EvoDroid: Segmented Evolutionary Testing of Android Apps

Riyadh Mahmood
Computer Science Dept.
George Mason University
Fairfax, VA, USA
rmahmoo2@gmu.edu

Nariman Mirzaei
Computer Science Dept.
George Mason University
Fairfax, VA, USA
nmirzaei@gmu.edu

Sam Malek
Computer Science Dept.
George Mason University
Fairfax, VA, USA
smalek@gmu.edu

ABSTRACT

Proliferation of Android devices and apps has created a demand for applicable automated software testing techniques. Prior research has primarily focused on either unit or GUI testing of Android apps, but not their end-to-end system testing in a systematic manner. We present EvoDroid, an evolutionary approach for system testing of Android apps. EvoDroid overcomes a key shortcoming of using evolutionary techniques for system testing, i.e., the inability to pass on genetic makeup of good individuals in the search. To that end, EvoDroid combines two novel techniques: (1) an Android-specific program analysis technique that identifies the segments of the code amenable to be searched independently, and (2) an evolutionary algorithm that given information of such segments performs a step-wise search for test cases reaching deep into the code. Our experiments have corroborated EvoDroid's ability to achieve significantly higher code coverage than existing Android testing tools.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Android, Evolutionary Testing, Program Analysis

1. INTRODUCTION

Mobile app markets have created a fundamental shift in the way software is delivered to the consumers. The benefits of this software supply model are plenty, including the ability to rapidly and effectively deploy, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, this paradigm has leveled the playing field, allowing small entrepreneurs to compete head-to-head with prominent software development companies.

Platforms, such as Android, that have embraced this model of provisioning apps have seen an explosive growth in popularity. This paradigm, however, has given rise to a new set of concerns. Small organizations do not have the resources to sufficiently test their products, thereby defective apps are made available to the consumers of these markets. These defects are exploited with malicious intent compromising the integrity and availability of the apps and devices on which they are deployed. This is nowhere more evident than in Google Play, a popular Android app market, where numerous security attacks have been attributed to vulnerable apps [35]. The situation is likely to exacerbate given that mobile apps are poised to become more complex and ubiquitous, as mobile computing is still in its infancy.

Automated testing of Android apps is impeded by the fact that they are built using an *application development framework (ADF)*. ADF allows the programmers to extend the base functionality of the platform using a well-defined API. ADF also provides a container to manage the lifecycle of components comprising an app and facilitates the communication among them. As a result, unlike a traditional monolithic software system, an Android app consists of code snippets that engage one another using the ADF's sophisticated event delivery facilities. This hinders automated testing, as the app's control flow frequently interleaves with the ADF. At the same time, reliance on a common ADF provides a level of consistency in the implementation logic of apps that can be exploited for automating the test activities, as illustrated in this paper.

The state-of-practice in automated system testing of Android apps is random testing. Android Monkey [3] is the industry's de facto standard that generates purely random tests. It provides a brute-force mechanism that usually achieves shallow code coverage. Several recent approaches [18–20, 29, 32, 38] have aimed to improve Android testing practices. Most notably and closely related to our work is Dynodroid [32], which employs certain heuristics to improve the number of inputs and events necessary to reach comparable code coverage as that of Monkey.

Since prior research has not employed evolutionary testing and given that it has shown to be very effective for event driven software [27], [30], we set out to develop the first evolutionary testing framework targeted at Android, called *EvoDroid*. Evolutionary testing is a form of search-based testing, where an *individual* corresponds to a test case, and a *population* comprised of many individuals is evolved according to certain heuristics to maximize the code coverage. The most notable contribution of EvoDroid is its ability to overcome the common shortcoming of using evolutionary techniques for system testing. Evolutionary testing techniques [22,30,36,37] are typically limited to local or unit testing, as for system testing, they are not able to promote the genetic makeup of good individuals during the search.

EvoDroid overcomes this challenge by leveraging the knowledge of how Android ADF specifies and constrains the way apps can be

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635896

built. It uses this platform-specific knowledge to statically analyze the app and infer a model of its behavior. The model captures (1) the dependencies among the code snippets comprising the app, and (2) the entry points of the app (i.e., places in the code that the app receives external inputs). The inferred model allows the evolutionary search to determine how the individuals should be crossed over to pass on their genetic makeup to future generations. The search for test cases reaching deep into the code occurs in *segments*, i.e., sections of the code that can be searched independently. Since a key concern in search-based testing is the execution time of the algorithm, EvoDroid is built to run the tests in parallel on Android emulators deployed on the cloud, thus achieving several orders of magnitude improvement in execution time.

The remainder of this paper is organized as follows. Section 2 provides a background on Android. Section 3 outlines an illustrative example that is used to describe our research. Section 4 motivates the research problem using the illustrative example. Section 5 provides an overview of our approach, while Sections 6 to 8 provide the details and results. The paper concludes with a summary of the related research in Section 9 and a discussion of our future work in Section 10.

2. ANDROID BACKGROUND

The Google Android framework includes a full Linux operating system based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. It is based on the Dalvik Virtual Machine (DVM) [6] for executing programs written in Java. Android also comes with an *application development framework* (ADF), which provides an API for application development and includes services for building GUI applications, data access, and other component types. The framework is designed to simplify the reuse and integration of components.

Android apps are built using a mandatory XML *manifest* file. The manifest file values are bound to the application at compile time. This file provides essential information to an Android platform for managing the life cycle of an application. Examples of the kinds of information included in a manifest file are descriptions of the app's components among other architectural and configuration properties. Components can be one of the following types: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*. An Activity is a screen that is presented to the user and contains a set of layouts (e.g., *LinearLayout* that organizes items within the screen horizontally or vertically). The layouts contain GUI controls, known as view widgets (e.g., *TextView* for viewing text and *EditText* for text inputs). The layouts and its controls are typically described in a configuration XML file with each layout and control having a unique identifier. A Service is a component that runs in the background and performs long running tasks, such as playing music. Unlike an Activity, a Service does not present the user with a screen for interaction. A Content Provider manages structured data stored on the file system or database, such as contact information. A Broadcast Receiver responds to system wide announcement messages, such as the screen has turned off or the battery is low.

Activities, Services, and Broadcast Receivers are activated via *Intent* messages. An Intent message is an event for an action to be performed along with the data that supports that action. Intent messaging allows for late run-time binding between components, where the calls are not explicit in the code, rather made possible through Android's messaging service. All major components, including Activity and Service, follow pre-specified lifecycles [1] managed by the ADF. The lifecycle event handlers are called by the ADF and play an important role in our research as explained later.

3. ILLUSTRATIVE EXAMPLE

We use a simple Android app, called Expense Reporting System (ERS), to illustrate our research. The ERS app allows users to submit expense report from their Android devices. As shown in Figure 1, ERS provides two use cases that allow the user to create two types of report: *quick report* and *itemized report*.

When *quick report* is chosen, the user enters the expense item name and the amount, and subsequently presented with the summary screen. The user can choose to submit or quit the application on the summary screen.

The *itemized report* option presents the user with the option to enter the number of line items by tapping the plus and minus buttons. When *next* is tapped, the application prompts the user to enter the expense name and amount. This screen is repeated until all line items have been entered. Once all items are entered, the user is presented with a summary screen with the line items, their amount, and the total amount. The user can again choose to submit or quit the application at this time.

