DetReduce: Minimizing Android GUI Test Suites for Regression Testing

Wontae Choi†
University of California, Berkeley
wtchoi@cs.berkeley.edu

George Necula
University of California, Berkeley
necula@cs.berkeley.edu

Koushik Sen
University of California, Berkeley
ksen@cs.berkeley.edu

Wenyu Wang†
University of Illinois, Urbana-Champaign
wenyu2@illinois.edu

ABSTRACT
In recent years, several automated GUI testing techniques for Android apps have been proposed. These tools have been shown to be effective in achieving good test coverage and in finding bugs without human intervention. Being automated, these tools typically run for a long time (say, for several hours), either until they saturate test coverage or until a testing time budget expires. Thus, these automated tools are not good at generating concise regression test suites that could be used for testing in incremental development of the apps and in regression testing.

We propose a heuristic technique that helps create a small regression test suite for an Android app from a large test suite generated by an automated Android GUI testing tool. The key insight behind our technique is that if we can identify and remove some common forms of redundancies introduced by existing automated GUI testing tools, then we can drastically lower the time required to minimize a GUI test suite. We have implemented our algorithm in a prototype tool called DetReduce. We applied DetReduce to several Android apps and found that DetReduce reduces a test-suite by an average factor of 16.9× in size and 14.7× in running time. We also found that for a test suite generated by running SWIFT-SWARM and a randomized test generation algorithm for 8 hours, DetReduce minimizes the test suite in an average of 14.6 hours.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging;

KEYWORDS
Android, GUI, Test minimization

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 ’18, May 27–June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05...$15.00
https://doi.org/10.1145/3180155.3180173

1 INTRODUCTION
In recent years, there has been a significant surge in the usage and development of apps for smartphones and tablets. Developers are writing more apps for mobile platforms than for desktops. The complexity of mobile apps often lies in their graphical user interfaces (GUIs). Testing efforts of such apps mostly focus on the behavior of graphical user interfaces.

Several automated GUI testing techniques have recently been proposed. The techniques include learning-based testing [8, 29, 31, 32], model-based testing [1, 23, 45], genetic programming [27, 28], fuzz testing [25, 26, 37], and static-analysis based approaches [4, 33, 34, 44, 49]. The goal of the majority of these techniques is to achieve good code and screen coverage (i.e. covering all distinct screens of an app), and to find common bugs such as crashes and unreliability. Most of these techniques work by injecting sequences of automatically generated user inputs or actions to an app for several tens of hours. We consider each sequence of actions injected by these techniques to be a test case, and the set of all sequences of actions to be a test suite.

Although automated GUI testing techniques could find bugs, they tend to generate large test suites containing thousands of test cases. Each test case can contain tens to thousands of user actions. Such a large test suite can take several hours to execute, because the running time of a test suite is linear in the size of the test suite.1 However, regression tests should be fast so that they can be used frequently during development. Therefore, such test suites are difficult to use in regression testing.

In this paper, we address the problem of generating a small regression GUI test suite for an Android app. We assume that we are given a large test suite generated by an existing automated GUI testing tool. We also assume that the test suite is replayable in the sense that if we rerun the test suite multiple times we get the same coverage and observe the same sequence of app screens. (The evaluation section has details on how to obtain a replayable

1 For a GUI app, it is recommended to restrict the amount of computation performed by each event handler to improve responsiveness. Therefore, the running time of a GUI test case tends to be linear in the length of the test case.
We noticed that if our state abstraction is too coarse-grained our feasibility checks often fail, leading to longer running time. On the other hand, if we use a too fine-grained state abstraction, we fail to identify many redundancies. One contribution of this paper is to design a good enough abstraction that works well in practice.

One advantage of our algorithm over delta-debugging or other black-box algorithms is that we do not blindly generate all possible new test cases that can be constructed by dropping some actions. Rather, we use a suitable state abstraction to only drop potentially redundant loops. Another advantage is that we create new test cases by combining fragments from input test cases. This enables us to come up with new, longer test cases which cannot be generated using delta-debugging or other test suite reduction techniques. Longer test cases are usually better than multiple shorter test cases because we do not have to perform a clean restart of an app. A clean restart of an app requires us to kill the app, erase app data, and erase SD card contents, which is very time consuming. A longer test case in place of several shorter test cases avoids several such expensive restarts.

We have implemented our algorithm in a prototype tool, called DetReduce, for Android apps. The tool is publicly available at https://github.com/wtchoi/swifthand2. We applied DetReduce to several apps and found that DetReduce could reduce a test-suite by a factor of 16.2x in size and a factor of 14.7x in running time on average. We also found that for a test suite generated by running SwiftHAND [5] and a random testing algorithm [5] for 8 hours, DetReduce can reduce the test suite in an average of 14.6 hours. We are not aware of any existing technique that could get such huge reduction in the size of a large GUI test suite in such a reasonable amount of time. Note that DetReduce often runs longer than generating all test cases; however, running DetReduce is a one-time cost. Once a regression suite has been generated, it will be run many times and each run will take a fraction of the time required to generate all test cases.

