (66.6%)	595 (33.8%)	30.2	998	0	120.0
A09	good-weather	4.4-1	399	2464	298	1711 (69.4%)	1037 (42.1%)	43.9	1460	31	120.0
A10	hn-android-master	5.0	524	2256	90	1567 (69.5%)	95 (4.2%)	46.7	1559	0	120.0
A11	KISS	3.7.2	760	4554	130	2611 (57.3%)	1035 (22.7%)	156.9	2207	0	120.0
A12	materialistic	3.2	1765	7082	123	3265 (46.1%)	493 (7.0%)	175.7	3131	0	120.0
A13	nanoConverter-master1	0.7.87	71	996	28	425 (42.7%)	328 (32.9%)	1.0	374	0	120.0
A14	Notepad-102	1.0.2	216	1757	27	977 (55.6%)	144 (8.2%)	36.0	925	0	120.0
A15	OpenBikeSharing	1.10.0	196	1223	30	768 (62.8%)	114 (9.3%)	26.1	748	0	120.0
A16	opensudoku	2.5.2	645	3674	108	1873 (51.0%)	381 (10.4%)	15.7	1771	0	120.0
A17	radiocells-scanner-android	0.8.18	1158	7424	144	2943 (39.6%)	1118 (15.1%)	107.1	2675	0	120.0
A18	runnerup	1.90.1	2277	15609	1	3883 (24.9%)	0 (0.0%)	2.9	3457	0	120.0
A19	Signal-Android	4.33.0	9824	43727	1121	11985 (27.4%)	0 (0.0%)	1891.4	10057	0	120.0
A20	SuntimesWidget	0.13.4	2481	12996	220	5889 (45.3%)	7425 (57.1%)	195.2	5038	28	120.0
A21	Timber	1.6	2101	11792	165	3343 (28.3%)	962 (8.2%)	187.3	3199	0	120.0
A22	uhabits	1.7.9	975	4445	92	2808 (63.2%)	220 (4.9%)	180.9	2614	0	120.0
A23	vanilla-master	1.0.80	1472	10733	78	5924 (55.2%)	3527 (32.9%)	166.2	4866	0	120.0
A24	wikipedia-r	2.7.237	6354	24069	397	10816 (44.9%)	1611 (6.7%)	547.8	10117	0	120.0
Overall			44408	230736	4612	85922 (37.2%)	26402 (11.4%)	4973.0	76960	918	2880.0

TABLE II: Production-level Android code repositories and the test suite generation results achieved by EvoDroid on them.

ID	#Class	#Line$_{tot}$	#Test	#Line$_{cov}$	%Cov	T (min)
Repo1	169	2717	1014	2188	80.5%	17.1
Repo2	455	2433	6474	1811	74.4%	23.4
Repo3	459	11806	3673	4870	41.3%	29.1
Repo4	634	16610	4072	5574	33.6%	44.0
Repo5	2014	37542	24226	14093	37.5%	108.9
Repo6	6416	126793	50579	58741	46.3%	368.2
Repo7	7412	82807	33345	26506	32.0%	324.6
Repo8	10640	175621	47875	37742	21.5%	476.6
Repo9	12206	352405	79442	98993	28.1%	456.8
Overall	40405	808734	250700	250518	31.0%	1848.7

We used nine Android code repositories from Douyin Co., Ltd., as the subjects to address RQ3. For confidentiality reasons, we refer to the code repositories as Repo1, Repo2, etc., hereafter. Table II lists all the nine repositories in increasing order of the number of classes contained in them. For each repository, the table gives its ID and the numbers of classes (#Class) and lines of code (#Line$_{tot}$) in it, respectively.

To address RQ4, we applied EvoDroid to two popular Android apps developed at Douyin Co., Ltd. Table III summarizes the basic information about the apps. For each app, the table gives its ID, its number of monthly active users (MAUs) in millions, and the total number of merge requests checked with the generated tests (#MR). In particular, App1 is an online video-sharing social media platform, and App2 is an online video editing system. These two apps are from different problem domains, aim for different users, and have different levels of popularity, which helps increase the representativeness of the experimental results as EvoDroid's behaviors in practical use.