4. RESEARCH CHALLENGE

Achieving high code coverage in Android apps, such as ERS, requires trying out a large number of sequences of events such as user interactions and system notifications. Our research is inspired by prior work [30] that has shown evolutionary testing to be effective when sequences of method invocation are important for obtaining high code coverage. However, application of evolutionary testing has been mostly limited to the unit level [22,30,36,37], as when applied at the system level, it cannot effectively promote the genetic makeup of good individuals in the search.

Figure 2a illustrates the shortcoming of applying an evolutionary approach for system testing of ERS. Here, we have two *individuals* in iteration 1 of the search. In this representation, an individual is comprised of two types of genes: *input genes* (e.g., values entered in text fields) and *event genes* (e.g., clicked buttons). The test case specified in an individual is executed from the left most gene to the right most gene. In essence, each individual is a test script.

Using the screenshots of ERS in Figure 1, we can see that the two individuals in iteration 1 of Figure 2a represent reasonable tests, as each covers a different part of the app. For system testing, we would need to build on these tests to reach deeper into the code. The problem with this representation, however, is that there is no effective approach to pass on the genetic make up of these individuals to the next generations. For instance, from Figure 2a, we can see that the result of a crossover between the two individuals in it-

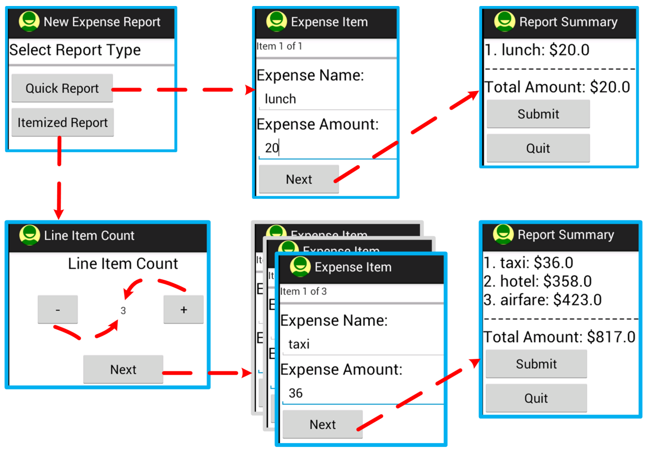


Figure 1: Expense Report System (ERS).

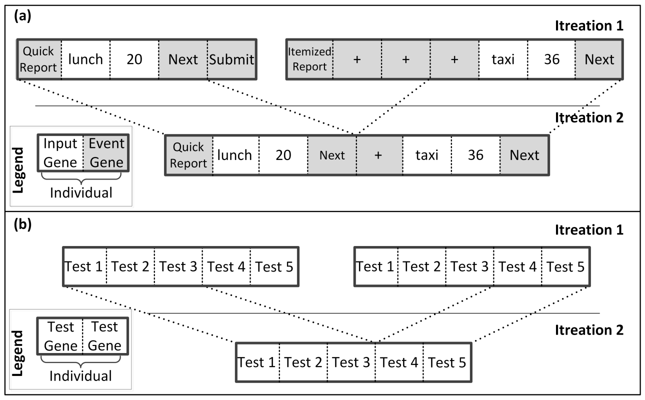


Figure 2: Existing evolutionary testing techniques: (a) a representation where the individual represents a test case, and (b) a representation where the individual represents a test suite

eration 1 is a new individual in iteration 2 that does not preserve the genetic makeup of either parents in any meaningful way. In fact, using the screenshots of ERS in Figure 1, we can see that the tests cannot even be executed. There are two issues that contribute to this: (1) The crossover strategy does not consider which input and action genes are coupled to one another. For instance, the genes "lunch", "20", and "Next" are coupled with one another, as only together they can exercise the *Expense Item* screen. (2) The crossover strategy mixes genes from two different execution paths in the system. Thus, it produces a test that is likely to be either not executable or inferior to both its parents. In evolutionary search, the inability to promote and pass on the genetic makeup of good individuals to the next generations is highly detrimental to its effectiveness.

To overcome the issues with this representation, prior approaches [17,27,28,34] use evolutionary algorithm in conjunction with GUI crawling techniques. One such approach, called EXSYST [27], represents *test suites* as individuals and *tests* as genes, as depicted in Figure 2b. This approach generates tests that correspond to random walks on the GUI model. An individual is comprised of many random tests, i.e., each gene of the individual corresponds to a system test. EXSYST evolves the suites of tests to minimize the number of tests and maximize coverage. However, the probability of a single gene (test) achieving deep coverage remains the same as in the case of random testing. The overall coverage is no better than the *initial population* (randomly generated tests), as the evolutionary algorithm is mainly used to minimize the number of tests.

EvoDroid is the first evolutionary testing approach for system testing of Android apps. To that end, it had to overcome the conceptual challenges of using evolutionary techniques for system testing. EvoDroid achieves this through a unique representation of individuals and a set of heuristics that allow the algorithm to maintain and promote individuals with good genetic makeup that reach deep into the code.

5. APPROACH OVERVIEW

The overall EvoDroid framework is shown in Figure 3. The input is an Android app's source code. From the source code, EvoDroid extracts two types of models, representing the app's external interfaces and internal behaviors, to automatically generate the tests: *Interface Model* (IM) and *Call Graph Model* (CGM). The models are automatically extracted by analyzing the app's code.

IM provides a representation of the app's external interfaces and in particular ways in which an app can be exercised, e.g., the inputs and events available on various screens to generate tests that are valid for those screens. A partial representation of IM for the ERS is shown in Figure 4b. EvoDroid uses the IM to determine the structure of individuals (tests), i.e., the input and event genes that are coupled together.

CGM is an extended representation of the app's call graph. A typical call graph shows the explicit method call relationships. We augment that with information about the implicit call relationships caused by events (messages). An example of CGM for the ERS is shown in Figure 5. A particular use case (e.g., *quick report* or *itemized report* from Figure 1) follows a certain path through the CGM. EvoDroid uses CGM to (1) determine the parts of the code that can be searched independently, i.e., *segments*, and (2) evaluate the fitness (quality) of different test cases, based on the paths they cover through the CGM, thus guiding the search.

Using these two models, EvoDroid employs a step-wise evolutionary test generation algorithm, which we call *segmented evolutionary testing*. It aims to find test cases covering as many unique CGM paths from the starting node of an app to all its leaf nodes. In doing so, it logically breaks up each path into segments. It uses heuristics to search for a set of inputs and sequence of events to incrementally cover the segments. By carefully composing the test cases covering each segment into system test cases covering an entire path in the CGM, EvoDroid is able to promote the genetic makeup of good individuals in the search.

EvoDroid executes the automatically generated test cases in parallel, possibly on the cloud, to address scalability issues. The test cases are evaluated based on a fitness function that rewards code coverage and uniqueness of the covered path.

The focus of EvoDroid is on generating test cases that maximize code coverage, not on whether the test cases have passed or failed. We acknowledge that automatically generating test oracles is a significant challenge. This has been and continues to be the focus of many research efforts. Currently, we collect two types of results from the execution of tests: any exceptions that may indicate certain software faults as well as code coverage information.

Section 6 describes the models used for testing, while Section 7 presents the details of EvoDroid.

6. APPS MODELS EXTRACTION

EvoDroid needs three types of information about the app under test for automatically generating test cases: (1) the genes comprising a valid individual, e.g., determining the input fields and GUI controls that should be paired up to have a valid test case for an Activity (which as you may recall from Section 2 represents a GUI screen), (2) the app's segments, i.e., parts of the app that can be searched separately to avoid the crossovers issues described earlier, and (3) the fitness value of different test cases. We developed Android-specific program analysis techniques to infer two models that can provide EvoDroid with this information.