2 OVERVIEW

Automated GUI testing tools, such as Monkey [12], A3E [2], DynamicDroid [26], Mobiguitar [1], and Orbit [45], explore the GUI of an app automatically without any prior knowledge about the behavior of the app. These automated tools are, however, not good at generating concise regression test suites that could be used for testing in incremental development of the apps and in regression testing. We propose a heuristic technique that helps to create a small regression test suite for a GUI app given a large test suite generated by an automated GUI testing tool. We next give a brief overview of our technique using formal notation and a series of examples.

2.1 Definitions and Problem Statement

Trace. The execution of an app on a sequence of user inputs can be denoted by a trace of the form \( s_0 a_1 c_1 s_1 a_2 c_2 \ldots a_n c_n s_n \). Each \( s_i \) is an abstract state of the program, usually computed by abstracting the screen of the app. Each \( s_{i-1} a c s_i \) is a transition, denoting that the app transitioned from state \( s_{i-1} \) to state \( s_i \) on user input (or action) \( a \) and covered the set of branches \( C_i \) during the transition. Several event handlers could be triggered during a transition: the branches covered during the transition are the branches of the triggered event handlers (and the methods transitively called from
these event handlers) that are executed during this transition. Here we focus on branch coverage, but one could use other kinds of coverage for $C_i$.

**Coverage.** If $s_{i-1} a_i C_i s_i$ is a transition, then $C_i \cup \{s_i\}$ is the coverage of the transition. In the coverage we include both the set of branches and the abstract states visited by the transition. We can similarly define the coverage of a trace $\tau = s_0 a_1 C_1 s_1 a_2 C_2 s_2 \ldots a_n C_n s_n$ as the union of the coverage of all the transitions in the trace, i.e. $C(\tau) = \bigcup_{i \in [1,n]} (C_i \cup \{s_i\})$.

**Replayable traces.** In our technique, we are only interested in replayable traces. A trace $\tau = s_0 a_1 C_1 s_1 a_2 C_2 s_2 \ldots a_n C_n s_n$ of an app is replayable if every time the app is given the sequence of user actions $a_1, a_2, \ldots, a_n$ in a state $s_0$ (the initial state of the app), it generates the exact trace $\tau$.

**Test suite: A set of replayable traces.** We assume that an automated testing tool for GUI generates a set $T_0$ of replayable traces that can be treated as a regression test suite. The coverage of a set of traces $T$, denoted by $C(T)$, is defined as the union of the coverage of the traces contained in the set. The cost of a set of traces $T$ is the pair $(\sum_{\tau \in T} |\tau|, |T|)$. The first component of the pair gives the number of transitions present in the traces in $T$. This number roughly estimates the amount of time necessary to replay the traces in $T$. Between the replay of two traces, one needs to perform a clean restart of the app by erasing the app data and SD card contents, which has high cost. In order to take that cost into account, we have a second component in the pair corresponding to the number of clean restarts necessary to replay all the traces in $T$.

**Problem statement.** Given a set of replayable traces $T_0$, the goal is to find a minimal set of traces $T_0$ such that $T_0$ is replayable, $T_0$ consists of transitions from the traces in $T_0$, $C(T_0) = C(T_0)$, and the cost of $T_0$ is minimal. Unfortunately, finding a minimal $T_0$ is intractable for the following reasons. First, without the replayability requirement, the problem can be reduced to an instance of the prize-collecting traveling salesman problem (PCTSP), a well-known NP-hard problem [3]. With the replayability requirement, a solution found by solving the corresponding PCTSP problem may include non-replayable traces. Therefore, we need to solve multiple PCTSP problems until finding a replayable solution. Instead of solving the problem of finding the global minimum, we developed a two-phase heuristic algorithm, which we found to work effectively in practice.

### 2.2 Limitations of Existing Approaches

In any test-case reduction technique, we need to construct new traces. Although the creation of a trace takes little time, we have to ensure that the trace can be replayed. It is impossible to precisely determine if a trace is replayable. In our technique, we check if a trace is replayable by executing it few times. We found experimentally that if a trace is non-replayable, it will fail to replay within eight executions with very high probability. Faithfully executing a single transition in a trace could take a few seconds because after injecting an input or action, we need to wait until the screen stabilizes. Therefore, executing a trace composed of several transitions could take several minutes. Moreover, after executing each trace we need to perform a clean restart, which takes several seconds. Therefore, it is generally time consuming to check if a trace is replayable. This is the key bottleneck faced by a GUI test suite reduction technique.

