Reducing Combinatorics in GUI Testing of Android Applications

Nariman Mirzaei[†] Joshua Garcia[*] Hamid Bagheri[*] Alireza Sadeghi[*] Sam Malek[*]

[†]Department of Computer Science
George Mason University
nmirzaei@gmu.edu

[*]Department of Informatics
University of California, Irvine
{joshug4,hamidb,alirezs1,malek}@uci.edu

ABSTRACT

The rising popularity of Android and the GUI-driven nature of its apps have motivated the need for applicable automated GUI testing techniques. Although exhaustive testing of all possible combinations is the ideal upper bound in combinatorial testing, it is often infeasible, due to the combinatorial explosion of test cases. This paper presents *TrimDroid*, a framework for GUI testing of Android apps that uses a novel strategy to generate tests in a combinatorial, yet scalable, fashion. It is backed with automated program analysis and formally rigorous test generation engines. TrimDroid relies on program analysis to extract formal specifications. These specifications express the app's behavior (i.e., control flow between the various app screens) as well as the GUI elements and their dependencies. The dependencies among the GUI elements comprising the app are used to reduce the number of combinations with the help of a solver. Our experiments have corroborated TrimDroid's ability to achieve a comparable coverage as that possible under exhaustive GUI testing using significantly fewer test cases.

Keywords

Android, Software Testing, Input Generation

1. INTRODUCTION

With well over a million apps, Android has become one of the dominant mobile platforms [16]. Android app markets, such as Google Play, have created a fundamental shift in the way software is delivered to consumers, with thousands of apps added and updated on a daily basis. The majority of these apps are developed at a nominal cost by entrepreneurs that do not have the resources for properly testing their software. Hence, there is an increasing demand for applicable automated testing techniques. One key obstacle towards achieving test automation for GUI-driven Android apps is the lack of effective techniques for test input generation.

A recent study of existing tools by Choudhary et al. [15] claims Android Monkey [1], a random-testing program for Android, to be the winner among the existing test input generation tools. Android Monkey provides a random mechanism that often achieves shallow code coverage. Several recent research efforts [8, 9, 11, 21, 27, 38], including our own [28–31], have aimed to improve Android testing practices. However, to the best of our knowledge, no prior research has explored a *fully automated* combinatorial GUI testing approach in the context of Android.

This is mainly because *exhaustive* combinatorial GUI testing is often viewed to be impractical due to the explosion of possible combinations for even the smallest applications [19]. A more practical alternative is *t-way* combinatorial testing [33], where all combinations for only a subset of GUI widgets (i.e., *t*) are considered [20]. But even under t-way testing, the number of generated test cases could grow rapidly. Moreover, without a systematic approach to determine the interactions, arbitrary selection of *t* widgets to be combinatorily tested is bound to be less effective than an exhaustive approach in terms of both code coverage and fault detection.

An opportunity to automate the testing activities in Android is presented by the fact that apps are developed on top of an *Application Development Framework (ADF)*. The Android ADF ensures apps developed by a variety of suppliers can interoperate and coexist together in a single system (a phone), as long as they conform to the rules and constraints imposed by the framework. The Android ADF constrains the life cycle of components comprising an app, the styles of communication among its software components, and the ways in which GUI widgets (e.g., buttons, check-boxes) and other commonly needed functionalities (e.g., GPS coordinates, camera) can be accessed. An underlying insight in our research is that the knowledge of these constraints along with the metadata associated with each app can be used to automate many software testing activities, specifically combinatorial testing of apps.

In this paper, we present TrimDroid (**T**esting **R**educed GU**I** Co**M**binations for And**DROID**), a fully-automated combinatorial testing approach for Android apps. Given an Android APK file, TrimDroid employs static analysis techniques that are informed by the rules and constraints imposed by the Android ADF to identify GUI widgets that interact with one another.[1] Thus, the set of interacting widgets become candidates for t-way combinatorial testing. By avoiding the generation of tests for widgets that do not interact, TrimDroid is able to significantly reduce the number of tests. For identifying the interactions, TrimDroid statically analyzes the control- and data-flow dependencies among the widgets and actions available on an app. Finally, TrimDroid uses an efficient constraint solver to enumerate the test cases covering all possible combinations of GUI widgets and actions.

Our evaluation of TrimDroid shows that it achieves the same coverage as exhaustive combinatorial testing, but reduces the number of test cases by 57.86% on average and by as much as 99.9%. This reduction is important, as it not only reduces the time it takes to execute the test cases, but also significantly decreases the effort required to inspect the test results.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '16 Austin, Texas USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884853

[1]An APK file is a Java bytecode package used to install Android apps.

The paper is organized as follows. Section 2 presents an illustrative app to motivate and describe the research. Section 3 provides an overview of TrimDroid. Sections 4 and 5 describe the extraction of required models and dependencies from apps. Sections 6 and 7 describe enumeration of execution scenarios and generation of test cases, respectively. Section 8 presents our experimental evaluation of TrimDroid. The paper concludes with an overview of the related research, and a discussion of our future work.

2. ILLUSTRATIVE EXAMPLE

We use a simple Android app, called Expense Reporting System (ERS), to illustrate our research. This app allows a user to maintain a log of meal expenses incurred during a trip. Figure 1 depicts two of ERS's *Activities*: *NewReportActivity* and *ItemizedReportActivity*. [2]

NewReportActivity is the *main* Activity, i.e., it is the first screen presented to the user when an app is invoked. From *NewReportActivity*, the user can select the *Destination*, enter an allowable expense *Amount*, the *Currency*, and initiate the creation of two types of reports: *Itemized Report* and *Quick Report*. *ItemizedReportActivity* allows the user to enter an itemized list of meal expenses, including (1) the total days of the trip, and (2) the number of meals purchased on the trip's first and last day. When *Total Days* is 1 (i.e., the first and last days are the same), the check-boxes corresponding to the last day meals are disabled (see Figure 1a). On the other hand, *QuickReportActivity* (not shown in Figure 1 for brevity) allows the user to provide an aggregate number for the meal expenses incurred on a trip.

Regardless of the approach used for entering the expenses, the user is led to a confirmation page, where she can submit the expenses, and is presented with a summary report that she can save. An overview of the relationships among the Activities comprising the ERS are depicted in Figure 3.

Testing of GUI-driven apps, such as ERS, requires utilizing a large number of event sequences. These sequences are often generated by GUI interactions involving radio-boxes, check-boxes, drop-down lists, etc. Exhaustive combinatorial testing [20], i.e., a brute-force approach that tries all possible GUI combinations, is often computationally prohibitive.

An alternative approach to exhaustive testing is *t-way* combinatorial testing [26]. Consider a GUI screen under test that has a total of n widgets. *t-way* combinatorial testing requires that all possible t-way combinations of widget values are selected, where $t < n$. The most common type of t-way testing is *pairwise* testing, where $t = 2$ [33]. Although t-way testing produces a smaller number of tests, it is less effective than exhaustive testing in terms of both code coverage and fault detection. For instance, when pairwise testing is used, code that depends on the interaction of three or more GUI widgets may remain uncovered.

To illustrate the challenges of combinatorial testing, consider a situation in which the user clicks on the *ItemizedReport* button of *NewReportActivity* and subsequently on the *Next* button of *ItemizedReportActivity* (see Figure 1). *NewReportActivity* contains the *Destination* drop-down list with 10 choices, and the *Currency* check-box with 3 exclusive choices. Let us also assume two values of 100 and 0 have been identified as proper input classes for the *Amount* field. This would result in a total of $10 \times 3 \times 2 = 60$ unique combinations for *NewReportActivity*. Similarly, *ItemizedReportActivity* contains the *Total Days* drop-down list with 6 choices, the *First Day Meals* and *Last Day Meals*, each of which has 3 inclusive choices, resulting in a total of $6 \times 2^3 \times 2^3 = 384$ unique combinations.

[2]An *Activity* is a type of Android component that represents a GUI screen.