6.1 Interface Model

The Interface Model (IM) provides information about all of the input interfaces of an app, such as the widgets and input fields belonging to an Activity. It also includes information about the application- and system-level Intents handled by each Activity. The IM is obtained by combining and correlating the information contained in the configuration files and meta-data included in Android APK (such as Android Manifest and layout XML files).

First we list all the Android components (e.g., Activities, Services) comprising an app with the help of information found in the Manifest file. Afterwards, for each Activity we parse the corresponding layout file. An example of such layout file for *ExpenseItemActivity* is shown in Figure 4a. It is quite straightforward to ob-

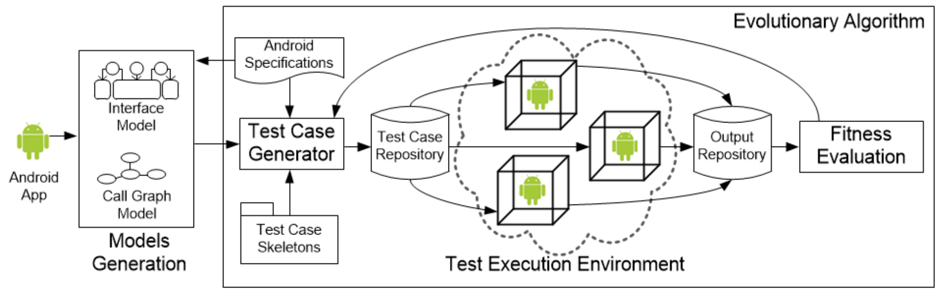


Figure 3: EvoDroid Framework.

tain all information on each screen, such as widget type, name, and identifier from this XML document to generate the IM. Figure 4b depicts the IM for the ERS Activities. We use the information captured in IM to determine the structure (genes) of individuals for testing each component of the app.

6.2 Call Graph Model

The Call Graph Model (CGM) contains a set of *connected* call graphs capturing the different possible invocation sequences within a given application. We use MoDisco [13], an open source program analysis tool, to extract the app's call graph. However, since Android is an event driven environment, MoDisco generates *disconnected* call graphs for each app. Figure 5 shows ERS's CGM. As described later, we have extended MoDisco to infer the dashed lines to create a fully connected graph.

The root node of each call graph snippet is a method that no other part of the application explicitly invokes. There are two types of root nodes:

1. *Inter-component root nodes*: these root nodes represent methods in a component that handle events generated by other components or Android framework itself, e.g., an Activity generating a *StartActivity* event that results in another Activity's *onCreate()* method to be called, or the Android framework sending a *Resume* event that results in an Activity's *onResume()* method to be invoked.

2. *Intra-component root nodes*: these root nodes correspond to events that are internal to a component. For example, a Button on an Activity has a *Click* event associated with it. This event is handled by a class within the same Activity that implements the *OnClickListener* interface and overrides the *onClick()* method. These sorts of callback handlers are also root nodes, as they are called by the Android framework.

The inter-component root nodes are the logical break points for segments, and the inputs received at these nodes form the structure of individuals for the corresponding segments. We can determine the structure of this input using the IM. On the other hand, the intra-component root nodes do not mark a new segment, as they do not result in the execution to move to a different component (e.g., different screen), and thus are not susceptible to the crossover problem.

Finally, for EvoDroid to generate tests and to determine their fitness, it needs the CGM to be fully connected. To that end, we have extended MoDisco with an Android-specific program analysis capability to infer the relationships among the disconnected nodes of the call graph. As depicted in Figure 5, we start with the *onCreate()* root node of the *main* Activity, which we know from Android's ADF specification to be the starting point of all apps. We then identify the Intent events and their recipients, as well as GUI controls and their event handlers, to link the different parts of

```
<LinearLayout xmlns:android="http://schemas.android
      .com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
    ...
    <EditText
        android:id="@+id/expenseNameId"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <requestFocus />
    </EditText>
    <EditText
        android:id="@+id/expenseAmoundId"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <requestFocus />
    </EditText>
    <Button
        android:id="@+id/nextBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Next" />
    ...
</LinearLayout>
```

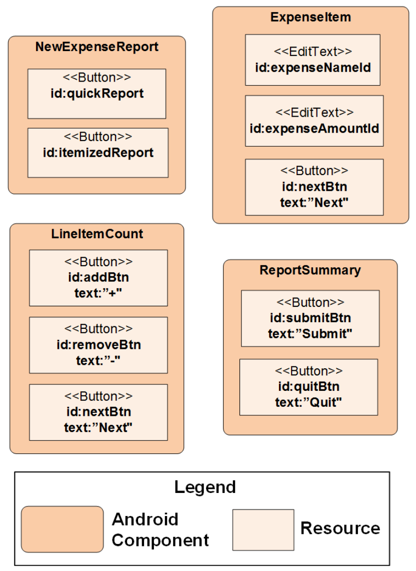
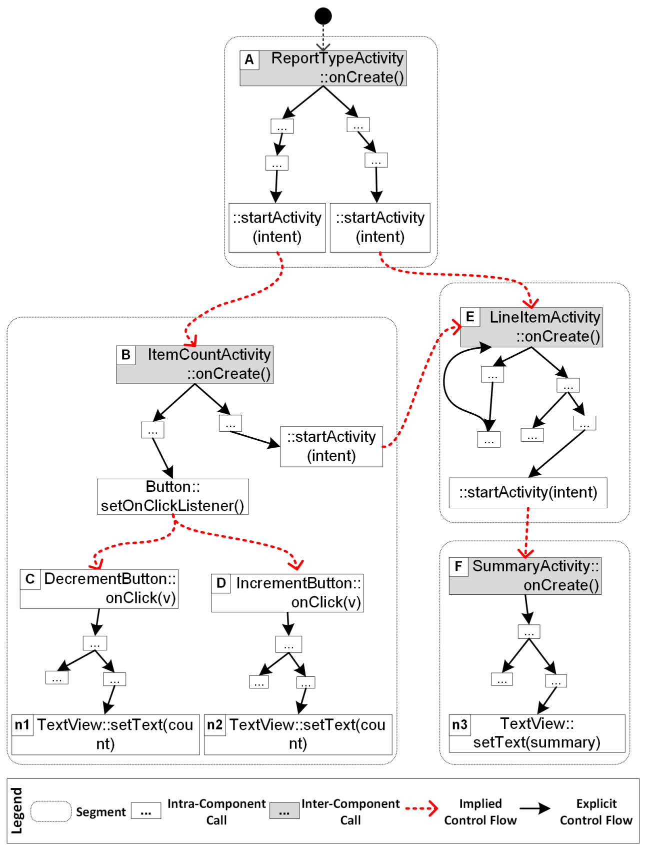

ExpenseItem
<<EditText>> id:expenseNameId
<<EditText>> id:expenseAmountId
<<Button>> id:nextBtn text:"Next"
NewExpenseReport
<<Button>> id:quickReport
<<Button>> id:itemizedReport
LineItemCount
<<Button>> id:addBtn text:"+"
<<Button>> id:removeBtn text:"-"
<<Button>> id:nextBtn text:"Next"
ReportSummary
<<Button>> id:submitBtn text:"Submit"
<<Button>> id:quitBtn text:"Quit"
Legend
Android Component
Resource

Figure 4: (a) Parts of the layout file for ExpenseItem Activity, (b) ERS Interface Model