![Figure 1: A partial model of a file browser app.](image-url)

Existing test minimization techniques, such as delta-debugging [48] and genetic algorithms [27], will create and check the replayability of many traces. Therefore, these techniques cannot scale when the initial set of traces is large. But all existing automated techniques for GUI test generation create large numbers of traces. According to Clapp et al. [7]'s experimental results, their variant of delta-debugging can take a few hours to several tens of hours to handle traces composed of only 500 transitions. In our experiments, we had to handle test cases having 10,000 transitions. If we linearly extrapolate the timings reported by Clapp et al. to 10,000 transitions, delta-debugging could potentially take a month. We seek a technique that can minimize a test suite in a day or less.

### 2.3 Our Observations

We observed that the set of traces generated by an automated testing tool has many redundancies. Our technique for GUI test suite reduction tries to remove these redundancies. We next describe these redundancies using a series of examples.

#### 2.3.1 Redundant Traces

Among the traces in a test suite, the coverage provided by some traces is a subset of coverage provided by the remaining traces. Such traces can be removed from the test suite without decreasing the cumulative coverage. Our technique finds such redundant traces and removes them from the test suite.

#### 2.3.2 Redundant Loops

We also observed that there could be redundancies within a trace, for example, if the trace contains a redundant loop. A loop in a trace is a sub-trace of the trace that begins and ends in the same abstract state. Traces generated by automated testing tools tend to contain many loops, and some such loops do not provide additional coverage over the coverage that can be achieved by the trace without the loop and the remaining traces. Such loops are redundant and can be removed from the trace if the resulting trace can be replayed. We next illustrate such redundant loops using a couple of examples.

All examples utilize the file browser app shown in Figure 1. When the app starts, it shows the root directory (abstract state/screen $s_0$). In this initial state/screen, a user can invoke a pop-up menu ($s_1$) by touching the menu button on the screen (the button at the top-right corner with three dot characters). Once the menu is visible, the user can close the menu by touching the same button. Selecting an item in the menu (screen ($s_2$)), the user can invoke a pop-up menu ($s_3$) by touching the menu button on the screen (the button at the top-right corner with three dot characters). Once the menu is visible, the user can close the menu by touching the same button. Selecting an item on the menu results in a completely different abstract state/screen of the app. Pressing the Option button leads the app to the option screen ($s_4$). The app also allows the user to navigate the file system (abstract state/screen $s_5$ and $s_3$). Note we intentionally made the directories /foo and /foo/foo to have the same look in order to keep the example simple.
Example 2.1. (A redundant loop) Assume that the user touches the menu button three times. This input sequence will open the pop-up menu, close it, and then open it again. The user then selects the first item (i.e. the Option button) of the menu. This action will lead the app to the configuration screen. Assume there are no event handlers associated with the menu open and close events. The execution of the above action sequence will then generate the following trace: $s_0 \xrightarrow{\text{Menu}} s_1 \xrightarrow{\text{Menu}} s_0 \xrightarrow{\text{Option}} s_3$, where $\emptyset$ is the empty set and $C_o$ denotes the test coverage generated when the app moves to the configuration screen. In this trace, the sub-trace $s_0 \xrightarrow{\text{Menu}} s_1 \xrightarrow{\text{Menu}} s_0$ forms a loop since it begins and ends with the same abstract state. The coverage of this loop only contains the states $s_1$ and $s_0$. The coverage of the loop is a subset of the coverage of the rest of the trace ($C_o \cup \{s_0, s_1, s_2\}$). Thus, the loop does not add any extra coverage than what is already achieved by the rest of the trace. This makes sense because the loop merely opens and closes the pop-up menu. After removing the loop from the trace, if the modified trace is replayable, we can replace the original trace with the modified trace. Removing the loop gives us the following shorter replayable trace: $s_0 \xrightarrow{\text{Menu,$C_o$}} s_1 \xrightarrow{\text{Option,$C_o$}} s_2$.

Example 2.2. (A non-redundant loop) A loop is non-redundant when the loop provides coverage that cannot be achieved by the rest of the trace(s). Let us assume that the app now has event handlers attached to the menu open and close events. Re-executing the same sequence of actions from Example 2.1 will generate the following slightly different trace: $s_0 \xrightarrow{\text{Menu,$C_p$}} s_1 \xrightarrow{\text{Menu,$C_o$}} s_0 \xrightarrow{\text{Option,$C_o$}} s_2$, where $C_p$ and $C_o$ denote the coverage generated by executing the menu open and close event handlers, respectively. In this modified trace, the loop contributes the test coverage $C_o$ which cannot be achieved by any other transition in the trace. Therefore, the loop should not be removed from the trace.