TABLE III: Using the generated unit tests to detect regression bugs in production-level Android apps.

ID	MAU	#MR	#Issues		
			Detected	Confirmed	Fixed
App1	> 600m	2096	46	46	36
App2	> 130m	433	35	35	34
Overall		2529	81	81	70

B. Experimental Protocol

All the experiments on EvoDroid and EvoObj ran on a cloud infrastructure, with each run of a tool using exclusively one virtual machine instance, configured to use one core of an Intel Xeon Platinum 8362 CPU@2.80GHz, 32GB of RAM, Debian 5.4.143, and Oracle's JDK 1.8.0. In view of the randomness involved in test generation with both EvoDroid and EvoObj, we repeat the experiments of a tool under each configuration for five times and use the average results achieved for the analysis and comparison.

To address RQ1, we simply apply EvoDroid to generate tests for all classes in each app and compare the results with that produced by Stoat.

To address RQ2, we apply both EvoDroid and EvoObj to generate unit tests for the selected methods and classes and compare their generation results. Here, applying EvoObj at the method or class level and applying EvoDroid at the class level are straightforward. When applying EvoDroid to generate unit tests for a particular method, we first identify the method's containing class, then apply EvoDroid to generate unit tests for the identified class, and finally measure the code coverage achieved by the produced tests with respect to the method. Note that, in such settings, EvoDroid's running time obviously includes the time spent on generating unit tests for other methods, so the two tools' execution times should not be compared directly.

To address RQ3, we apply EvoDroid to generate unit tests for all classes in each repository and assess the results.

To address RQ4, we first apply EvoDroid to generate unit tests for classes from the base version of each app and then run the produced tests against the updated versions of the same app. Since a passing test for the base version app may fail for various reasons when executed on the updated version app, but not all such failures reveal regression faults in the app, we introduced a group of rules to automatically filter out failures that are less likely caused by regression faults and report only the remaining failed tests to the developers. For instance, failures causing exceptions like `UnsatisfiedLinkError` are often due to discrepancies between the actual and shadow implementations of the Android environment and should be filtered out. Similarly, failures causing exceptions in packages like `org.robolectric.res` usually are caused by limitations in the Robolectric framework and should be ignored too. We then invite the developers to manually inspect the reported cases to determine whether they uncover any regression bugs.

C. *Metrics*

Each run of a test generation tool produces a collection of tests for the target code. We use the line coverage achieved by the resultant tests, with respect to the total number of lines of code contained in the target code unit, be it a method, a class, or a group of classes, to measure the effectiveness of the test generation process. Meanwhile, to evaluate the efficiency of a test generation process, we utilize the commonly used metric T that measures the wall clock generation time. Since we let EvoDroid run to its end in our experiments, but Stoat had exactly 120 minutes to produce the results listed in Table [34], we also compare the average line coverage achieved by the two tools *per minute*. We measure EvoDroid's effectiveness for bug detection in terms of the number of regression bugs discovered, confirmed, and fixed.

D. *Experimental Results*

In this section, we report on the experimental results and answer the research questions.

1) *RQ1: Unit Test Generation for Open-Source Android Apps*

Table I lists, for each open-source Android app, the number of tests EvoDroid generated (#Test), the number of LoCs covered by EvoDroid/Stoat generated tests (#L$_c$), the number of LoCs *uniquely* covered by EvoDroid/Stoat generated tests (#L$_u$), and the wall-clock time in minutes that EvoDroid/Stoat used to generate the tests (T), respectively.

On the one hand, EvoDroid generated in total 4612 unit tests that covered 85922 lines of the source code in the 24 apps, achieving a code coverage of 37.2%, while the test cases generated by Stoat covered 76960 lines of code. A Wilcoxon signed rank test [35] confirms that the difference between the achieved coverage is statistically significant at $\alpha = 0.05$ ($p = 2.3e - 6$), and Cohen's d effect size [36], i.e., a measure that estimates the magnitude of the observed differences, is 0.28, which suggests the difference is small. On the other hand, 26402 lines of code, or 11.4% of the total lines of code, were only covered by EvoDroid but not by Stoat. Overall, such results suggest that EvoDroid is effective in generating unit tests for Android apps and complementary to GUI test generation tools like Stoat.