A ReportTypeActivity ::onCreate()
::startActivity (intent)
::startActivity (intent)
B ItemCountActivity ::onCreate()
::startActivity (intent)
Button:: setOnClickListener()
C DecrementButton:: onClick(v)
D IncrementButton:: onClick(v)
n1 TextView::setText(count)
n2 TextView::setText(count)
E LineItemActivity ::onCreate()
::startActivity(intent)
F SummaryActivity:: onCreate()
n3 TextView:: setText(summary)
Legend
Segment
Intra-Component Call
Inter-Component Call
Implied Control Flow
Explicit Control Flow
Figure 5: Part of ERS's Call Graph Model

the call graph and arrive at the final CGM. We know that the links would have to be to other root nodes, and achieved through sending of Intent events. The links formed as a result of this inference tell us the implied control flow as depicted with the dashed lines in Figure 5. In the case of inter-component events, the sender of Intent identifies its handler as one of the Intent's parameters. In the case of intra-component events, the root node responsible for handling that event is registered as a callback method with the sender. For instance, a button's *onClick()* method is registered with an object that implements the OnClickListener to receive a callback when the button is clicked. Examples of this in Figure 5 are *DecrementButton* and *IncrementButton* that are registered with *ItemCountActivity*, which implements the *OnClickListener* interface. As the call graph snippets are linked and connected, they are traversed in a similar fashion to arrive at the final connected CGM for the app.

7. EVODROID

The goal of EvoDroid is to find a set of test cases that maximize code coverage. This is encoded as covering as many unique paths from the starting node of the CGM to its leaf nodes. In the context of ERS, depicted in Figure 5, it is to find test cases from node A to leaf nodes $n1$, $n2$, and $n3$. For example, possible paths in this graph are $A \rightarrow B \rightarrow C \rightarrow n1$ and $A \rightarrow B \rightarrow E \rightarrow F \rightarrow n3$. The former involves two segments, while the latter involves four segments.

For each such path in Figure 5, EvoDroid starts from the beginning node and searches for test cases that can reach the leaf nodes. Each test case is represented as an individual in EvoDroid and its genes are the app inputs and the sequence of events. Unlike any prior approach, EvoDroid takes each path in the CGM, breaks it into segments, and runs the evolutionary search for each segment separately. Accordingly, the evolutionary process described here is repeated for each segment along each path in the CGM.

For each segment in each path, a population with a configurable number of individuals is generated. The evolutionary process is continued until all of the paths and their segments are covered or a configurable threshold (e.g., time limit, certain level of code coverage, number of total test cases, etc.) is reached. The search is abandoned for a segment, and potentially a path, if the coverage is not improved after a configurable number of generations. This ensures the search does not waste resources on genes that cannot be further improved; it also prevents the search from getting stuck in infinite loops when there are cycles in the path. Fitness is measured based on how close an individual gets to reach the next segment as well the uniqueness of the covered path. With each iteration, EvoDroid breeds new individuals by crossing over current individuals selected with likelihood proportional to their fitness value, and then mutates them (e.g., changes some of the input values or events).

The *ideal* individuals from each segment are saved. An ideal individual is a test that covers the entire segment and reaches the root node of another segment. An ideal individual from the previous segment is prepended to the genes of a new individual for the next generation, as described further in the next section. Essentially the test cases gradually build on the solutions found for the prior segments to build up to a system test case. A segment may also optionally be skipped if it was covered while attempting to cover another segment. For example, in Figure 5, since the segment $E \rightarrow F$ is shared in the following two paths $A \rightarrow B \rightarrow E \rightarrow F \rightarrow n3$ and $A \rightarrow E \rightarrow F \rightarrow n3$, it would only need to be evolved once (assuming ideal individuals were found the first time). Similarly, if while evolving $A \rightarrow B$, the algorithm inadvertently reaches $A \rightarrow E$, those ideal individuals are saved and EvoDroid may optionally skip solving $A \rightarrow E$.

The maximum number of individuals or test cases executed in the search process can be calculated as follows:

$$T = \sum_{i=1}^{|path|} Seg_i \times gen_{seg_i} \times pop_{gen_{seg_i}} \quad (1)$$

where $|path|$ is the number of unique paths (from the starting node to the leaf nodes) in CGM, *seg* is the number of segments for each path, *gen* is the number of generations per segment, and *pop* is the population or the number of individuals per generation.

The remainder of this section describes the details of EvoDroid.

7.1 Representation

The models from Section 6 are used to determine the structure of genes for each segment. IM tells us the inputs, their data type (such as integer, double etc.), the number of GUI elements (such as buttons), and/or system events relevant to the current segment.

An individual is represented as a vector shown in Figure 6a. Here, *previous segment* corresponds to the genes of an ideal individual from the previous segment, *Input [1..n]* corresponds to specific input values from the current segment, and *Event [1..m]* corresponds to the sequence of possible user actions or system events from the current segment. Each index in the vector contains a gene. The previous segment is a recursive relationship.

The number of input genes is fixed, as we only need to change the input values, i.e., mutate the existing input genes. The number of event genes is variable to handle the situations in which unexplored parts of the application require a certain number of button clicks or certain sequence of events. For instance, in the *Line Item Count* screen from Figure 1, the *plus* button must be clicked more than the *minus* button and before clicking the *next* button to be able to reach the next screen. We execute the test case specified in an individual from the left most gene to the right most gene including all previous segment genes, essentially traversing a path in the CGM.

7.2 Crossover

The first step in creating a new individual for the next generation is crossover. This process selects two individuals from the current population and creates a new individual by mixing their genetic makeup. EvoDroid uses a multi-point probabilistic crossover strategy. There is at least one, and potentially multiple, crossover points between the two selected individuals.

The segment crossover probability is calculated as follows:

$$p(c) = \frac{1}{e^{(s-c)}} \tag{2}$$

where e is a configurable constant to achieve a decay factor, s is the index of the current segment being searched, and c is the index of a prior segment between 1 and s. The probability is *1.0* for the current segment, that is to say when $c = s$. This exponential decay function ensures that the earlier genetic makeup is not changed frequently, while leaving the possibility open to find individuals that may explore new areas of the search space.

The crossover point for the current segment can be at any gene index and at most the length of the smaller of the two individuals. We only allow one crossover for the current segment, as this is sufficient to create variability in the new individual. Figure 6c shows the crossover steps for a pair of parent individuals in *segment 3* of ERS. The newly created individual inherits part of the genetic makeup of the parents.

The previous segments are treated separately from the current segment and the probability function p(c) dictates the chance of crossover in each segment. There can potentially be a crossover at each of the previous segments, but we only allow swapping of the entire ideal individual for each segment, not in the middle of a previous ideal individual. Figures 6b and c show how ideal individuals found in prior segments are used to arrive at the parent individuals in Figure 6c. Here the new individual in Figure 6c inherits the previous segment individual from the left parent. If the probability function *p(c)* had dictated otherwise, it would have been inherited from the parent on the right.

This crossover strategy aims to preserve the genetic makeup of the solutions found for earlier segments, as we only allow the crossover to use the complete ideal individual for a given segment. Any ideal individual from that segment can be substituted, as they are all solutions for that segment. The previous segments for the new individual in Figure 6c share the same path (as the evolutionary process is applied within the context of a path), thus the structure of the individuals at each previous segment line up properly.