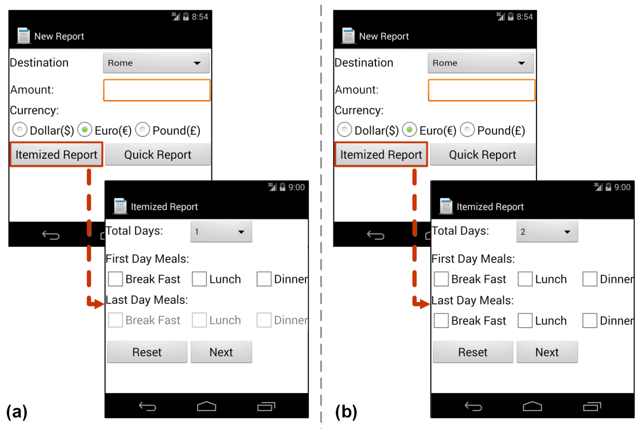

Figure 1: Screenshots for a part of ERS app: (a) when total days is 1, the check-boxes for *Last Day Meals* are disabled, and (b) when the total days is greater than 1, the check-boxes for *Last Day Meals* are enabled.

Since the widget values selected on one Activity could impact the behavior that is manifested in subsequent Activities, for GUI system testing, we also need to consider the interaction of widgets across Activities. Thus, the number of all unique tests for the above use case is $60 \times 384 = 23,040$. The number of tests would continue to grow if we consider the other Activities comprising this app. This approach is infeasible in practice, in terms of both the effort required to execute the tests and the effort required in assessing the results.

TrimDroid drastically reduces the number of tests for achieving a comparable coverage as exhaustive GUI testing. The insight guiding our research is that not all GUI widgets and actions interact with one another.

To that end, TrimDroid statically extracts the control- and data-flow dependencies among the GUI widgets, event handlers, and Activities of an app, and it does so without access to source code, rather from the app's APK file.

An example of GUI widget interaction can be gleaned from Figure 1a. Here, we can see that when *Total Days* obtains a value of 1, *Last Day Meals* check-boxes are disabled, thus indicating a dependency between these two widgets, implying that their combinations should be tested. On the other hand, if our analysis indicates that *Total Days* and *First Day Meals* are indeed independent of one another, we can safely conclude that their combinations do not need to be tested. TrimDroid detects such dependencies, which provide the basis for combinatorial generation of tests.

To appreciate the significant reductions possible this way, consider the use case of ERS described earlier. TrimDroid generates only $max\{(6 \times 2^3), 2^3\} = 48$ tests for *ItemizedReportActivity* when the *Next* button is clicked. That represents a reduction of 336 combinations compared to the exhaustive approach. TrimDroid realizes that the $(6 \times 2^3) = 48$ possible combinations for *Total Days* and *Last Day Meals* are independent of the $2^3 = 8$ possible combinations for *First Day Meals*. Since we can use combinations of independent widgets in the same test, the dependent widgets with the biggest number of unique combinations determine the number of generated tests. Here, the 48 combinations for *Total Days* and *Last Day Meals* are merged with the 8 combinations for *First Day Meals* to produce 48 widget combinations for *ItemizedReportActivity*. For testing both activities together, TrimDroid produces $60 \times 48 = 2,880$ tests, representing a reduction of $20,160$ tests compared to the exhaustive approach.

Assuming an accurate extraction of dependencies through static analysis, the reduced set of tests generated using TrimDroid would be as effective as exhaustively generated tests in terms of their coverage and fault detection power.

3. APPROACH OVERVIEW

Figure 2 depicts a high-level overview of TrimDroid, which is comprised of four major components: *Model Extraction*, *Dependency Extraction*, *Sequence Generation*, and *Test-Case Generation*. Together, these components produce a significantly smaller number of test cases than an exhaustive combinatorial technique, yet achieve a comparable coverage.

Similar to our previous work [29], *Model Extraction* produces two types of models by statically analyzing an Android app:

- Interface Model (*IM*) provides a representation of all the GUI inputs of an app, including the input widgets and events (actions) for each Activity. TrimDroid uses the IM to determine how a GUI screen can be exercised in order to generate the tests for it.
- Activity Transition Model (*ATM*) is a finite state machine representing the event-driven behavior of an Android app, including the relationships among its Activities and their event handlers (transitions). Since our research targets GUI testing, we only extract information that is related to Activities, not other Android components (e.g., Services). Figure 3 depicts the *ATM* for the entire ERS app.

These models are represented in Alloy [22], a formal specification language based on first order relational logic. Alloy specifications can be analyzed using Alloy Analyzer, thereby allowing us to systematically explore the combinatorial space with the help of a constraint solver.

In a step parallel to *Model Extraction*, *Dependency Extraction* identifies GUI-induced dependencies among app elements using a combination of control- and data-flow analysis techniques. *Dependency Extraction* identifies three types of dependencies (1) when one GUI widget depends on the value of another widget, e.g., a drop-down menu is disabled, because a check-box is not selected, (2) when a GUI event handler depends on a widget value, e.g., a button handler method uses the selected value of a check-box, and (3) when an Activity depends on the widget values from a preceding Activity, e.g., the widget values from a preceding Activity are included in the payload of an *Intent* starting a new Activity.[3] These dependencies are also represented in the form of Alloy specifications and used by *Test-Case Generation* in a later step for pruning the combinatorial space of tests.

Sequence Generation uses the Alloy Analyzer to synthesize sequences of events that cover the paths in the *ATM*. Each path in the *ATM* represents a sequence of events in a possible use case. A good coverage of the *ATM* is essential for achieving high code coverage. TrimDroid covers the paths using the *prime path* coverage criterion, known to subsume most other graph coverage criteria [10].

Finally, *Test-Case Generation* constructs system tests by performing three key steps. First, it traverses the sequences of events representing the paths produced by *Sequence Generation*. Second, for each step in a given sequence, it uses Alloy Analyzer to generate value combinations for different GUI widgets. To that end, *Test-Case Generation* utilizes (1) the sets of dependent widgets generated by *Dependency Extraction* and (2) the specification of each widget in the *IM*. Lastly, *Test-Case Generation* merges the value combinations to create tests that cover the entire sequence of events in each path of the ATM. The generated tests can then be executed using Robotium [6], an Android test-automation framework.

[3]All Android components 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.

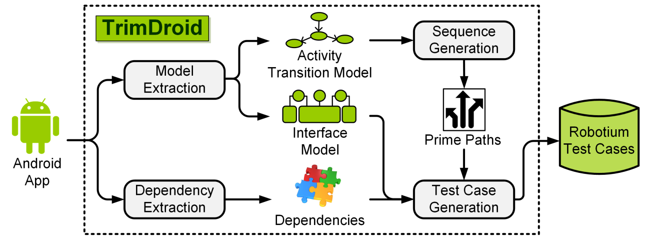

Figure 2: A high-level overview of TrimDroid

The next four sections describe the four components of TrimDroid in more detail.

4. MODEL EXTRACTION

TrimDroid extracts two types of Alloy models for each app: *Interface Model (IM)* and *Activity Transition Model (ATM)*. We define each model and describe the extraction process for each in the remainder of this section.

4.1 Interface Model

The *IM* provides information about all of the GUI inputs of an app, such as the widgets and input fields belonging to an Activity. More formally, the IM is defined as follows:

Definition 1. The *IM* of an app is a tuple $\langle A, E, W, I \rangle$, where

- A is a finite, non-empty set of Activities of the app.
- E is a finite set of event handlers of the app (e.g., *onClick()* is the handler for a button click). Each Activity $a \in A$ has a set of event handlers $eHandlers(a) \subseteq E$.
- W is a finite set of GUI widgets of the app (e.g., a check-box, radio-button). Each Activity a has a set of widgets $widgets(a) \subseteq W$.
- I is a finite set of input classes for widgets of the app. Each widget w has a set of input classes $ic(w) \subseteq I$. Each input class is a partition of the input domain of each widget. For instance, input classes of a check-box are *checked* and *unchecked*, while input classes of a drop-down menu are its choices.