Example 2.3. (Another non-redundant loop) A loop can be non-redundant if the removal of the loop makes the trace non-replayable, even if it does not achieve any new coverage. Note that if we use the concrete state of the app instead of a screen abstraction, a trace will be replayable if we remove a loop. However, due to abstraction, the start and end states of a loop may not correspond to the same concrete state. Therefore, the trace may not be replayable after the removal of a loop. Let us illustrate this with an example. This time, we are going to navigate the file system to reach the folder containing pictures (i.e. state $s_3$). This task can be done by simply touching the /foo folder three times. The execution of the sequence of actions will generate the following trace: $s_0 \xrightarrow{\text{foo,$C_f$}} s_1 \xrightarrow{\text{foo,$C_f$}} s_0 \xrightarrow{\text{foo,$C_f$}} s_3$, where $C_f$ denotes the test coverage generated when opening a folder only containing subfolders, and $C_{f_2}$ denotes the test coverage generated when opening a folder only containing files. The trace has three loops (the first transition, the second transition, and the sub-trace containing the first two transitions). The third loop cannot be removed because removing it will reduce the coverage of the trace. The first and seconds loops, however, look identical and one may think that one of the two loops can be removed from the trace. However, removing one of these two loops will make the trace non-replayable because touching the /foo folder twice leads the app to the screen showing the contents of /foo/ /foo folder and we will miss $C_{f_2}$. The trace obtained after removing one of the loops is non-replayable because our coarse-grained state abstraction maps three distinct app states to $s_0$. However, if we do not use the abstraction, we will have unbounded number of states, which will make both automated test generation and test minimization fail. This example shows that a loop is non-redundant if its removal makes the trace non-replayable. Note that determining whether removing a loop will have an impact on the rest of a trace can only be determined by trying to replay the modified trace.

2.3.3 Redundant Trace Fragments. While analyzing traces in automatically generated test suites of several apps, we observed that many traces share common sub-traces. If we execute these traces, the common sub-traces get executed multiple times (once for each trace) but new coverage is only achieved when a common sub-trace is executed for the first time. We can avoid redundant execution of these common sub-traces if we can combine fragments of traces in a manner that avoids repetitions of common sub-traces. Combining fragments of traces will also result in longer traces. Such longer traces will reduce the number of restarts, which are more expensive operations than triggering an action. The next example describes how common sub-traces contribute to redundancy.

Example 2.4. (Splicing three traces) Consider the following three artificially-crafted traces:

1. $s_0 a, c_1 s_1 b, c_2 s_2 c, c_3 s_3 d, c_4 s_4$
2. $s_0 a, c_1 s_1 d, c_3 s_3 e, c_5 s_5 f, c_6 s_6$
3. $s_0 a, c_1 s_1 b, c_2 s_2 c, c_3 s_3 f, c_5 s_5$

Note the first and second traces have two common sub-traces: $s_0 a, c_1 s_1 b, c_2 s_2 c, c_3 s_3$. Similarly, the first and third traces have the common prefix $s_0 a, c_1 s_1 b, c_2 s_2 c, c_3 s_3$. By combining fragments of the traces, we can create the following trace: $s_0 a, c_1 s_1 b, c_2 s_2 c, c_3 s_3 f, c_5 s_5$. The spliced trace is constructed by appending sub-trace $s_4 e, c_5 s_5 f, c_5 s_5$ to the first trace, and then by appending the sub-trace $s_3 f, c_5 s_5$ to the resulting trace. The new trace gets rid of six actions and two restart operations from the original traces. Note that the spliced trace still contains two copies of the sub-trace $s_2 e, c_3 s_3$, which we could not get rid of. If the spliced trace is replayable, it can replace the original traces in the test suite. The running time of the spliced trace will be approximately half of the original traces, while providing the same coverage. This example shows that we can aggressively combine fragments from multiple traces while getting rid of redundant fragments (including redundant prefixes). However, in practice, we found that traces composed of a large number of fragments from different traces tend to be non-replayable. Therefore, in our technique we limit the number of different trace fragments that we can combine to a small bound, which is three in our implementation.

2.4 Our Approach

State abstraction. In our discussion so far, we assumed we have a suitable state abstraction that enables us to cluster similar-looking screens. The performance of our technique for test reduction depends heavily on our choice of state abstraction. If we choose a fine-grained abstraction, then our technique runs faster, but may miss many opportunities for reduction. On the other hand, if we
pick a coarse-grained abstraction, many of the traces that our technique constructs become non-replayable. Therefore, our technique spends more time in checking replayability of various traces, but we get a bigger reduction. We observed that a human tester can easily identify screens that are similar by analyzing what is visible on the screen. Ideally we needed an abstraction that judges that two app screens to be the same if and only if a human tester finds the two screens visually identical. After analyzing several apps, we picked a state abstraction based on information from the GUI component tree. The details of the abstraction are described in Section 4.

Removing redundancies. We propose a two-phase algorithm to remove redundancies from a GUI test suite. The first phase removes redundant traces and redundant loops greedily. It first removes redundant traces by greedily selecting traces such that each selected trace contributes new coverage to the coverage of the set of already selected traces. The non-selected traces are then redundant and are removed from the test suite. It then removes redundant loops from each remaining trace. In order to remove redundant loops in a trace, the algorithm creates the set of all traces obtained from the trace by removing zero or more loops. It then selects a trace from the set that does not decrease cumulative coverage, lowers cost of the trace maximally, and is replayable. Such a trace replaces the original trace in the test suite.