Note that this crossover strategy does not provide any guarantees that the input values and events satisfying an earlier segment in a path will be able to satisfy later segments in that path. For example, solving a particular constraint in segment *3* may require a specific value to have been entered in segment *1*. Indeed, the objective of the search is to find such combinations. The evolutionary search, guided by heuristics embedded in the fitness function, naturally weeds out sub-optimal tests. In addition, since we save many ideal individuals for each segment, each with different input/event genes, EvoDroid is quite effective at eventually discovering individuals that solve the entire path.

7.3 Mutation

Mutation changes parts of the genetic makeup of the newly created individual. Only the current segment genes are mutated with a probability threshold that is configurable. We mutate both input and event genes with several *creation*, *transformation* and *remove* operations.

The first type of mutation is done to the input genes of an individual. The creation of a numerical input includes boundary values, random, special/interesting values such as the number zero. For a string input, we generate purely random, uniformly distributed characters from the alphabet of a certain length, or *null*. Transformation operations for inputs include random value of same primitive data type, bit-flipping, arithmetic operations, and binary space reduction between boundary values. Removal operation for inputs is not applicable; they are included as *null* instead.

The second type of mutation is done to the event genes. The creation operator simply creates an event from the list of valid events specified in the IM. The number of added events is random with a minimum of one and a configurable upper threshold. Transformation operations for events include swapping event gene indexes, changing one event to another, and inserting a new event at a random index. Removal operation for events removes one or more event genes. The length of the overall individual can change as a result. Figure 6d show the mutation of a single gene to create a final unique individual that is different from both parents.

7.4 Fitness

A key aspect of evolutionary algorithms is the notion of fitness. In each generation, individuals are assessed for their fitness with respect to the search objective to be selected to pass on their genes. The fitness value ranges from 0 to 1. EvoDroid considers two factors when assessing the fitness of individuals. The first is the distance traveled (number of nodes covered between segments) to reach the next segment, and the second is the uniqueness of the path covered compared to the other individuals in the same generation. The fitness of an individual i is determined as follows:

$$f(i) = \left(\frac{x}{n}\right) + u(i) \tag{3}$$

where x is the number of covered nodes in the path to the destination segment, n is the total number of nodes in the path to the destination segment, and $u(i)$ is the uniqueness function of the individual as follows:

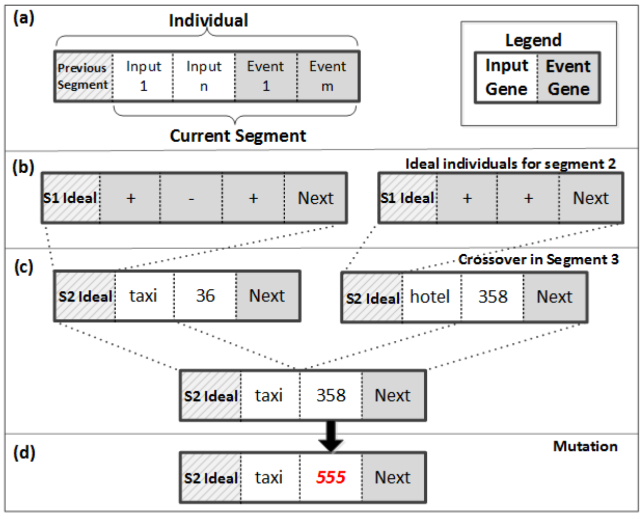


Figure 6: EvoDroid's (a) representation of individual, (b) ideal individuals from segment 2, (c) crossover steps for creating an individual in the 3rd segment, and (d) mutation

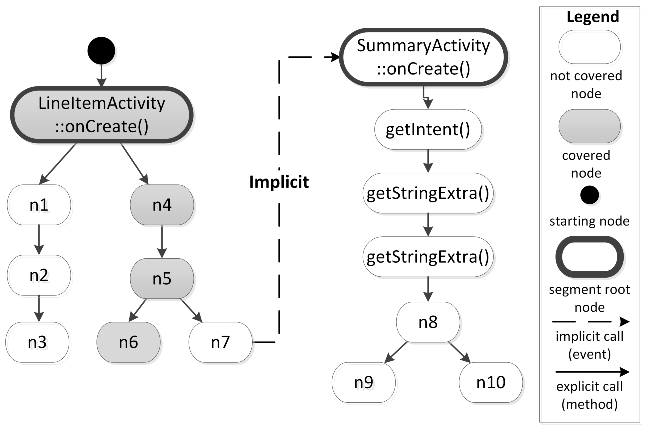


Figure 7: Fitness evaluation

$$u(i) = \left(1 - \left(\frac{x}{n}\right)\right) \times \sum_{k=1}^{l} \left(\frac{unique(r_k)}{l+k}\right) \tag{4}$$

where r_k is the covered node at index k in the path covered by the individual, and $unique(r_k)$ is 1 if the covered node at index k is unique compared to other individuals' coverage at the same index, and 0 otherwise, and l is the length of the path that this individual has covered.

When an individual for a given segment covers the entire segment path, we identify it as an *ideal* individual for that segment with a fitness score of 1. Of course, this means that there can be multiple individuals per generation that are ideal for a segment.

For illustration of how fitness is calculated, consider the hypothetical example depicted in Figure 7. It shows that the total distance (number of nodes) to reach *SummaryActivity* from *LineItemActivity* is 5. When the first individual executes, the shaded nodes are marked as covered by that individual. It covers 3 out of the 5 nodes along the path to segment root node, so it gets a distance score of $x/n = 3/5 = 0.6$, a uniqueness score of $u(i) = 0.4 \times 1/(4+1) + 1/(4+2) + 1/(4+3) + 1/(4+4) = 0.25$ as the entire path is unique at this time, for a total fitness score of $f(i) = 0.85$.

If another individual test case is executed but covers the path with nodes *LineItemActivity* $\rightarrow n1 \rightarrow n2 \rightarrow n3$, it would get a distance score of $x/n = 1/5 = 0.2$, and additionally a uniqueness score. The length of the path this individual covered is 4 and all but the first node are unique (i.e., $n1$, $n2$, $n3$), so the uniqueness score is $u(i) = .8 \times (1/(4+2) + 1/(4+3) + 1/(4+4)) = 0.34$. The total fitness score for this individual would be $f(i) = 0.2 + 0.34 = 0.54$. Although the individual did not cover much of the path to the destination segment node, it is awarded a fractional score as it may discover a new area of uncovered code.

Note that the formulation of eq. 3 and 4 ensures that the uniqueness score alone never makes the value of fitness function to be 1 without reaching the destination. This prevents an individual to be labeled ideal without first reaching the destination. Finally, a configurable number of test cases with the highest scores are directly copied to the future generations without any changes, to ensure the individuals with the best genetic makeup remain in the population.

8. EVALUATION

We evaluated EvoDroid on a large number of apps with varying characteristics. The goal of our evaluation was twofold: (1) compare the EvoDroid code coverage against the prior solutions, and (2) characterize its benefits and shortcomings.

8.1 Experiment Environment

Evolutionary testing requires the execution of a large number of tests. This is especially challenging in the case of the Android emulator [2], as it is known to be slow even when running on workstations with the latest processors and abundant memory. To mitigate this issue, we have developed a novel technique to execute the tests in parallel, possibly on the cloud, which makes it suitable for use by small as well as large organizations.

We set up an instance of Amazon EC2 virtual server running Windows Server 2008, and configured it with Java SDK, Android SDK, Android Virtual Device, and a custom test execution manager engine developed by us. For each test, the test execution manager launches the emulator, installs the app, sets up and executes the test. It is also responsible for persisting all of the results, along with the log and monitored data, to an output repository. A virtual machine image was created from the above instance to be replicated on demand. With this, we were able to scale in near-linear time and cut down on the execution time. We report the results for both extremes: when the test cases are executed in sequence using a single processor, and when they execute completely in parallel.