Model Extraction obtains the *IM* by analyzing the information contained in the meta-data included in an Android APK file, namely its XML-based *manifest* and *layout* files. More specifically, *Model Extraction* determines all the Activities within an app from its manifest file. Subsequently, for each Activity, *Model Extraction* parses the corresponding layout file to obtain all information for each widget, such as its name, id, input type, etc. Our current implementation extracts the input classes for widgets that provide users with a list of options, such as check-boxes and drop-down menus, directly from the layout files. We use the same layout files to divide the domain space of text-boxes into different classes based on the limits imposed on the text box values (e.g., max length). For unbounded text boxes, and other unbounded widgets, we use a configurable set of input classes that can be defined by the user.

4.2 Activity Transition Model

An *ATM* represents the high-level behavior of an app's GUI in terms of its Activities and the transitions resulting from invocations of its event handlers. More formally, the *ATM* is defined as follows:

Definition 2. The *ATM* of an app is a finite state machine represented as a tuple $\langle A, a_0, E, F \rangle$, where

- A is a finite, non-empty set of Activities.
- a_0 is the starting Activity (i.e., *main* Activity), defined in an app's manifest file.

- E is a finite set of directed transitions from the starting Activity to final Activities, labeled by event-handler names. Each transition represents an event handler and denoted as $a_i \xrightarrow{e_k} a_j$, where $a_i, a_j \in A$ and e_k is an event handler.
- F is a finite, non-empty set of final Activities in the *ATM*.

Figure 3 shows the ATM for the ERS app. To obtain an *ATM* such as this, *Model Extraction* first determines the Activities $A = \{a_0, a_1, a_2, a_3, a_4\}$ comprising the app from its manifest file. To determine the transitions between the Activities, *Model Extraction* performs a depth-first traversal of *main* Activity's call graph starting from its *onCreate()* method, which we know from Android's ADF specification to be the starting point of all apps. In the context of ERS, this corresponds to *NewReportActivity*'s *onCreate()* method. For each encountered node in the call graph, *Model Extraction* checks whether it would result in an activity transition, and if so, adds it to set E.

A call may result in a transition in two ways:

1. *Inter-component transition:* these are implicit calls that result in the transfer of control from one Activity to another Activity. For instance, in the example of ERS in Figure 1, when the *Itemized Report* button is clicked, the corresponding handler calls Android's *startActivity* method, which sends an Intent message resulting in the transfer of control to *ItemizedReportActivity*'s *onCreate()* method. In this case, we extract the destination from the Intent, and add the following transition $E = E \cup \{a_0 \xrightarrow{onClick(ItemizedReport)} a_2\}$.

2. *Intra-component transition:* these are implicit calls to GUI event handlers in an Activity that result in a transition back to the same Activity. For instance, the *ItemizedReportActivity* has a *Click* event associated with its *Reset* button. This event is handled by the Activity's *onClick()* method that is registered with that button. In this case, we add the following transition to the model: $E = E \cup \{a_2 \xrightarrow{onClick(Reset)} a_2\}$.

Upon traversing the call graph of a_0, the above process repeats for all of the Activities remaining in A. Finally, we populate the set F with the Activities that do not have any outgoing inter-component transitions, and if they do, it is only to nodes that precede them.

We implemented the *Model Extraction* component on top of Soot, a static-analysis framework for Java [37]. To analyze an Android app, TrimDroid utilizes the Dexpler transformer [13] to translate Android's Dalvik bytecode to Jimple, Soot's intermediate representation. By leveraging Soot and Dexpler, TrimDroid works with an app's source code as well as its APK file.

5. DEPENDENCY EXTRACTION

TrimDroid uses the dependencies among the app elements to determine the combinations that should be tested, and those that can be safely pruned. To that end, *Dependency Extraction* determines three types of dependencies as described further below.

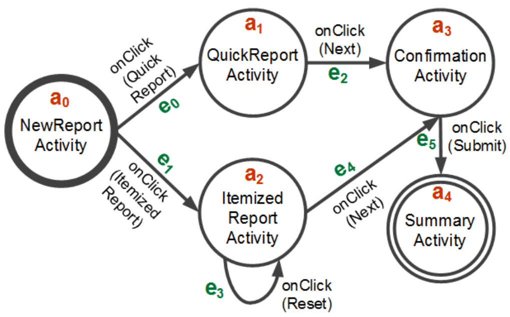

Figure 3: Activity Transition Model for the ERS app

Algorithm 1: *wDep*

```
Input: $a \in A$
Output: $WD \subset \mathcal{P}(widgets(a))$
1  $WD \leftarrow \emptyset$;
2  $depPairs \leftarrow \emptyset$;
3  foreach $meth \in a.Methods$ do
4      foreach $w$ used in a conditional statement $stmt_1$ of $meth$ do
5          if $isAWidget(w)$ then
6              foreach $w_{1a}$ used along either branch of $stmt_1$ do
7                  if $isAWidget(w_{1a})$ then
8                      $depPairs \leftarrow depPairs \cup \{\{w_{1a}, w\}\}$;
9              foreach $r_v$ defined along either branch of $stmt_1$ do
10                 foreach $w_2 \in widgetsWhoseValueAffects(r_v)$ do
11                     $depPairs \leftarrow depPairs \cup \{\{w_2, w\}\}$;
12                 foreach conditional statement $stmt_2$ that uses $r_v$ do
13                     foreach $w_{1b}$ used along either branch of $stmt_2$ do
14                         if $isAWidget(w_{1b})$ then
15                             $depPairs \leftarrow depPairs \cup \{\{w_{1b}, w\}\}$;
16 $WD \leftarrow merge(depPairs)$;
17 $WD \leftarrow WD \cup isolateRemainingWidgets(WD, widgets(a))$;
```

5.1 Widget Dependency

Two widgets $w1$ and $w2$ are dependent if combinations of their values affect an app's control- or data-flow. Widget combinations that affect the control-flow impact the code coverage of generated tests; widget combinations that affect the data-flow determine the state of the system under test. Here are two possible dependencies between $w1$ and $w2$ that our approach detects:

(Case 1) $w2$'s use depends on the value of $w1$. This can occur in two situations. First, a widget $w1$ is used in a conditional statement, and widget $w2$ is used along either branch of that statement. An example of the first case is shown below, where `lastBreakFast` is dependent on `totalDays`:

```
if((String.valueOf(totalDays.getSelectedItem()))
    .equals("1")) {
    lastBreakFast.setEnabled(false);   }
```

The second situation occurs when the value of widget $w1$ affects a reference r, and $w2$'s use depends on the value of r. An example of this case is shown below, where the use of `totalDays` is indirectly dependent on `firstBreakfast` based on the variable `mealsCount`:

```
if(firstBreakFast.isChecked())
  mealsCount++;
if(mealsCount > 0)
  totalMeals = totalDays.getValue()*3 + mealsCount;
```

(Case 2) In a conditional statement, widget $w1$ is used and reference r is defined in its block, and r is later used in the block of another conditional statement, where $w2$ is used. An example of this case is shown below, where the value combinations of `firstBreakfast` and `firstLunch` impact the value of `mealsCount`:

```
if(firstBreakFast.isChecked())
  mealsCount++;
if(firstLunch.isChecked())
  mealsCount++;
```

Algorithm 1 defines *wDep*, which partitions $widgets(a)$ based on the two cases above. The algorithm takes an Activity a as input and produces WD, a partition for $widgets(a)$ where $WD \subset \mathcal{P}(widgets(a))$.

For each method, *wDep* iterates over each reference w that is used in a conditional statement and determines if w refers to a widget (lines 3–5 of Algorithm 1). To make that determination, $isAWidget(w)$ traverses the definition-use chain of w to determine

Algorithm 2: *hDep*

```
Input: $a \in A, e \in eHandlers(a)$
Output: $HD \subset \mathcal{P}(widgets(a))$
1 $HD \leftarrow \emptyset$;
2 foreach $r$ is used in $e$ do
3     if $isAWidget(r)$ then
4         foreach $wd \in wDep(a)$ do
5             if $r \in wd$ then
6                 $HD \leftarrow HD \cup \{wd\}$;
7                 break;
```

Algorithm 3: *aDep*

```
Input: $a_i, a_j \in A, e \in eHandlers(a)$
Output: $boolean$
/* get the intent sent from $a_i$ to $a_j$ in $e$ */
1 $I \leftarrow a_i.getIntent(e, a_j)$
2 foreach $payload \in I.IntentExtras()$ do
3     $refs \leftarrow getAffectedReferences(payload)$
4     foreach $r \in refs$ do
5         if $isAWidget(r)$ then
6             return true;
7 return false;
```

if any of its definitions refers to a widget. At this point, *wDep* distinguishes the two cases that result in widget dependencies.