The second phase removes redundant trace fragments as much as possible. For that it constructs a new set of traces by combining fragments of the traces in the set computed by the first phase. When splicing trace fragments, we found it useful to limit the number of fragments in spliced trace to a small number (three in our case), because a trace composed of many fragments tends to be non-replayable in practice. Thus, the second phase of the algorithm first creates the set of candidate traces composed of a bounded number of trace fragments. It then constructs a new test suite by greedily selecting traces from the set of candidate traces.

In both phases, whenever our algorithm generates a new trace, it checks whether the trace is replayable or not by executing it a few times. This prevents the algorithm from adding a non-replayable trace to the resulting regression test suite. If the algorithm finds a trace to be non-replayable, it identifies and saves the shortest non-replayable prefix of the trace. In the future, if the algorithm finds that a new trace starts with one of these saved prefixes, then it can safely infer that the trace is non-replayable and discard it. This optimization helps the algorithm aggressively discard some non-replayable traces without executing them multiple times. We describe the reduction algorithm formally in the next section.

3 ALGORITHM

3.1 Redundant Loop and Trace Elimination

In order to construct a minimal set of traces, we only retain the traces from $T_0$ whose cumulative coverage is same as the coverage of $T_0$. Then we remove as many loops from the remaining traces as possible while maintaining the same cumulative coverage and the replayability of the traces. This results in a set of traces $T_R$ whose cost is much lower than the cost of $T_0$. During the removal of loops, our algorithm discovers that certain trace prefixes are not replayable. We speedup the loop elimination phase by pruning out the traces whose prefix matches the non-replayable prefixes. We next describe this algorithm formally.

Given a trace $\tau$, we say that a sub-trace $\tau'$ of $\tau$ is a loop if it begins and ends in the same state. For example, if in the trace $\tau = \tau_0 a_1 \tau_1 a_2 \tau_2 a_3 \tau_3 \ldots a_n \tau_n \gamma_n$, there exists two states $s_i$ and $s_j$ such that $i \neq j$ and $s_i = s_j$, then the sub-trace $\tau' = s_i a_{i+1} \tau_{i+1} a_{i+2} \tau_{i+2} a_{i+3} \tau_{i+3} \ldots a_j \tau_j$ is a loop. If we remove the loop from $\tau$, we get a shorter trace $\tau' = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots a_n \tau_n \gamma_n$. A trace obtained after eliminating one or more loops from $\tau$ may no longer be replayable. Thus, any time our algorithm removes a loop from a trace, we need to check if the resulting trace is still replayable. Let $L(\tau)$ be the set of all traces obtained by removing different combinations of zero or more loops from $\tau$. Note that $L(\tau)$ contains $\tau$.

The pseudocode of the algorithm is shown in Algorithm 1. The algorithm uses a function $\text{Replay}$ which takes a trace $\tau$ and returns $\tau$ if the trace is replayable; otherwise, the function returns the shortest prefix of $\tau$ that is not replayable. Therefore, the check $\tau = \text{Replay}(\tau)$ tell us whether $\tau$ is replayable or not. In the first part of the algorithm, we remove all redundant traces. To do so we create an empty set $T$ to store the non-redundant traces. The algorithm goes over each trace $\tau$ in $T_0$. If $C(\tau)$ has some coverage that is not already present in $C(T)$, then $\tau$ is not redundant and we add $\tau$ to $T$. After going over all traces in $T_0$, $T$ will contain non-redundant traces of $T_0$ such that $C(T) \subseteq C(T_0)$.

In the second part, the algorithm performs redundant loop elimination. It maintains a set of reduced traces $T_R$ which is initialized to the empty set. The algorithm goes over each trace $\tau$ in $T$. It then goes over each trace $\tau'$ in $L(\tau)$ (the set of all traces obtained from $\tau$ by removing zero or more loops) in the increasing order of cost. If $C(\tau') \cup C(T_R) = C(\tau) \cup C(T_R)$ and $\tau'$ is replayable, i.e. if $\tau' = \text{Replay}(\tau')$, the algorithm adds $\tau'$ to $T_R$ and stops processing elements of $L(\tau)$. This indicates that the algorithm has computed a trace possibly shorter than $\tau$. On the other hand, if $\tau'$ is not replayable, then any trace in $L(\tau)$ having $\text{Replay}(\tau')$ as a prefix is removed from the set $L(\tau)$ because all such traces will also be non-replayable. This reduces the number the traces that we have to process from the set $L(\tau)$ because all such traces will also be non-replayable. Thus, the algorithm adds $\tau'$ to $T_R$ and stops processing elements of $L(\tau)$. This indicates that the algorithm has computed a trace possibly shorter than $\tau$. On the other hand, if $\tau'$ is not replayable, then any trace in $L(\tau)$ having $\text{Replay}(\tau')$ as a prefix is removed from the set $L(\tau)$ because all such traces will also be non-replayable. This reduces the number the traces that we have to process from the set $L(\tau)$ because all such traces will also be non-replayable. Thus, the algorithm adds $\tau'$ to $T_R$ and stops processing elements of $L(\tau)$. This indicates that the algorithm has computed a trace possibly shorter than $\tau$. On the other hand, if $\tau'$ is not replayable, then any trace in $L(\tau)$ having $\text{Replay}(\tau')$ as a prefix is removed from the set $L(\tau)$ because all such traces will also be non-replayable. This reduces the number the traces that we have to process from the set $L(\tau)$ because all such traces will also be non-replayable. Thus, the algorithm adds $\tau'$ to $T_R$ and stops processing elements of $L(\tau)$.