We have implemented EvoDroid using ECJ [9], a prominent evolutionary computing framework. In the experiments, we used EMMA [10] to monitor for code coverage, and all of the test cases were in Robotium [15] format. Our implementation and evaluation artifacts are available from [11].

8.2 Experiment Setup

We compare EvoDroid with Android Monkey [3] and Dynodroid [32] in terms of code coverage and execution time. Android Monkey is developed by Google and represents the state-of-the-practice in automated testing of Android apps. It sends random inputs and events to the app under test. Dynodroid [32] is a recently published work from researchers at Georgia Tech that uses a smaller number of inputs and events than Monkey for reaching similar coverage. We are not able to compare directly with EXSYST [27] and EvoSuite [28], as they are not targeted for Android. We do not compare against [38] as that is for model generation only, while EvoDroid creates models and performs a step-wise segmented evolutionary search.

At first blush it may seem unreasonable to compare EvoDroid, a whitebox testing approach, against Android Monkey and Dynodroid, which are blackbox and greybox testing approaches, respectively. However, there are two reasons that makes this comparison relevant. First, most Android apps can be reverse engineered using one of the existing tools (see [5,7,8,16]) to obtain the source code necessary for whitebox testing. Therefore, our approach could be used for testing almost all Android apps. Second, in the evaluation of any stochastic search algorithm, it is desirable to compare the results of the algorithm against the unbiased random sample of the solution space. We believe the comparison against Monkey and Dynodroid helps us in that vein. It should be noted that unlike EvoDroid, Monkey and Dynodroid are not designed to run in a distributed manner, and neither tools are configurable to run for a specific amount of time.

To achieve a fair comparison with Android Monkey and Dynodroid, however, we had to allot each approach similar number of events. The events for Android Monkey and Dynodroid are similar to the genes in EvoDroid. Since there is no one-to-one mapping, we ran EvoDroid first, and then mapped the total number of generated tests to Monkey and Dynodroid events. Thus, the number of events allotted for running Monkey and Dynodroid varied as a function of the number of test cases executed for EvoDroid as follows: $max(g) \times t$, where g is the maximum number of genes allowed for

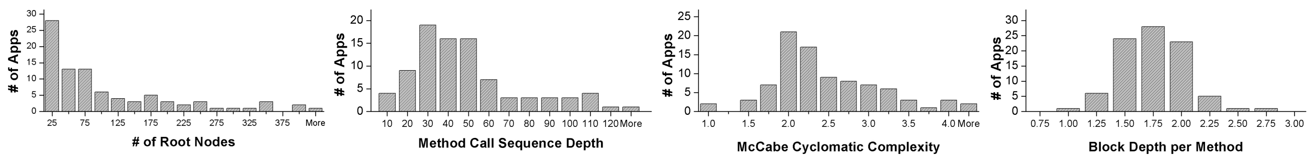


Figure 8: Android complexity metrics distribution from a random sample of 100 apps

test cases in EvoDroid and t is the number of test cases executed for a given app.

In all experiment scenarios, for each segment in each path, we used a maximum of 10 generations of 10 individuals with a maximum of 10 genes. The number of maximum generations along with the coverage of all segments served as the terminating conditions. Euler's constant ($e = 2.718$) was used as the crossover decay number in eq. 2, and during the mutation phase each gene had a 20% chance of mutation.

To evaluate EvoDroid, two sets of experiments were performed. The first on apps developed by independent parties from an open source repository, and the second on synthetic apps. The synthetic apps helped us benchmark EvoDroid's characteristics in a controlled setting.

8.3 Open Source Apps

We selected 10 open source apps to evaluate the line coverage between EvoDroid, Monkey, and Dynodroid. We were not able to run Dynodroid on two of the subject apps, and thus we are not able to report on those. As shown in Table 1, EvoDroid consistently achieves significantly higher coverage than both Monkey and Dynodroid. On average EvoDroid achieves 47% and 27% higher coverage than Monkey and Dynodroid, respectively.

The generation of test oracles is outside the scope of our work, nevertheless we collected information about unhandled exceptions, which allowed us to detect several defects in these apps. For instance, we found several cases of unhandled *number format exception* in Tipster, TippyTipper, and Bites that were due to either leaving the input fields empty, clicking a button that clears the input field followed by clicking a button that would operate on the inputs, or simply putting a string that could not be converted to a number. As another example, we found a defect in Bites, an app for finding and sharing food recipes, in which an unhandled *index out of bounds exception* would be raised when editing recipes without adding the ingredients list first.

Some of the reasons for not achieving complete coverage are unsupported emulator functions, such as *camera*, as well as spawning asynchronous tasks that may fail, not finish by the time the test finishes, and thus not get included in the coverage results. Other reasons include code for handling external events, such as receiving a text message, dependence on other apps, such as calendars and contacts lists, and custom exception classes that are not encountered or thrown. Additionally, some of the applications contained dead code or test code that was not reachable, thus the generated EvoDroid model would not be fully connected. Indeed, in many of these apps achieving 100% coverage is not possible, regardless of the technique.

The limitations of emulator, peculiarities in the third-party apps, and incomplete models made it very difficult to assess the characteristics of EvoDroid independently. In other words, it was not clear whether the observed accuracy and performance was due to the aforementioned issues, and thus an orthogonal concern, or due to the fundamental limitations in EvoDroid's approach to test generation. We, therefore, complemented our evaluation on real apps with a benchmark using synthetic apps, as discussed next.

8.4 Synthetic Benchmark Apps

To control the characteristics of the subjects (i.e., apps under test), we developed an Android app generator that synthesizes apps with different levels of complexity for our experiments. Since we needed a way of ensuring the synthetic apps were representative of real apps, we first conducted an empirical study involving 100 real world apps chosen randomly from an open source repository, called F-Droid [12]. The selected apps were in various categories, such as education, Internet, games, etc. We analyzed these apps according to four complexity metrics that could impact EvoDroid:

- *Root Nodes per App* — the number of disconnected call graphs in the app; these are the methods called by ADF, and potentially the break points for EvoDroid segments.
- *Method Call Sequence Depth* — the longest method call sequence in the app.
- *McCabe Cyclomatic Complexity* — the average number of control flow branches per method.
- *Block Depth per Method* — the average number of nested condition statements per method.

Figure 8 shows the distribution of these metrics among the 100 Android apps from F-Droid. Our app generator is able to synthesize apps with varying values in these four metrics. Since we wanted to evaluate the accuracy and performance of EvoDroid on subjects with different levels of complexity, we had to derive some complexity classes from this data. For that, we aggregated the data collected through our empirical study, as shown in Figure 8, and divided it into 9 equal complexity classes, ranging from 1 to 9. For instance, the 1st complexity class corresponds to the 10th percentile in all of the four metrics shown in Figure 8. Essentially an app belonging to a lower class is less complex with respect to all four metrics than an app from a higher class.

8.4.1 Impact of Complexity

To benchmark the impact of complexity on EvoDroid, we generated two apps for each complexity class. Apps were set up such that exactly 1 path contained no input constraints, while other paths contained nested conditional input constraints. These constraints were generated to simulate the *Block Depth per Method* dimension, and would have to be satisfied in order for the search to progress

Table 1: Open source apps line coverage.