To determine the first situation of Case 1, *wDep* checks if any other variable w_{1a} is used and references a widget (lines 6–7 of Algorithm 1). If so, *wDep* creates a widget dependency $\{w_{1a}, w\}$ (line 8 of Algorithm 1).

To obtain widget dependencies for Case 2, *wDep* identifies any variable r_v defined after the conditional statement where w is referenced (line 9 of Algorithm 1). For any r_v whose value is affected by widget w_2 (line 10 of Algorithm 1), *wDep* creates the widget dependency $\{w_2, w\}$ (lines 11 of Algorithm 1). Here, $widgetsWhoseValueAffects(r_v)$ returns the widgets used in a conditional statement whose value affects a reference r_v by traversing r_v's definition-use chain.

To identify the second situation of Case 1, *wDep* further checks if reference r_v is used in a second conditional statement (line 12 of Algorithm 1). If a reference to a widget w_{1b} is used along either branch, then *wDep* creates a widget dependency $\{w_{1b}, w\}$ (lines 13–15 of Algorithm 1).

The widget dependency pairs are then merged (line 16). Two widget dependency pairs are merged if any of their elements intersect. For instance, two dependency pairs $\{w_\alpha, w_\beta\}$ and $\{w_\beta, w_\omega\}$ are merged, and the resulting set $\{w_\alpha, w_\beta, w_\omega\}$ is stored in *WD*. Finally, widgets that do not interact with any other widget, and thus not part of any dependency pair set, are each isolated into their own singleton set and added to *WD* (line 17).

5.2 Handler Dependency

To further reduce the number of test cases, *Dependency Extraction* identifies the dependencies between widgets and event handlers. This kind of dependency occurs when a widget value is used in an event handler, indicating that all combinations of the widget and the event resulting in the invocation of event handler should be tried. As an example of this, consider the following code snippet, where the value of `totalDays` is used in the `onClick()` method of `NextButton` in the *ItemizedReportActivity* of ERS:

```
public class NextButton implements OnClickListener
{
  public void onClick(View v) {
    totalDaysValue = String.valueOf(totalDays.
        getSelectedItem());
...}}
```

If an event handler uses multiple widgets, all combinations of those widgets according to their widget dependencies need to be tested together with the handler's event.

Thus, the pruning of irrelevant test combinations is achieved through determining the widgets that are not used by event handlers. For example, the `onClick()` handler for the *Reset* button of *ItemizedReportActivity* clears the screen regardless of the values of the widgets. Hence, no value combinations of the widgets on *ItemizedReportActivity* need to be tested with the *Reset* button.

Algorithm 2 defines *hDep*, which partitions $widgets(a)$ into a set based on handler dependencies. The input to the algorithm is an activity a and an event handler e. The output of the algorithm is *HD*, a partition of $widgets(a)$ where $HD \subset \mathcal{P}(widgets(a))$.

5.3 Activity Dependency

The third type of dependency involves the widget values in one Activity that may impact the behavior of another Activity. If so, we need to test all combinations of those widgets in a first Activity impacting a second activity with all combinations of widgets in the second Activity. Since Activities in Android communicate using Intent messages, we say the value of a widget w in an Activity a_i may impact another Activity a_j, if it affects the payload of an Intent that is sent from a_i to a_j. For example, as shown in the following code snippet, *NewReportActivity* sends an Intent that starts the *ItemizedReportActivity* and passes the selected value for `currencyRB` in the payload of the Intent:

```
public class ItemizedReportButton implements
          OnClickListener {
  public void onClick(View v) {
    int selectedId = currencyRB.getCheckedRadioButtonId
        ();
    currencyRB = (RadioButton)findViewById(selectedId);
    String currency = currencyRB.getText();
    Intent intent = new Intent(this,
        ItemizedReportActivity.class);
    intent.putExtra("currency", currency);
    startActivity(intent);}}
```

On the other hand, from the above code snippet, we can see that the value of *Destination* drop-down menu from *NewReportActivity* does not impact the Intent sent to *ItemizedReportActivity*. Thus, there is no need to test all combinations of widgets on *NewReportActivity* with the widgets on *ItemizedReportActivity*. This provides us with yet another opportunity to prune the tests.

Algorithm 3 defines *aDep*, which determines whether an Activity has a dependency to widget values selected in a preceding Activity. The algorithm takes two Activities a_i and a_j, corresponding to the source and destination of an Intent, an event handler e, realizing the transition between the two activities, and returns true if an Activity dependency exists and false otherwise.

6. SEQUENCE GENERATION

In GUI system testing, a test is comprised of two parts: sequence of events (e.g., button clicks) and selection of input values (e.g., drop-down menu choices). In this section, we describe how TrimDroid produces sequences of events that represent possible use cases for the system. In the next section, we provide the details of how the dependencies are used to determine the combination of input values for each sequence of events.

Our approach for the generation of event sequences is based on using a formal language to describe the *ATM* as well as the coverage criteria for traversing it. We then use an automated constraint solver to exhaustively synthesize the space of possible paths. Each path in the *ATM* represents a sequence of event handlers triggered in a possible use case for the system. These paths can be generated using any given coverage criteria (e.g., node coverage, edge-pair coverage). TrimDroid relies on *prime path coverage* as it has been shown to *subsume* most other graph coverage criteria [10]. A coverage criterion α subsumes coverage criterion β, if and only if 100% α coverage implies 100% β coverage [35].

TrimDroid represents an *ATM* in the form of an Alloy model [22]. Alloy is a formal modeling language with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [22], making it an appropriate language for declarative specification of both application models and properties to be verified. Listing 1 shows (part of) the Alloy specification of ATM, specifically the signatures for `activity`, `simplePath` and `primePath`. Each Activity has a set of event handlers (`eHandlers`), and a field (`isStart`), indicating whether it is a starting activity or not. Lines 4–11 present the `simplePath` signature along with its *facts* that specify the elements involved in, and the semantics of, a *simple path*, respectively. A simple path is a sequence of *transitions* from the starting activity (i.e., a_0), where no activity node appears more than once, except possibly when the first and last nodes are the same. A *prime path* then, as specified in lines 12–14, is a simple path that does not appear as a proper sub-path of any other simple path. An example of a prime path in the *ATM* of Figure 3 is: $a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_4} a_3 \xrightarrow{e_5} a_4$.

A test path satisfies prime path coverage if and only if it starts from a starting node and ends in a final node while covering a prime path in a graph [10]. The prime-path criterion limits visits of each loop to one, since simple paths have no internal loops. It also limits the number of generated paths, as it only contains paths that are not sub-path of any other path. The *ATM* of our running example (see Figure 3) thus includes three prime paths, automatically generated using Alloy Analyzer.

7. TEST GENERATION

Each system test $st \in ST$ is comprised of a sequence of *activity tests*: $st = \langle at_{a_0}, at_{a_1}, ..., at_{a_n} \rangle$. An *activity test* consists of widget value combinations and an event that exercises a particular Activity and results in a transition, either to itself or to another Activity, according to *ATM*.

TrimDroid generates the system tests in two steps. First, it generates the activity tests using the widget value combinations and events available on each Activity. Afterwards, it combines activity tests into a sequence that represents a GUI system test to cover a particular prime path.