Practical concerns. The algorithm relies on a robust implementation of $\text{Replay}(\tau)$. However, in practice it is not easy to have a precise implementation of $\text{Replay}(\tau)$ that will guarantee that $\text{Replay}(\tau)$ returns $\tau$ if and only if $\tau$ is replayable. Such an implementation would require us to track the entire state of the app including the state of any internet server it might be interacting with. Moreover, if we make the implementation of $\text{Replay}(\tau)$ too precise, in many acceptable cases it will report that $\tau$ is not replayable. In our tool, we make a practical trade-off where we re-execute the trace $\tau$ a few times, which is ten in our experiments, and report the shortest prefix of $\tau$ that is non-replayable over all ten re-executions. If in all the ten executions we find that $\tau$ is replayable, $\text{Replay}(\tau)$ returns $\tau$.

The algorithm also needs to compute $L(\tau)$, i.e. the set of traces obtained from $\tau$ by removing 0 or more loops. Our implementation does not compute the set $L(\tau)$ ahead of time. Rather it performs a
Algorithm 1 Eliminate redundant traces and loops

1: procedure EliminateRedundantTracesAndLoops($T_k$)
2: $T ← ∅$ ▷ Part 1: Eliminate redundant traces.
3: for $τ ∈ T_k$ do ▷ Loop over the input trace set.
4: if $C(τ) ⊈ C(T)$ then
5: $T ← T ∪ \{τ\}$
6: $T_r ← ∅$ ▷ Part 2: Eliminate redundant loops.
7: for $τ ∈ T$ do ▷ Loop over the filtered trace set $T$.
8: $T_L ← L(τ)$
9: while $T_L ≠ ∅$ do
10: $τ’ ← \text{argmin}_ι |τ|$ w.r.t. $T$.
11: $T_L ← T_L \setminus \{τ’\}$
12: if $C(τ’) ∪ C(T) = C(T)$ then
13: if $τ’ = \text{Replay}(τ)$ then
14: $T_r ← T_r ∪ \{τ’\}$
15: break the inner loop
16: else
17: $T_L ← \{τ ∈ T_L \mid \text{Replay}(τ) \text{ is not a prefix of } τ\}$
18: return $T_r$

depth-first traversal of the trace $τ$ to enumerate the traces in $L(τ)$ one-by-one from the shortest to the longest one.

Finally, the result of the first phase of the algorithm will change depending on the order of picking elements from $T_k$ (line 3) and $T$ (line 7). Our implementation uses queues to store test cases, which guarantees that test cases are always handled in first-come-first-serve order. We tried various other orderings, but we observed that the results did not vary significantly.

3.2 Trace Splicing

While analyzing traces in the set $T_r$, i.e. the traces generated by loop and trace elimination, we noticed traces often share common sub-traces. Therefore, if we can combine traces in order to avoid common sub-traces as much as possible, we generate longer traces. This is good, since longer traces avoid expensive restarts and avoid execution of redundant sub-traces. However, we also found that the more traces we combine, the more likely we are to get non-replayable traces. We found experimentally that if we combine three or fewer trace fragments, we could still reap the rewards of longer traces (avoiding restarts and redundant execution) while creating replayable traces. Based on these observations, we devised the second part of our minimization algorithm where we combine fragments from different traces to create longer replayable traces.

A trace fragment is a contiguous portion of a trace obtained by removing a prefix and a suffix of the trace. For example, if $τ = s_0s_1C_1s_2C_2 . . . s_nC_ns_n$ is a trace, then for any $i, j \in [0, n]$ where $i ≤ j$, $s_i . . . s_{i+1}C_{i+1} . . . s_jC_{j+1}$ is a fragment of the trace $τ$. A set of trace fragments $τ_1, τ_2, . . . , τ_m$ can be combined to form the trace $τ_1τ_2 . . . τ_m$ if $τ_1$ begins with the state $s_0$ and for all $i \in [2, m]$, the end state of $τ_{i−1}$ is the same as the first state in $τ_i$. Given a set of traces $T_r$, let $T_k$ be the set of all traces obtained by combining at most $k$ trace fragments from the traces in $T_r$.