App Name	SLOC	EvoDroid	Android Monkey	Dynodroid
CalAdder	142	82%	18%	-
Tipster	280	89%	51%	48%
Munchlife	392	78%	45%	61%
JustSit	556	84%	37%	69%
AnyCut	1095	84%	6%	66%
TippyTipper	1649	82%	51%	-
NotePad	1655	76%	59%	65%
Bites	2301	80%	32%	39%
PasswordMaker	2824	76%	30%	36%
Bookworm	5840	77%	9%	43%

Table 2: Execution time for testing apps from different complexity classes (in minutes).

Complexity Class	# of Segments	EvoDroid Test Cases	Monkey/ DynoDroid Events	EvoDroid Single CPU	EvoDroid Parallel	Android Monkey	DynoDroid
1	4	30	300	41.55	1.65	1.52	235.57
1	6	40	400	44.10	1.75	1.55	273.27
2	8	50	500	52.00	2.01	1.52	309.43
2	9	60	600	88.31	2.20	1.53	325.78
3	17	130	300	258.77	2.40	1.65	458.90
3	19	140	140	293.38	2.53	1.58	407.60
4	25	210	2100	686.47	3.31	1.97	1091.00
4	33	220	2200	388.30	3.11	1.72	1073.40
5	36	260	2600	796.23	4.13	1.88	592.27
5	48	290	2900	692.77	3.92	2.23	823.57
6	49	280	2800	1107.79	4.90	2.02	623.57
6	47	520	5200	957.97	4.43	2.13	1100.70
7	109	750	7500	2725.27	5.11	2.50	1233.90
7	110	1110	11100	2999.87	5.53	2.82	-
8	233	1800	18000	7645.32	6.20	4.03	-
8	282	1960	19600	8035.17	6.53	4.17	-
9	345	3640	36400	13713.25	6.81	5.43	-
9	487	3680	36800	21395.50	7.59	6.00	-

further and attain deeper coverage. The generated conditional statements had a 50% satisfiability probability given a random input value. Of course, some of the conditional statements were nested in the synthetic apps, resulting in a lower probability of satisfying certain paths.

The line coverage results are summarized in Figure 9a. As the complexity class of apps increases, the coverage for Monkey and Dynodroid drops significantly. Since EvoDroid logically divides an app into segments, the complexity stays relatively the same, i.e., it is not compounded per segment. In all experiments, EvoDroid achieves over 98% line coverage. The cases where 100% coverage is not reached is due to EvoDroid abandoning the search when reaching the maximum number of allowable generations. Increasing the number of generations is likely to resolve those situations.

Once Monkey traverses a path, it does not backtrack or use any other systematic way to test the app. Therefore, Monkey's test coverage is shallow as others have confirmed in [32]. Dynodroid periodically restarts from the beginning of the app, and is able to outperform Monkey. Note that for very complex apps, Dynodroid would crash, and thus we were not able to obtain results.[1]

Table 2 summarizes the execution time for EvoDroid, Monkey, and Dynodroid. Even though the execution time for EvoDroid significantly increases as the apps become more complex, it could be alleviated by running EvoDroid in parallel, possibly on the cloud. EvoDroid parallel times are roughly equivalent to the worst path execution time. The numbers presented assume as many parallel instances running as there are test cases. In practice, we expect EvoDroid to be executed on several machines, but perhaps not hundreds, producing an execution time in between the worst case and best case reported in Table 2. We can see that as the depth of the segments increases, the time to execute EvoDroid increases. The results also show that Monkey runs fairly quickly, while Dynodroid takes longer, as one would expect due to its backtracking feature.

8.4.2 Impact of Constraints

Input constraint satisfaction is a known weakness of search based testing techniques. A set of experiments was conducted to assess the efficacy of our approach as the satisfiability probability of conditional statements was lowered below 50%. We took the second app from the 3rd complexity class (shown in Figure 9a) and lowered the satisfiability probability of its conditional statements to 25%, 10%, and 1%. As shown in Figure 9b, when the probability of constraint satisfiability decreases, the line coverage drops significantly for EvoDroid. Android Monkey coverage stays the same as it takes the one path with no constraints and does not backtrack. The coverage for Dynodroid drops also, but remains better than Monkey, as it restarts from the beginning several times during execution.

The results demonstrate that EvoDroid (as well as any other evolutionary testing approach) performs poorly in cases where the apps are highly constrained (e.g., the probability of satisfying many conditional constraints with random inputs is close to zero, such as an *if* condition that *specifies* an input value to be *equal* to a *specific value*). Fully addressing this limitation requires an effective approach for solving the constraints, such as symbolic execution, as described further in Section 10. Fortunately, from Figure 8 we see that for a typical Android app, the average cyclomatic complexity is approximately 2.2 and block depth is approximately 1.75. These numbers are encouraging, as they show that on average most Android apps are not very constrained.

8.4.3 Impact of Sequences

Given the event driven nature of Android apps, there are situations when certain sequences of events must precede others or certain number of events must occur to execute a part of the code. We evaluated EvoDroid for these types of situations by generating apps from the 3rd complexity class with ordered sequence lengths ranging from 1 to 5. Sequences of events with these lengths would have to be satisfied, per segment in all paths, in order to proceed with the search (e.g. certain buttons on an Activity must be clicked in a certain sequence of length 1 to 5).

Figure 9c summarizes the results from these experiments. While EvoDroid's coverage decreases, it does so at a much slower pace than Monkey or Dynodroid. We observe that EvoDroid is effective in generating system tests for Android apps involving complex sequence of events. This is indeed one of the strengths of EvoDroid that is quite important for Android apps as they are innately event driven.

9. RELATED WORK

The Android development environment ships with a testing framework [4] that is built on top of JUnit. Robolectric [14] is another testing framework for Android apps that runs tests without relying on the Android emulator. While these frameworks automate the ex-

[1]This is because above 10,000 events Dynodroid needs more than 3.4GB memory, which is more than the maximum memory size the 32-bit virtual machine that Georgia Tech researchers provided us for our experimentation could support.

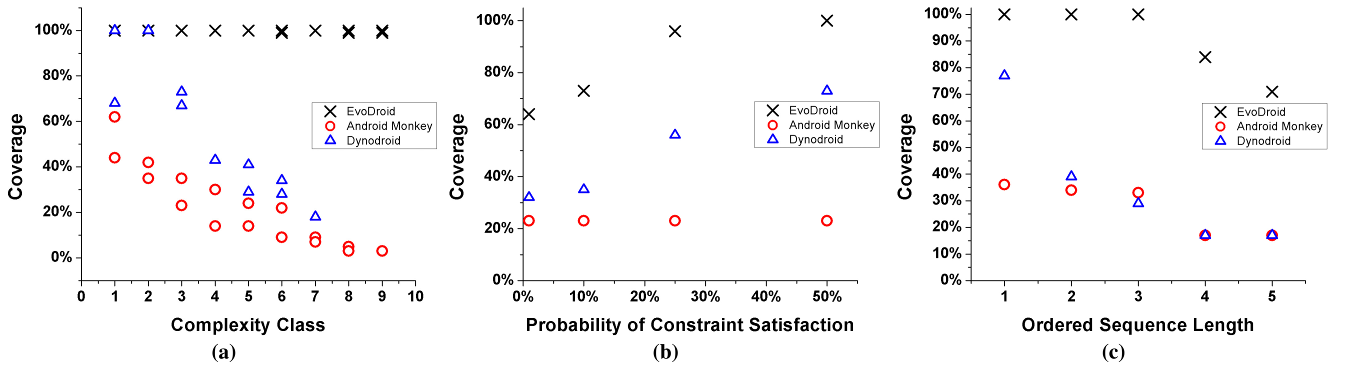


Figure 9: Benchmark Apps coverage results: (a) Line coverage results for testing apps from different complexity classes (b) Impact of constraints on line coverage (c) Impact of sequences of events on line coverage

ecution of the tests, the test cases themselves have to be developed manually.