To illustrate, we use the execution scenario for the ERS app shown in Figure 1. The ATM for ERS (recall Figure 3) shows its five activities (denoted as a) and their transitions (denoted as e). The input classes for widgets on `NewReportActivity` are captured in Figure 4a, where $ic(w)$ indicates the possible values for widget w. Figure 4b captures the same information for `ItemizedReportActivity`. In addition, the widget dependencies (recall Algorithm 1), handler dependencies (recall Algorithm 2), and activity dependencies (recall Algorithm 3) are all denoted in Figure 4c.

```
1   abstract sig activity{
2     isStart: one IsStart,                  // indicator of a starting activity
3     eHandlers: set activity }                   // activity's event handlers
4   sig simplePath{
5     first: one activity,                               // the starting activity
6     transitions: activity->activity }{         // sequence of transitions
7       first.isStart = Yes
8       first in transitions.activty
9       all x: activity | lone x.transitions
10      transitions in eHandlers
11      no x: transitions.activity| x !in first.*(transitions) }
12  sig primePath extends simplePath{ }{
13    // an activity with no outgoing transition
14    no finalActivity[transitions].eHandlers }
15  fun finalActivity[r: activity->activity] : one activity {
16    r[activity]-r.activity }
```

Listing 1: Specifications for *ATM* in Alloy.

7.1 Activity Test Generation

We generate tests for an Activity a in three steps:

(Step 1) For each event $e \in eHandlers(a)$, we use Alloy Analyzer to enumerate over all combinations of widget values in a that are dependent on e. To determine those combinations, we utilize the set of handler dependencies. Let $h \in hDep(a,e)$ represent a set of dependent widgets with respect to an event handler e. We calculate WC_h, i.e., the widget combinations for h, using the Alloy Analyzer, as the Cartesian product of all the input classes for its widgets:

$$WC_h \equiv \bigotimes_{j=1}^{|h|} ic(w_j), where\ w_j \in h$$

For instance, in ERS as shown in Figure 4c, we can see that $hDep(a_0, e_1)$ is comprised of two sets: $\{dest, cur\}$ and $\{amount\}$. Each one of these two sets indicates a widget dependency among its members, as well as a handler dependency with respect to e_1 (i.e., `"ItemizedReport.onClick"`). We can determine the combination of their elements as shown in Figure 4d.

(Step 2) To generate tests for Activity a with respect to event e, every widget $w \in h$, where $h \in hDep(a,e)$, must be assigned a value. To achieve this, all widget combinations, i.e., all instances of WC_h, are combined into one final set. However, since these widgets are independent from one another, we simply need to *merge* them, rather than calculate their cross-product, as follows:

$$WC_{hDep(a,e)} \equiv merge(H), where\ H = \{h | h \in hDep(a,e)\}$$

Definition 3 (Merge)**.** Given a set of sets S, let $m = \forall s \in S, max(|s|)$, we merge all members of S into C defined as follows:

$$C = \{c_i | c_i \equiv \bigcup_{\forall s \in S} x_{(i \mod |s|)},$$
$$where\ 0 \leq i \leq (m-1) \wedge x_i \in s\}$$

Thus, considering the widget combination in Figure 4d, we can calculate the final set of combinations for Activity a_0 (`NewReportActivity`) in the context of event handler e_1 (`ItemizedReport.onClick`) as shown in Figure 4e.

As shown in Figure 4a, $|ic(dest)| = 10$, $|ic(cur)| = 3$ and $|ic(amount)| = 2$. Thus, merging $WC_{\{dest,cur\}}$ with $WC_{\{amount\}}$ produces 30 unique possible combinations. Note that since $WC_{\{amount\}}$ only has 2 combinations, when we merge it with $WC_{\{dest,cur\}}$, which has 30 combinations, we simply need to ensure all of its unique values are included in the generated tests. In this case, we chose values 100 and 0 for the first two combinations and simply chose 100 for all the remaining combinations. Since we know that *amount* does not interact with the other widgets, we just need to ensure all of its unique values are included in the combinations. However, we still need to include a value for *amount* for all combinations, as the event handler depends on it.

(Step 3) Given all widget value combinations for an Activity a in relation to an event handler $e \in eHandlers(a)$, we can now construct all of the corresponding activity tests $AT_{hDep(a,e)}$. To that end, we simply augment each element of the set $WC_{hDep(a,e)}$ with the action corresponding to the triggering of event handler, i.e., e, as follows:

NewReportActivity			
#	$ic(dest)$	$ic(amount)$	$ic(cur)$
1	Rome	100	Euro
2	London	0	Dollar
3	Rome		Pound
.	.		
.	.		
10	Berlin		

(a)

ItemizedReportActivity							
#	$ic(totalDays)$	$ic(fbf)$	$ic(fl)$	$ic(fd)$	$ic(lbf)$	$ic(ll)$	$ic(ld)$
1	1	true	true	true	true	true	true
2	2	false	false	false	false	false	false
3	3						
.	.						
.	.						
6	6						

(b)

Dependencies
$wDep(a_0) = \{\{dest, cur\}, \{amount\}\}$ $wDep(a_2) = \{\{totalDays, lbf, ll, ld\}, \{fbf\}, \{fl\}, \{fd\}\}$
$hDep(a_0, e_1) = \{\{dest, cur\}, \{amount\}\}$ $hDep(a_2, e_3) = \{\}$
$aDep(a_0, e_1, a_2) = true$ $aDep(a_2, e_3, a_2) = false$

(c)

#	$\mathbf{WC}_{(\mathbf{dest,cur})}$	$\mathbf{WC}_{(\mathbf{amount})}$
1	Rome, Euro	100
2	Rome, Dollar	0
3	Rome, Pound	
4	London, Euro	
5	London, Dollar	
.	.	
.	.	
30	Berlin, Pound	

(d)

#	$\mathbf{WC}_{\mathbf{hDep(a_0,e_1)}}$
1	Rome, Euro, 100
2	Rome, Dollar, 0
3	Rome, Pound, 100
4	London, Euro, 0
5	London, Dollar, 100
.	.
.	.
30	Berlin, Pound, 100

(e)

#	$\mathbf{AT}_{\mathbf{hDep(a_0,e_1)}}$
1	{{Rome , Euro, 100}, ItemizedReport}
2	{{Rome , Dollar, 0}, ItemizedReport}
3	{{Rome , Pound, 100}, ItemizedReport}
4	{{London , Euro, 0}, ItemizedReport}
5	{{London , Dollar, 100}, ItemizedReport}
.	.
.	.
30	{{Berlin, Pound, 0}, ItemizedReport}

(f)

Figure 4: An example to illustrate TrimDroid's generation of tests: (a) input classes of widgets in `NewReportActivity`, (b) input classes of widgets in `ItemizedReportActivity`, (c) dependency sets for `NewReportActivity` and `ItemizedReportActivity`, (d) widget combinations for dependent widgets in a_0 with respect to e_1, (e) final set of combinations for Activity a_0 (`NewReportActivity`) in the context of event handler e_1, and (f) generation of Activity Tests for a_0 with respect to e_1.

$$AT_{hDep(a,e)} \equiv WC_{hDep(a,e)} \bigotimes e$$

For instance, in our running example, the set of test combinations for Activity a_0 (`NewReportActivity`) in relation to e_1 (`ItemizedReport.onClick`) is represented in Figure 4f. On the other hand, to test Activity a_2 (`ItemizedReportActivity`) in relation to e_3 (`Reset.onClick`) no value combinations are needed, as $WC_{hDep(a_2,e_3)} = \{\}$.

Tests for an Activity a can be calculated as the union of all generated tests in relation to its event handlers:

$$AT_a \equiv \bigcup_{\forall e \in eHandlers(a)} AT_{hDep(a,e)}$$

This section illustrated two ways in which TrimDroid reduces the number of generated tests. First, since TrimDroid has determined that $\{dest, cur\}$ is independent of $\{amount\}$, instead of calculating all their combinations by taking their cross-product, it simply merges the two sets of combinations (recall Definition 3). Second, since TrimDroid detects there are no dependencies between the *"Reset"* button's event handler and any of the widgets on `ItemizedReportActivity`, it does not generate tests involving any of the widget combinations for that particular event.