The pseudocode of the algorithm is shown in Algorithm 2. The algorithm first constructs the set $T_k$ from the set $T_r$. The algorithm initializes the final set of minimized traces $T_m$ to the empty set. The algorithm then does the following in a loop: it finds a trace $τ$ in $T_k$ such that $τ$ results in the maximal increase in coverage over the coverage of $T_m$, i.e. $τ$ maximizes $|C(τ) \setminus C(T_m)|$. If no such trace is found in $T_k$, the algorithm returns $T_m$. Otherwise, if $τ$ is replayable, it removes $τ$ from $T_k$ and adds it to $T_m$. If $τ$ is not replayable, then all traces in $T_k$ having $\text{Replay}(τ)$ as a prefix are removed from $T_k$. This step speeds up the search for optimal $τ$ in future iterations. The loop is then repeated.

The above algorithm terminates and computes a $T_m$ such that $C(T_m) = C(T_r)$. The algorithm terminates because in each iteration we increase the coverage of $T_m$, and the coverage of $T_m$ cannot be increased beyond the coverage of $T_r$ (which is same as the coverage of $T_k$). The algorithm also ensures that $C(T_m) = C(T_r)$ because for any finite $k$, $T_k$ contains the traces in $T_r$. Therefore, in the worst case if none of the traces obtained by combining two or more trace fragments from different traces are replayable, we will end up adding all the traces in $T_r$ to $T_m$. This ensures that $C(T_m) = C(T_r)$.

Computing $T_k$. Algorithm 2 uses a declarative specification to describe the trace set $T_k$. We next describe an algorithm to compute the set efficiently. For any set of traces, we can construct a labeled transition system composed of the transitions of the traces in the set. Formally, if $T_r$ is a set of traces, we construct a labeled transition system $Q_{T_r} = (S, s_0, L, A, C, δ)$, where

- $S$ is the set of all states in $T_r$, • $s_0$ is the initial state of the app,
- $L \subseteq S \times N \times N$ is a set of labels, • $A$ is the set of all actions in $T_r$,
- $C$ is the set of coverage sets in $T_r$, and
- $δ$ is a set of labeled transitions of the form $s_{i−1}a_{ij}C_ia_{ij}C_i$ $s_i \mid \exists τ_j ∈ T_r$. $s_{i−1}a_{ij}C_i$ $s_i$ is the $i^{th}$ transition in $τ_j$.

Informally, an element of $δ$ is a transition from a trace in $T_r$, augmented with a pair of indices denoting the trace to which the transition belongs and the position of the transition in the trace.

Note that for any trace in $T_r$, there is a path in the labeled transition system $Q_{T_r}$. Moreover, any path in $Q_{T_r}$ represents a trace that could be obtained by combining trace fragments from $T_r$. We can check if a path from $Q_{T_r}$ belongs to $T_k$ by analyzing the labels of the path as follows. Two consecutive transitions with labels $(j_1, i_1)$ and $(j_2, i_2)$ in a path constitute a switch if either $j_1 ≠ j_2$ or $i_1 + 1 ≠ i_2$. A path in $Q_{T_r}$ belongs to $T_k$ if the number of switches in the path is less than $k$. The algorithm to construct $T_k$ enumerates the paths in $Q_{T_r}$ using depth-first search and discards a path as soon as the number of switches along the path reaches $k$. The algorithm terminates since $k$ is finite and the number of trace fragments is bounded.

4 IMPLEMENTATION

We implemented our test reduction algorithm in a prototype tool called DETREduce. DETREduce works for Android apps, but could be implemented for other platforms supporting graphical user interfaces. The tool is publicly available at https://github.com/wtchoi/swifthand2.
We observed that the following information are useful and often sufficient to characterize a screen:

- Which GUI components are actionable? For example, Checkbox and TextButton components are actionable in Android, while TextBox and DecorationBar components are not.
- Which attribute values of actionable GUI components are visible on the screen? For example, for a screen with a checkbox, one could observe if the checkbox is checked or not.

We extract this information from a raw GUI component tree. The abstraction is computed by first collecting a set of actionable GUI components (i.e., the components with event handlers) from a GUI component tree. Each collected component is augmented with the access path to the component from the root in the GUI tree. Unnecessary attributes are then removed from each component.

5 EVALUATION

We aim to answer the following research questions.

- RQ1: How effective is DetReduce in reducing test suites?
- RQ2: Does DetReduce run in a reasonable amount of time?
- RQ3: Does DetReduce preserve test coverage?
- RQ4: Does DetReduce preserve fault-detection capability?
- RQ5: How many re-executions are required to demonstrate the replayability of a trace?
- RQ6: What will happen to the splicing algorithm if the number of fragments in traces is increased beyond three?

To answer RQ1 to RQ5, we generated test suites using two test generation algorithms (SWIFTHAND [5] and RANDOM [5]) on eighteen benchmark apps and applied DetReduce to reduce them. To answer RQ6, we analyzed the relationship between the likelihood of finding a replayable trace and the bound on the number of fragments in traces using four relatively complicated apps. We used five smartphones (Motorola XT1565) to run benchmark apps.