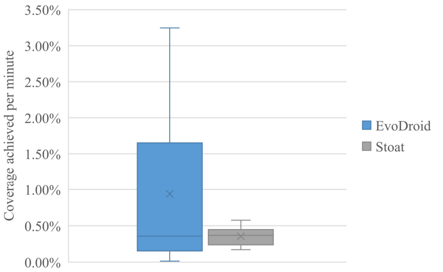

Figure 3: Distribution of the line coverage achieved by EvoDroid and Stoat per minute.

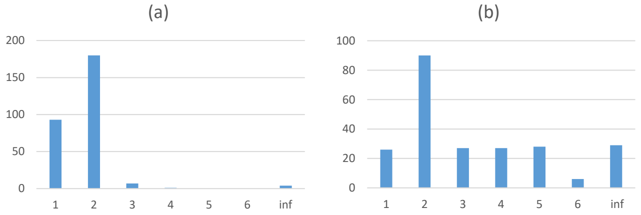

Figure 4: Distribution of the ratio a/b, where a and b are the line coverage achieved by EvoDroid and EvoObj on individual methods and classes, respectively ($a \neq 0 \wedge b \neq 0 \wedge a \neq b$).

Figure 3 plots the distribution of line coverage achieved by EvoDroid and Stoat per minute over various apps. The figure demonstrates that EvoDroid is more efficient than Stoat regarding the line coverage achieved per minute.

Answer to RQ1: EvoDroid is effective and efficient in generating unit tests for open-source Android apps, and it complements the Stoat GUI test generation tool.

2) *RQ2: Unit Test Generation for Java Code*

At the method level, EvoDroid and EvoObj achieved 25.7% and 19.7% line coverage on the 2500 methods, respectively. On 361 methods, EvoDroid and EvoObj generated at least one test case and achieved different coverage. The histogram in Figure 4(a) shows the distribution of the ratio a/b, where a and b are the line coverage achieved by EvoDroid and EvoObj, respectively, across the 361 methods. In the figure, a bar at position x of height y ($x \leq 6$) indicates that, for y methods, the ratio of line coverage achieved by EvoDroid to that achieved by EvoObj was in range [x-1, x), and the bar at position inf of height y indicates that, for y methods, that ratio was greater than or equal to 6.

At the class level, EvoDroid and EvoObj generated at least one test on 619 of 750 classes, containing in total 6657 methods. The two tools achieved 52.4% and 19.4% line coverage on those classes, respectively, and the coverage they achieved was different on 218 classes. The histogram in Figure 4(b) shows the distribution of the ratio a/b, where a and b are the line coverage achieved by EvoDroid and EvoObj, respectively, across the 218 classes.

Figure 4 clearly demonstrates that, while EvoObj was highly effective on a significant number of methods, probably because EvoObj's highly focused analysis of their branches and relevant object states is really necessary to find out what kind of input objects are needed to execute those branches, EvoDroid is more effective in test suite generation overall, likely because it strikes a better balance between the costs and effectiveness of its analysis on object structures. Besides, the overall higher line coverage achieved by EvoDroid also confirms the rationale that, generating a unit test suite for all methods in a class instead of for individual methods makes it easier for test generation tools to come across objects in more diversified states, which, when used in new tests, are more likely to trigger executions covering different program paths.

Note that the test generation results achieved by EvoObj in our experiments are significantly worse than those reported in the EvoObj paper [25] (but they are still better than what EvoSuite achieved). In fact, despite running the experiments in EvoObj's replication package multiple times, we could not reproduce the coverage reported earlier or achieve similar results. Given that we consulted the authors of the EvoObj paper, and they could not reproduce the previous results either, we decided to use the new results we obtained in our experiments to address this RQ.