7.2 System Test Generation

To generate the GUI system tests ST for a given path $a_i \xrightarrow{e_p} a_j \xrightarrow{e_q} a_k$ in an ATM, we first generate the activity tests for each transition (event handler) in the manner described in the previous section. Next, if a_i and a_j are dependent with respect to e_p (i.e., $aDep(a_i, e_p, a_j) = true$), we enumerate all combinations of $AT_{hDep(a_i,e_p)}$ and $AT_{hDep(a_j,e_q)}$ by calculating their cross-product:

$$ST_{a_i \xrightarrow{e_p} a_j \xrightarrow{e_q}} \equiv AT_{hDep(a_i,e_p)} \bigotimes AT_{hDep(a_j,e_q)}$$

Otherwise, a_i and a_j are independent with respect to e_p, in which case we apply the merge operator, resulting in a reduction of generated tests:

$$ST_{a_i \xrightarrow{e_p} a_j \xrightarrow{e_q}} \equiv merge(AT_{hDep(a_i,e_p)}, AT_{hDep(a_j,e_q)})$$

Note that for transition $a_j \xrightarrow{e_q} a_k$ that ends with a final Activity a_k (e.g., a_4 in Figure 3), a_k has no activity tests as it has no outgoing transition, and in turn, contributes no combinations to the system tests.

To illustrate the process, consider a situation in ERS where the goal is to generate a system test for the path $a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_3} a_2$ in Figure 3. There is an activity dependency between a_0 and a_2 with respect to e_1, i.e., $aDep(a_0, e_1, a_2) = true$ as shown in Figure 4c. We thus create all combinations of activity tests for both activities in that transition to build the system tests ST, as shown in Figure 5.

#	$\mathbf{ST}_{\mathbf{a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_3} a_2}}$
1	$\langle\{\{Rome, Euro, 100\}, ItemizedReport\}, \{Reset\}\rangle$
2	$\langle\{\{Rome, Dollar, 100\}, ItemizedReport\}, \{Reset\}\rangle$
3	$\langle\{\{Rome, Pound, 100\}, ItemizedReport\}, \{Reset\}\rangle$
4	$\langle\{\{London, Euro, 100\}, ItemizedReport\}, \{Reset\}\rangle$
5	$\langle\{\{London, Dollar, 100\}, ItemizedReport\}, \{Reset\}\rangle$
.	.
.	.
30	$\langle\{\{Berlin, Pound, 100\}, ItemizedReport\}, \{Reset\}\rangle$

Figure 5: System tests for the path $a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_3} a_2$ in ERS

Our approach produces a total of 30 system tests for this path, each indicated as a sequence in the set ST. Following the generation of system tests, *Test-Case Generation* transforms each test case to proper Robotium format for execution [6].

8. EVALUATION

To evaluate our approach, we measure TrimDroid's ability to reduce test suites, while maintaining effectiveness. To assess effectiveness, we compare TrimDroid's code coverage and execution time against exhaustive combinatorial testing as well as prior Android testing techniques. Specifically, we investigate the following three research questions:

- **RQ1:** How do TrimDroid, exhaustive GUI combinatorial, and pairwise testing compare with respect to the size of generated test suites and their execution time?
- **RQ2:** How do TrimDroid, exhaustive GUI combinatorial, and pairwise testing compare with respect to code coverage?
- **RQ3:** How effective is TrimDroid compared to prior Android test automation techniques?

For investigating these questions, we use several real-world apps from an open-source repository, called F-Droid [4]. Each selected app satisfies the following criteria: (1) its source code is available; and (2) it uses only standard GUI widgets of the Android Application Development Framework, e.g., it is a *native mobile app*, rather than a *web mobile app*. The first criterion ensures that we can properly measure code coverage; the second criterion is due to the limitation of our static program analysis that only supports standard Android libraries and widgets. These apps are selected from different categories, such as productivity, entertainment, and tools.

Table 1 lists these apps. For each *App*, Table 1 depicts its size as measured using lines of code (*LOC*). We compare TrimDroid's coverage against exhaustive combinatorial testing, since its coverage *subsumes* the coverage of all other combinatorial testing techniques [20]. We used prime path coverage criterion (recall Section 6) for both exhaustive testing and TrimDroid to allow for a fair comparison.

We also compare TrimDroid's coverage with M[agi]C [32]—a pairwise GUI combinatorial testing technique that has been applied on Android apps, among others. M[agi]C requires the user to manually construct two types of models for the software under test: a model identifying the input classes for all the widgets, and a model that captures the transitions between the screens. In fact, the former is equivalent to TrimDroid's *ATM*, and the latter to its *IM*. To ensure a fair comparison, we manually transformed the *ATM* and *IM* models that TrimDroid generated automatically for each app into models that can be used by M[agi]C. M[agi]C uses a post-optimization algorithm that reduces the number of generated test cases after executing them once. This is achieved by removing the input combinations for paths that share events. Finally, we did not use the optimization option of M[agi]C to ensure the generated test cases achieve the maximum code coverage.

Although assessing TrimDroid's fault-detection ability is a primary concern of ours, currently there is no organized set of open-source Android apps with known defects and fault reports that can be used to evaluate TrimDroid's fault detection ability. Alternatively, mutation testing can be used, where the mutants replicate actual faults. Unfortunately, there is no support for mutation testing of Android apps to this date. Particularly, no fault model exists for Android apps, preventing production of mutants that can substitute for real faults. One of the authors has recently begun to investigate the challenges of mutation testing for Android applications [17].

All of our experiments were conducted on a machine with 16GB memory and a quad core 2.3GHz processor. We used Android Virtual Devices (Android emulators) with 2GB RAM, 1GB SD Card, and the latest version of Android that is compatible with the app, except for Dynodroid, whose in-box emulator uses Android 2.3. A fresh emulator was created for each app along with only default system applications. During the experiments, we used EMMA [3] to monitor code coverage. Specifically, we measured line coverage by running all of the generated test cases on each app. TrimDroid, subject apps, and our research artifacts are publicly available [7].

8.1 Test-Suite Reduction

To answer RQ1, we compare the test suites generated by TrimDroid, exhaustive combinatorial testing and M[agi]C in terms of size and execution time. For each *App*, Table 1 shows the size and execution time of test cases for both techniques. The table also shows the reduction of test cases compared to exhaustive combinatorial testing in the right-most column.

We observe that in most cases TrimDroid is able to significantly reduce the number of generated tests compared to exhaustive testing. TrimDroid, on average, generates 57.86% fewer tests compared to exhaustive testing. The smaller number of tests that would need to be inspected, especially for human engineers, would result in significant savings in time and effort. By doing so, TrimDroid reduces the time needed to execute the tests by 57.46% on average. The savings are more pronounced in certain cases. For PasswordGenerator, TrimDroid eliminates more than 479,000 tests, which is a reduction in tests by multiple orders of magnitude. Furthermore, exhaustive testing crashes for PasswordGenerator (as denoted by the dash in Table 1) before generating all the app's test cases, as the massive size of its generated test suite depletes our machine's memory.

On average, TrimDroid generates 2 times more test cases than M[agi]C. However, as described later, the tests generated by TrimDroid achieve a substantially higher code coverage. Recall that while TrimDroid adopts t-way testing, where *t* is determined according to the dependencies extracted through program analysis, M[agi]C uses a fixed *t* for all apps, i.e., two. Apps for which TrimDroid has produced more tests than M[agi]C, harbor complex dependencies involving three or more widgets. Apps for which TrimDroid has produced fewer tests are lacking dependencies among their widgets, indicating that the pairwise strategy is producing unnecessary tests.

8.2 Effectiveness - Exhaustive and Pairwise

To answer RQ2, we compare the statement coverage resulting from the execution of test suites generated by TrimDroid, M[agi]C and exhaustive testing. The results are summarized in Figure 6. Each application is identified along the horizontal axis, while the vertical axis shows the statement coverage achieved by TrimDroid, M[agi]C and exhaustive testing. In all cases, TrimDroid achieves at least the same statement coverage as exhaustive testing and the same or better statement coverage than M[agi]C. For the *Password-Generator* app, exhaustive testing's failure to complete is depicted as 0% coverage, since the inability to generate the test suite is effectively 0% statement coverage.