Amalfitano et al. [18] described a crawling-based approach that leverages completely random inputs to generate unique test cases. In a subsequent work [19], they presented an automated Android GUI ripping approach where task lists are used to fire events on the interface to discover and exercise the GUI model. Hu and Neamtiu [29] presented a random approach for generating GUI tests that uses the Android Monkey to execute. We use program analysis to derive the models. This sets us apart from these works that employ black-box testing techniques.

Yang et al. [38] described a grey-box model creation technique that similar to our work is concerned with deriving models for testing of Android app and can potentially be substitued for our models. They also found that models generated using their approach could be incomplete. Jensen et al. [31] presented a system testing approach that combines symbolic execution with sequence generation. They attempt to find valid sequences and inputs to reach prespecified target locations. Their approach neither uses an evolutionary search technique like ours, nor is their goal maximizing code coverage. Anand et al. [20] presented an approach based on concolic testing of a particular Android library to identify the valid GUI events using the pixel coordinates. Dynodriod [32] is an input generation system for Android that was used extensively in our experiments.

Evolutionary testing falls under search based testing techniques and has typically been used to optimize test suites [17, 27, 28, 34] or has been applied at the unit level [30]. Evolutionary testing approaches such as [17, 27, 28, 34] have attempted to optimize a test suite for coverage. These are all blackbox approaches that build their GUI models at run time by executing and crawling or by recording user behavior, and therefore the generated models may not be complete. Since our approach uses program analysis, we obtain a more complete model of the app's behavior. They also differ from us in that they represent test cases as genes. Two unit level evolutionary testing approaches were presented in [36, 37]. A combination of approaches were also presented in [22, 30]. Evolutionary algorithm and symbolic execution techqniques were combined in [30], while evolutionary algorithm and hill climbing algorithm were used together in [22]. These techniques are all geared towards unit testing. An ant colony optimization search was used in [23] along with a call graph similar to ours for GUI testing. However, their call graph is generated by executing the system and connecting overlapping call nodes to attempt to form the entire call graph. This approach suffers from the same issue as the GUI crawling methods, meaning that the call graph model may be incomplete. Choi et al. [25] proposed a machine learning approach to improve the app models by exploring the states not encountered during manual testing. This work is complementary to our work, as it may be possible to use these improved models in EvoDroid.

There has also been a recent interest in using cloud computing to validate and verify software. TaaS is a testing framework that automates software testing as a service on the cloud [24]. Cloud9 [26] provides a cloud-based symbolic execution engine. Similarly, our framework is leveraging the computational power of cloud to scale evolutionary testing.

10. CONCLUDING REMARKS

We have presented EvoDroid, a novel framework for automated testing of Android apps. The key contributions of our work are (1) an automated technique to generate abstract models of the app's behavior to support automated testing, (2) a segmented evolutionary testing technique that preserves and promotes the genetic makeup of individuals in the search process, and (3) a scalable system-wide testing framework that can be executed in parallel on the cloud.

Although our approach has shown to be significantly better than existing tools and techniques for automated testing of Android apps, in the worst case scenario it can degrade quite a bit due to its inability to systematically reason about input conditions. This is a known limitation of search based algorithms, such as evolutionary testing. In our ongoing work [33], we are developing an Android-specific symbolic execution engine. We are extending Java Pathfinder, which symbolically executes pure Java code, to work on Android. We plan to use both techniques in tandem to complement one another. We are also exploring relational logic and the associated model finders for generating reduced combinatorics in testing Android apps [21].

Another weakness of our approach is not being able to fully generate models for apps that use third party libraries or native code. There is also a significant variability in the way the app code is generally written. As a result, the models may in fact be incomplete. However, as mentioned earlier and evaluated in Section 8.3, EvoDroid is able to work on partial and/or incomplete models. In the future, we plan to improve our models generation capabilities to handle a larger subset of the Android specifications.

11. ACKNOWLEDGMENTS

We would like to thank Aravind Machiry and Mayur Naik at Georgia Tech for helping us with the setup of Dynodroid. This work was supported in part by awards W911NF-09-1-0273 from the US Army Research Office, D11AP00282 from the US Defense Advanced Research Projects Agency, and CCF-1252644 from the US National Science Foundation.

12. REFERENCES

[1] Android developers guide, http://developer.android.com/guide/topics/fundamentals.html.

[2] Android emulator | android developers, http://developer.android.com/tools/help/emulator.html.

[3] Android monkey, http://developer.android.com/guide/developing/tools/monkey.html.

[4] Android testing framework, http://developer.android.com/guide/topics/testing/index.html.

[5] Apktool, http://code.google.com/p/android-apktool/.

[6] Dalvik - code and documentation from android's VM team, http://code.google.com/p/dalvik/.

[7] Dedexer, http://dedexer.sourceforge.net/.

[8] Dex2jar, http://code.google.com/p/dex2jar/.

[9] Ecj, http://cs.gmu.edu/ eclab/projects/ecj/.

[10] EMMA, http://emma.sourceforge.net/.

[11] Evodroid:segmented evolutionary testing of android apps, http://www.sdalab.com/projects/evodroid.

[12] F-droid, https://f-droid.org/.

[13] MoDisco, http://www.eclipse.org/modisco/.

[14] Robolectric, http://pivotal.github.com/robolectric/.

[15] Robotium, http://code.google.com/p/robotium/.

[16] Smali, http://code.google.com/p/smali/.

[17] I. Alsmadi, F. Alkhateeb, E. Maghayreh, S. Samarah, and I. A. Doush. Effective generation of test cases using genetic algorithms and optimization theory. *Journal of Communication and Computer*, 7(11):72–82, 2010.

[18] D. Amalfitano, A. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261, March 2011.

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, 2012.

[20] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, 2012.

[21] H. Bagheri, K. Sullivan, and S. Son. Spacemaker: Practical formal synthesis of tradeoff spaces for object-relational mapping. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, pages 688–693, 2012.

[22] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for java. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 185–194, 2010.

[23] S. Bauersfeld, S. Wappler, and J. Wegener. A metaheuristic approach to test sequence generation for applications with a gui. In *Proceedings of the Third International Conference on Search Based Software Engineering*, SSBSE'11, pages 173–187, 2011.

[24] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 155–160, 2010.

[25] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, 2013.

[26] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, Jan. 2010.

[27] F. Gross, G. Fraser, and A. Zeller. Exsyst: Search-based gui testing. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1423–1426, June 2012.

[28] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, 2012.

[29] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, 2011.

[30] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 297–306, 2008.

[31] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, 2013.

[32] 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*, ESEC/FSE 2013, pages 224–234, 2013.

[33] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[34] A. Rauf, A. Jaffar, and A. A. Shahid. Fully automated gui testing and coverage analysis using genetic algorithms. *International Journal of Innovative Computing, Information and Control (IJICIC) Vol*, 7, 2011.

[35] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, Mar. 2010.

[36] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128, 2004.

[37] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1053–1060, 2005.

[38] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, 2013.