> *Answer to RQ2:* EvoDroid *was more effective in terms of line coverage achieved than* EvoObj *in unit test generation at both the method and the class level.*

3) RQ3: Test Generation for Production-Level Android

Table II lists for each code repository the number of tests generated (#Test), the number of lines of code covered by the tests (#Line$_{cov}$), the coverage achieved (%Cov), and the amount of time in minutes EvoDroid needed to generate tests for the repository. It is clear that the classes in these repositories are considerably larger (with an average size of ≈20.01 LoC/class), and therefore most likely also more complex, than those in the open-source Android apps (with an average size of ≈5.20LoC/class). Nevertheless, EvoDroid generated in 1,848.7 minutes 250,700 tests for the repositories, covering 250,518 lines of code and achieving an overall line coverage of 31.0%, which is comparable to the line coverage that EvoDroid achieved on open-source Android apps in RQ1. Given that the overall test generation time for the repositories is only 205.4 minutes per repository, which is in the same order of magnitude as the average test generation time for open-source Android apps (i.e., 207.2 minutes per app), we consider the test generation process on these repositories efficient.

> *Answer to RQ3:* EvoDroid *is both effective and efficient in unit testing production-level Android code.*

4) RQ4: Detecting Regression Bugs in Production-Level Android Code

Table III reports for each production-level app the number of regression bugs detected by the generated unit tests as well as the numbers of such bugs manually confirmed and fixed by the corresponding developers. In our experiments, unit tests generated by EvoDroid helped developers detect 81 bugs in the two Android apps. All 81 bugs were later manually confirmed to be real regression bugs, and 70 of them have been fixed. The developers decided not to fix the remaining 11 bugs mainly for two reasons. First, the bug is of low priority because the chance of the bug being triggered in practice is slim, and the negative impact of the bug is small. Second, the risk that a fix will introduce new, more severe bugs is too high. This often happens when, e.g., the bug is located in legacy code that is no longer actively maintained.

> *Answer to RQ4: Tests generated by* EvoDroid *effectively revealed regression bugs in production-level Android code.*

E. Discussion

The experimental results presented in Section V-D highlight the importance of properly balancing various features and/or objectives when devising automated test generation techniques and tools that are practically beneficial. In this section, we discuss some of these features and objectives and the implications of our work for practitioners and researchers.

Scalability vs. effectiveness. EvoObj was highly effective on open-source Java projects that were relatively smaller and less complex but had limited scalability and, therefore, applicability to production-level code, which was a major reason why we devised EvoDroid. Interestingly, by striving to make EvoDroid a more scalable technique, we actually improved the overall effectiveness of EvoDroid as well. Such results suggest that, compared with effectiveness, scalability probably should carry equal, if not greater, weight when devising new and practical test generation techniques and tools.

Unit vs. GUI testing. There are a number of automated GUI testing techniques and tools for Android in the literature nowadays. Research on automated unit test generation for Android, however, is very limited. Given the widely acknowledged importance of unit testing in software quality assurance and the clearly imbalanced investment in unit and GUI testing for Android apps, we call on researchers and practitioners to pay more attention to and provide better technical and tool support for automated Android unit testing.

Property-based vs. regression testing. The oracle problem has been a long-standing challenge in automated unit testing. Instead of deriving the assertions for and detecting the defects in the same version of an Android app, EvoDroid automatically generates unit tests mainly for the purpose of regression testing, as was done in previous [26] and recent [37] work. The experimental results in both works show that automatically generated tests complement manually prepared tests in regression testing and can effectively reveal real bugs in the updated versions of programs under consideration. However, we still relied on human expertise in designing the rules for filtering out failed tests that were unlikely caused by regression faults. Developing reliable techniques to automatically differentiate such tests is an interesting direction to advance automated unit test generation in general and for Android code in particular.

F. Threats to Validity

One obvious threat to internal validity comes from possible mistakes in our implementation of EvoDroid. To mitigate this threat, we carefully reviewed our code and scripts to ensure their correctness before conducting the final experiments.

We mitigated the threats to external validity by conducting the experimental evaluation on subject Android apps from various sources and of different natures. We also applied EvoDroid to production-level Android apps so as to ascertain that it is effective on real-world Android apps.

VI. Related Work

A. Search-Based Test Generation

Search-based test generation was pioneered by Miller et al. [38]. In that work, Miller et al. proposed a search-based technique to generate test cases for functions with parameters of primitive types. Following their work, to handle the challenges involved in preparing appropriate, complex objects to test object-oriented programs, researchers have proposed various techniques to better represent the generated tests [39], to diversify the constructed object instances [40], and to more effectively test object-oriented containers [41].