Note that some subject apps (e.g., *autoanswer* and *httpmon*) heavily use Service components. Unlike Activity components, Service components are responsible for handling events initiated by the system rather than the GUI. For example, *autoanswer* provides Services that perform a task based on a set of predefined preferences when a phone call is received. Given that TrimDroid's focus is on GUI testing, it is no surprise that it does not achieve good coverage for these types of apps. In fact, when we compare against the highest possible coverage for a GUI-based testing approach in such apps, namely exhaustive GUI testing, we observe that TrimDroid achieves the same coverage.

Thus, in comparison to exhaustive testing, the results show that, although TrimDroid significantly reduces the number of tests, and subsequently their execution time, the resulting code coverage is not degraded at all. In principle, however, due to the limitations of static analysis (e.g., unsupported Android libraries), it is possible for TrimDroid to achieve less coverage than exhaustive testing, even though our experiments have not yet revealed such instances.

On average, TrimDroid achieves 13% more statement coverage than M[agi]C. This result supports the effectiveness of using the proposed dependency-based heuristics for reducing the number of tests, rather than fixed strategies, such as pairwise testing, that compromise on coverage.

8.3 Effectiveness - Other Android Testing

A meaningful comparison of Android test automation techniques is generally difficult, as each has its own unique objective. Our objective in TrimDroid has been to *reduce the number of tests in combinatorial GUI testing of Android apps without compromising on coverage*. On the other hand, several prior techniques have aimed to maximize code coverage through search-based techniques, regardless of the number of tests it takes to do so, which could pose a significant burden when the assessment of whether the tests have passed or failed entails manual effort. Nevertheless, we compare against the code coverage and execution time achieved by four prior techniques: Monkey [1], Dynodroid [27], M[agi]C [32] and our prior work, EvoDroid [29].

Table 1: Pruning effect in TrimDroid

App	LOC	Exhaustive Testing		M[agi]C		TrimDroid		Reduction
		Test Cases	Time (s)	Test Cases	Time (s)	Test Cases	Time (s)	
HashPass	429	128	515	15	50	32	126	75.00%
Tipster	423	36	243	20	44	24	156	33.33%
MunchLife	631	10	84	9	37	8	56	20%
Blinkenlights	851	54	252	5	36	22	112	59.25%
JustSit	849	50	236	10	42	16	74	68%
autoanswer	999	576	5655	17	196	12	118	97.91%
AnyCut	1095	6	38	4	38	6	38	0%
DoF Calculator	1321	1174	7292	107	953	30	373	97.44%
Divide&Conquer	1824	12	85	5	47	4	32	66.66%
PasswordGene	2824	> 48000	–	58	351	418	1263	99.91%
TippyTipper	2953	26	238	30	172	25	225	3.84%
androidtoken	3680	454	13336	19	189	42	686	90.74%
httpMon	4939	42	407	34	235	28	282	33.33%
Remembeer	5915	48	633	24	194	17	320	64.58%

Android Monkey, a widely used testing technique developed by Google, represents the state-of-practice and operates by sending random inputs and events to the app under test. Dynodroid uses several heuristics to improve on the number of inputs/events used by Monkey, and thus achieves similar coverage with fewer generated events. As both Monkey and Dynodroid are based on pseudo-random testing, using the same low number of events that are generated by TrimDroid may not be a fair comparison. To address that, we ran both Dynodroid and Monkey with 2,000 input events, which is the maximum input size for Dynodroid [27].

EvoDroid is a system testing technique that implements a novel evolutionary testing algorithm. EvoDroid's fitness is designed to maximize the statement coverage through exhaustively exploring the search space for event sequences. Note that EvoDroid is a search-based testing technique; thus, using the same low number of events that are generated by TrimDroid is not adequate for the evolutionary search to be effective. On the other hand, the main goal of TrimDroid is to generate a limited number of tests, which is crucial when the evaluation of tests (i.e., oracle) involves manual effort. To summarize, EvoDroid is intended to exhaustively test event sequences; TrimDroid is designed to comprehensively test the input space of GUI widgets, using a limited number of event sequences identified by utilizing prime paths. Consequently, a fair one-to-one comparison of the two techniques might not be possible. Having said that, we ran EvoDroid for ten evolutionary generations on all apps, which is the same setup as that used in [29], and compare the resulting statement coverage.

Few other tools exist, but we were unable to include them in our experiments, as we could not properly run them after significant consultation with their developers. A^3E [12] aims to discover the Activities comprising an app by covering a model similar to our notion of *ATM*. We were unable to run A^3E on any of our apps using the virtual machine provided by its developers. A^3E gets stuck when trying to start Troyd [24]—an integration testing framework for Android utilized by the tool.

We also attempted to run *SwiftHand* [14]. It uses (1) machine learning to infer a model of the app during testing, (2) the inferred model to generate user inputs that visit unexplored states of the app, and (3) the execution of the app on the generated inputs to refine the model. SwiftHand exits with an exception failing to locate the main Activity of the app. Based on our analysis, the issue may reside with the custom made instrumentation of the app under test. Our attempts to resolve the issues with the help of the tool developers have been unsuccessful to date.

The statement coverage and execution time for all five testing techniques are summarized in Table 2. Dynodroid cannot run on *TippyTipper* and *DoF Calculator*—denoted by a dash (-) in Table 2—since the newer Android APIs utilized by those apps are not

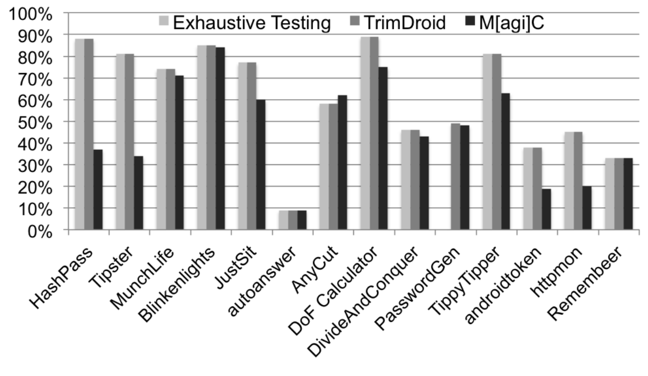

Figure 6: Statement coverage comparison

supported by Dynodroid. We could not run the current version of EvoDroid on *DivideAndConquer* due to use of unsupported APIs.

The results show that TrimDroid is able to achieve higher code coverage in most cases. Note that TrimDroid is targeted at GUI testing, and therefore only generates GUI events, while Dynodroid and EvoDroid support both system events as well as GUI events. As a result, for some apps, TrimDroid cannot achieve the same coverage as Dynodroid and EvoDroid. Nevertheless, TrimDroid's coverage, on average, outperforms Monkey by 27.36%, Dynodroid by 16.33%, M[agi]C by 14%, and EvoDroid by 4%.

In addition, TrimDroid runs 134 times faster than Dynodroid, 78 times faster than EvoDroid, 1.5 times slower than M[agi]C, and about 3 times slower than Monkey. TrimDroid's slower performance in comparison to M[agi]C and Monkey can be attributed to the analysis performed for extracting the models as well as the application of heuristics for reducing the number of tests.

9. RELATED WORK

In this section, we describe the most relevant research from two areas: Android testing and combinatorial testing.

9.1 Android Testing

The Android development environment ships with a powerful testing framework [2] that is built on top of JUnit. Robolectric [5] is another framework that separates the test cases from the device or emulator and provides the ability to run the tests directly by referencing the Android library. While these frameworks automate the execution of the tests, the test cases themselves still have to be manually developed.

Several prior approaches build on random testing techniques. Amalfitano et al. [8, 9] described a crawling-based approach that leverages completely random inputs to generate unique test cases. Hu and Neamtiu [21] presented a random approach for generating GUI tests that uses Android Monkey. Dynodroid [27] incorporates several heuristics to improve on Android Monkey's performance.

Several prior approaches have focused on the extraction of models for Android testing. *ORBIT* [38] is a grey-box model creation technique that creates a GUI model of the app for testing. A^3E [12] is a static taint analysis technique for building an app model for automated exploration of an app's Activities.

Unlike TrimDroid, these approaches focus on the construction of models for testing that are covered using a depth-first search strategy for generation of event sequences and random input data. Our work differs from them as we use prime path coverage, which subsumes all other graph coverage criteria, to generate the event sequences. We further generate the inputs for GUI widgets in a combinatorial fashion rather than using randomly generated input.

Choudhary et al. [15] compare several of the testing techniques

Table 2: Comparison of TrimDroid with other techniques

App	TrimDroid		M[agi]C		Monkey		Dynodroid		EvoDroid	
	Coverage	Time(s)	Coverage	Time(s)	Coverage	Time(s)	Coverage	Time(s)	Coverage	Time(s)
HashPass	88%	126	37%	50	40%	41	57%	33917	57%	23472
Tipster	81%	156	34%	44	67%	104	59%	33825	89%	22813
MunchLife	74%	56	71%	37	49%	75	54%	31421	78%	20965
Blinkenlights	85%	112	84%	36	49%	49	81%	25278	63%	22418
JustSit	77%	74	60%	42	35%	163	53%	41252	84%	20391
autoanswer	9%	118	9%	196	6%	43	10%	59672	8%	21883
AnyCut	58%	38	62%	38	6%	71	66%	21757	84%	19735
DoF Calculator	89%	373	75%	953	43%	70	-	-	36%	20713
DivideAndConquer	46%	32	43%	37	51%	58	83%	33644	-	-
PasswordGen	49%	1263	48%	351	42%	128	33%	31882	62%	26129
TippyTipper	81%	225	63%	172	42%	106	-	-	82%	23108
androidtoken	38%	686	19%	189	10%	67	11%	57813	29%	19188
httpmon	45%	282	20%	235	4%	46	6%	22563	36%	23403
Remembeer	33%	320	33%	194	27%	46	23%	53013	28%	22517

mentioned above by evaluating them according to four criteria: code coverage, fault detection capabilities, ease of use, and compatibility with different Android SDK versions. Their results shows that Monkey outperforms other studied approaches along the four mentioned criteria. They suggest that using a combination of these approaches may result in better performance. To that end, TrimDroid utilizes a combination of a model-based testing approach and a combinatorial approach.

Jensen et al. [23] presented a system testing approach that combines symbolic execution with sequence generation. The goal of their work is to find valid sequences and inputs to reach prespecified target locations, while TrimDroid aims to maximize the code coverage. Anand et al. [11] presented an approach based on concolic testing of a particular Android library to identify the valid GUI events using the pixel coordinates. Unlike our research, their approach does not generate systems tests, nor do they generate the tests in a combinatorial fashion. Finally, in our own prior research, we have developed techniques based on evolutionary search [28,29], as well as symbolic execution [30,31] for testing Android apps. Unlike our prior work, TrimDroid is targeted at GUI testing and explores a new combinatorial testing approach.

9.2 Combinatorial Testing

Combinatorial testing has shown to be an effective approach in GUI-based testing [39]. Approaches such as [18, 36] propose using greedy or heuristic algorithms to generate minimal sets of tests for a given combinatorial criteria. Nguyen et al. [32] proposed an approach that leverages manually constructed behavioral models of an app in pairwise testing of GUI-based applications. Kim et al. [25] introduced the idea of using static analysis to filter irrelevant features when testing software product lines. Petke et al. [34] showed that higher strength of t-way testing can be practical and more effective in the presence of constraints.

Our work differs from these approaches as we (1) specifically target Android apps, (2) automatically extract the models through program analysis, (3) use prime paths to generate the sequences of events, and (4) rely on a number of heuristics to determine the interacting widgets in order to reduce the number of tests without degrading the coverage.

10. CONCLUSION AND FUTURE WORK

We presented a fully-automated approach for generating GUI system tests for Android apps using a novel combinatorial technique. Our approach employs program analysis to partition the GUI input widgets into sets of dependent widgets. Thus, the GUI widgets with dependencies become candidates for combinatorial testing. We then use an efficient constraint solver to enumerate the test cases covering all possible combinations of dependent GUI widgets. Our experimental evaluation shows that TrimDroid is able to significantly reduce the number of tests in comparison to exhaustive combinatorial testing, without any degradation in code coverage.

While our program analysis heuristics have shown to be quite effective in pruning the test combinations, they have some known limitations. For example, the *IM* is constructed by analyzing the layout XML files statically, which does not handle cases where the Activity views are defined dynamically. In addition, our static analysis is subject to false negatives in certain rare cases, e.g., dependencies due to widget values being stored/read from the SD card, or dependencies occurring through global variables. Our future research involves improving the analysis to support such cases.

Recall from Section 4 that our current implementation uses a predefined set of input classes for unbounded widgets. In our future work, we plan to extend our previous work [30, 31] on symbolic evaluation of Android apps to systematically derive the input classes for those widgets.

The premise of our work is that the only available resource for automated testing is an app's APK file. If an app's specification is also available (e.g., in a formal machine-interpretable format), one could investigate the extraction of interactions from the app's specification as well as its implementation for combinatorial test generation. In practice, most apps on the market lack specifications that can be used effectively for automated testing. However, this means that if two widgets should have a dependency according to the intended specification of the software, but the implemented software does not have such a dependency, possibly because the developer failed to correctly realize the specification, TrimDroid would not generate tests to exercise those combinations.

In this work, we have focused on automatic test case generation, rather than automatic *test oracle* construction, resulting in oracles that determine whether test cases pass or fail. We believe that it will be necessary to have the user in the loop to generate oracles that assess intended app behaviors. Hence, reducing the number of test cases to be inspected is certainly beneficial. Moreover, the ability to generate tests that can achieve high code coverage has applications beyond testing for functional defects: Energy issues, latent malware, and portability problems are important concerns in the context of mobile devices that are often effectively detected by executing the code.

11. ACKNOWLEDGMENTS

This work was supported in part by awards CCF-1252644 from the National Science Foundation, D11AP00282 from the Defense Advanced Research Projects Agency, W911NF-09-1-0273 from the Army Research Office, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

12. REFERENCES

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

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

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

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

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

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

[7] Trimdroid, https://seal.ics.uci.edu/projects/trimdroid/.

[8] 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.

[9] 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, New York, NY, USA, 2012. ACM.

[10] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[11] 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, New York, NY, USA, 2012. ACM.

[12] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[13] A. Bartel, J. Klein, Y. LeTraon, and M. Monperrus. Dexpler:converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. of SOAP*, 2012.

[14] 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, New York, NY, USA, 2013. ACM.

[15] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet. In *To appear at the 30th International Conference on Automated Software Engineering*, ASE '15, 2015.

[16] R. Cozza, I. Durand, and A. Gupta. Market Share: Ultramobiles by Region, OS and Form Factor, 4Q13 and 2013. *Gartner Market Research Report*, February 2014.

[17] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *To appear in Proceedings of the 2015 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '15. IEEE, 2015.

[18] P. Flores and Y. Cheon. Pwisegen: Generating test cases for pairwise testing using genetic algorithms. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 2, pages 747–752, June 2011.

[19] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Event listener analysis and symbolic execution for testing gui applications. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 69–87, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[21] 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, New York, NY, USA, 2011. ACM.

[22] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[23] 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, New York, NY, USA, 2013. ACM.

[24] J. Jeon and J. S. Foster. Troyd: Integration testing for android. Technical Report CS-TR-5013, Department of Computer Science, University of Maryland, College Park, August 2012.

[25] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

[26] Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society.

[27] 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, New York, NY, USA, 2013. ACM.

[28] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 22 –28, June 2012.

[29] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, Hong Kong, China, November 2014. ACM.

[30] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *Proceedings of the 26th International Symposium on Software Reliability Engineering*, ISSRE '15, Washington, DC, November 2015. IEEE.

[31] 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.

[32] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM.

[33] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, Feb. 2011.

[34] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 26–36. ACM, 2013.

[35] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, SE-11(4):367–375, April 1985.

[36] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 49–59, Washington, DC, USA, 2004. IEEE Computer Society.

[37] R. Valle é-Rai, P. Co, E. Gagnon, L. Hendren, and V. Lam, P.and Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, CASCON'99*, 1999.

[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, Berlin, Heidelberg, 2013. Springer-Verlag.

[39] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated gui testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 405–408, New York, NY, USA, 2007. ACM.