Recently, Fraser et al. developed the EvoSuite technique that incorporates the aforementioned techniques and uses a search-based approach to evolve unit test suites [26]. EvoSuite starts by randomly generating an initial generation of unit test suites, which is then evolved with an evolutionary search toward satisfying a coverage criterion. The finally generated test suite is minimized before being returned to the user. EvoSuite employs not only standard evolutionary algorithms like the standard genetic algorithm and the cellular genetic algorithm but also evolutionary algorithms tailored for test generation like many-objective sorting algorithm (MOSA) [42] and DynaMOSA [43] to achieve high code coverage. EvoSuite has been extended to effectively handle the environmental dependencies [44], the network communications [31], and the implicit branching due to situations such as throwing runtime exceptions in unbranched control blocks [45]. Such extensions lay the foundation for the practice of test generation on enterprise-level applications.

The effectiveness of EvoSuite relies on the assumption that the search space is overall continuous and monotonic for fitness measurements, which, however, is seldom true with programs requiring object inputs. Lin et al. proposed the EvoObj approach that builds atop EvoSuite to address this problem [25]. EvoObj first builds through static analysis an object construction graph for each coverage target to capture the fields of the required input objects and their relationships. Then, it generates tests consisting of statements preparing the objects and invoking the corresponding methods under the guidance of the graph, and those tests are evolved by EvoSuite to satisfy the given coverage criterion. EvoDroid employs static analysis to build an input structure map for each method and utilizes the map to guide input generation for the method. Compared with that implemented in EvoObj, static analysis in EvoDroid is effective, much lighter, and less frequently invoked, which makes EvoDroid more effective and efficient in generating tests at the class level.

Interested readers are referred to survey papers by Phil McMinn [46] and Khari et al. [47] for more details.

B. Automated Test Generation for Android

Most existing techniques for automated Android test generation work at the GUI level. Monkey [48] is a test framework released and maintained by Google. It generates and sends pseudo-random streams of user/system events to running Android apps, and it has been applied to automatically identify defects in apps. Su et al. developed Stoat [32], an automated GUI test generation technique for Android. Stoat combines model-based stochastic exploration of app GUIs and random injection of system-level events to maximize the code coverage achieved by the generated tests. Stoat has been shown to outperform prior Android testing tools [32]. Ape [6] leverages runtime information during testing to improve the precision of models constructed for Android apps and utilizes the models to effectively generate GUI tests for the apps.

Efforts have also been made to generate unit tests for Android in the past two years. Particularly, JUnitTestGen [49] aims to generate unit tests for APIs of the Android platform, while TCTATCG [33] aims to create unit tests for Android components (e.g., activities, services, and intents). In contrast, EvoDroid is the first class-level unit test generation technique for Android code written by mobile developers.

Robolectric [24] provides a unit testing framework for Android by simulating the Android execution environment on Java virtual machines. By waiving the requirement that Android tests must be executed on Android devices or emulators, and therefore avoiding the overhead of packaging and deploying the tests, Robolectric greatly reduces the overall costs for Android unit testing. The integration of Robolectric into EvoDroid also enables Android unit test generation to benefit from techniques like dynamic analysis which are only accessible in Java, but not Android, environments.

VII. Conclusion

In this paper, we propose the EvoDroid approach to the automated generation of unit test suites for Android apps. EvoDroid generates high-quality unit test suites at the class level, and it employs a novel algorithm to effectively synthesize complex objects as test inputs based on input structure maps. Besides, it integrates the Robolectric framework to support running Android-specific tests on Java virtual machines. Experimental evaluation results show that EvoDroid is effective and efficient in generation unit tests for Android.

Acknowledgment

Yu Pei's work is supported in part by the Hong Kong Polytechnic University Fund under Grants P0051205 and P0051074.

References

[1] "appbrain," https://www.appbrain.com/stats/number-of-android-apps, last accessed May. 30, 2023.

[2] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[3] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon, "Exploiting the saturation effect in automatic random testing of Android applications," in *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2015, pp. 33–43.

[4] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*, 2020, pp. 1–12.

[5] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[6] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[7] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 115–127.

[8] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[9] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-based energy testing of android," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1119–1130.

[10] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.

[11] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 42–53.

[12] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: a deep learning-based approach to automated black-box android app testing," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1070–1073.

[13] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[14] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of Android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[15] T. B. de Assis, A. A. Menegassi, and A. T. Endo, "Amplifying tests for cross-platform apps through test patterns," in *SEKE*, 2019, pp. 55–74.

[16] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 595–605.

[17] ——, "Amplifying tests to validate exception handling code: An extended study in the mobile application domain," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–28, 2014.

[18] M. Pan, Y. Lu, Y. Pei, T. Zhang, and X. Li, "Preference-wise testing of android apps via test amplification," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, feb 2023.

[19] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: automatic maintenance of GUI test scripts for evolving mobile applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 161–171.

[20] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-based test script maintenance for android apps," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2018, pp. 215–225.

[21] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "GUI-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, 2020.

[22] T. Xu, M. Pan, Y. Pei, G. Li, X. Zeng, T. Zhang, Y. Deng, and X. Li, "Guider: Gui structure and vision co-guided test script repair for android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 191–203.

[23] "Androidjunitrunner," https://developer.android.com/training/testing/junit-runner, last accessed Feb. 15, 2023.

[24] "Robolectric," http://robolectric.org/, last accessed Oct. 10, 2022.

[25] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-based seed object synthesis for search-based unit testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1068–1080.

[26] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *2011 11th International Conference on Quality Software*. IEEE, 2011, pp. 31–40.

[27] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation," in *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*. Springer, 2017, pp. 33–48.

[28] Y. Li and G. Fraser, "Bytecode testability transformation," in *Search Based Software Engineering: Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings 3*. Springer, 2011, pp. 237–251.

[29] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 178–188.

[30] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "Ocat: Object capture-based automated testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 159–170.

[31] A. Arcuri, G. Fraser, and J. P. Galeotti, "Generating tcp/udp network data for automated unit test generation," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 155–165.

[32] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 245–256.

[33] J. Cao, H. Huang, and F. Liu, "Android unit test case generation based on the strategy of multi-dimensional coverage," in *2021 IEEE 7th International Conference on Cloud Computing and Intelligent Systems (CCIS)*, 2021, pp. 114–121.

[34] M. Pan, Y. Lu, Y. Pei, T. Zhang, and X. Li, "Preference-wise testing of android apps via test amplification," *ACM Transactions on Software Engineering and Methodology*, 2022.

[35] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[36] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Routledge, 1988.

[37] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based unit test generation at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 173–184. [Online]. Available: https://doi.org/10.1145/3663529.3663838

[38] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, no. 3, pp. 223–226, 1976.

[39] S. Wappler and J. Wegener, "Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm," in *2006 IEEE International Conference on Evolutionary Computation*, 2006, pp. 851–858.

[40] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2015.

[41] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Inf. Sci.*, vol. 178, no. 15, p. 3075–3095, aug 2008.

[42] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[43] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, “An empirical evaluation of evolutionary algorithms for unit test suite generation,” *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[44] A. Arcuri, G. Fraser, and J. P. Galeotti, “Automated unit test generation for classes with environment dependencies,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 79–90.

[45] P. Derakhshanfar and X. Devroey, “Basic block coverage for unit test generation at the sbst 2022 tool competition,” in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, 2022, pp. 37–38.

[46] P. McMinn, “Search-based software testing: Past, present and future,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163.

[47] M. Khari and P. Kumar, “An extensive evaluation of search-based software testing: A review,” *Soft Comput.*, vol. 23, no. 6, p. 1933–1946, mar 2019.

[48] “Ui/application exerciser monkey,” https://developer.android.com/studio/test/monkey, last accessed Oct. 10, 2022.

[49] X. Sun, X. Chen, Y. Zhao, P. Liu, J. Grundy, and L. Li, “Mining android api usage to generate unit test cases for pinpointing compatibility issues,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3551349.3561151