5.1 Experimental Setup

5.1.1 Benchmark Apps. We applied DetReduce to eighteen free apps downloaded from the Google Play store [11] and F-Droid [39]. Table 1 lists these apps along with their package name, the type of app, and the number of branches in the app (which offers a rough estimate of the size of the app.) Since the apps were downloaded directly from app stores, we have access to only their bytecode. Thirteen apps were used for experimental evaluation in previous research projects [6, 9, 49]; other apps, which we mark with asterisks, are newly selected. We excluded apps for which SWIFTHAND and RANDOM saturate the test coverage in less than an hour. Note that adding such apps would only improve experiment results because most of traces in test suites for such apps are redundant.

<table>
<thead>
<tr>
<th>app</th>
<th>package name</th>
<th>type</th>
<th>#br</th>
</tr>
</thead>
<tbody>
<tr>
<td>acar</td>
<td>com.zonewalker.acar</td>
<td>car manager</td>
<td>20380</td>
</tr>
<tr>
<td>amemo*</td>
<td>com.kconomy.android</td>
<td>finance</td>
<td>28141</td>
</tr>
<tr>
<td>astrid</td>
<td>org.tasks</td>
<td>task manager</td>
<td>16844</td>
</tr>
<tr>
<td>cnote*</td>
<td>dictapps.notepad.color.note</td>
<td>note</td>
<td>14524</td>
</tr>
<tr>
<td>dimoney*</td>
<td>com.bottleworks.dalymoney</td>
<td>finance</td>
<td>5099</td>
</tr>
<tr>
<td>emobile</td>
<td>org.epistudios.epimobile</td>
<td>fitness tracker</td>
<td>3200</td>
</tr>
<tr>
<td>explore</td>
<td>com.speedsoftware.expplorer</td>
<td>file system</td>
<td>54302</td>
</tr>
<tr>
<td>mileage</td>
<td>com.evancharlton.mileage</td>
<td>car manager</td>
<td>7728</td>
</tr>
<tr>
<td>minote</td>
<td>ip.gr.java_conf.hatalab.mnv</td>
<td>text editor</td>
<td>1959</td>
</tr>
<tr>
<td>money*</td>
<td>com.monefy.app.lite</td>
<td>finance</td>
<td>22615</td>
</tr>
<tr>
<td>sanity</td>
<td>cri.sanity</td>
<td>device manager</td>
<td>4610</td>
</tr>
<tr>
<td>tippyn</td>
<td>net.mandalia.tippynipper</td>
<td>tip calculator</td>
<td>3245</td>
</tr>
<tr>
<td>todo*</td>
<td>com.spell.apps.splendo</td>
<td>task manager</td>
<td>11858</td>
</tr>
<tr>
<td>table*</td>
<td>com.gabrielitter.timetable</td>
<td>scheduler</td>
<td>11858</td>
</tr>
<tr>
<td>vlc*</td>
<td>org.videonav.vlc</td>
<td>media player</td>
<td>14410</td>
</tr>
<tr>
<td>whohas</td>
<td>de.freewarepoint.whosaasystuff</td>
<td>inventory</td>
<td>369</td>
</tr>
<tr>
<td>xmp</td>
<td>org.hellibals.android.xmp</td>
<td>media player</td>
<td>5855</td>
</tr>
</tbody>
</table>

5.1.2 Generating a Replayable Test Suite to be Used for Minimization. To generate test suites to be used as inputs to DetReduce, we first collected execution traces by running an implementation of the SwiftHand [5] and Random [5] algorithms. We ran each for eight hours, then checked whether the generated traces are replayable by re-executing each trace ten times. For each non-replayable trace, we identified a non-empty replayable prefix of the trace and retained the prefix rather than throwing the entire trace away.

An app can generate a non-replayable trace for several reasons: a) the app has external dependency (e.g., it receives messages from the outside world, depends on a timer, or reads and writes to the file system), or b) the app has inherent non-determinism due to the use of a random number generator or multi-threading. We removed dependency on the outside world by resetting the contents of the SD card and the app data every time we restart. Nonetheless, it is impossible to eliminate all sources of non-determinism. Therefore, we replayed each trace generated by the SwiftHand and Random algorithms ten times to remove the non-replayable suffixes of traces. We determined experimentally that eight re-executions is sufficient to detect most of non-replayable traces for the benchmark apps.

5.1.3 Why we did not use Monkey to generate initial test suite? Monkey [12] is a fuzz testing tool for Android apps. Monkey is widely-used to automatically find bugs in real-world Android apps. We initially attempted to use Monkey to generate inputs for DetReduce; however, we found that Monkey is not capable of generating replayable traces. We now describe our experience with Monkey.

Monkey is a simple black box tool that reports only the sequence of actions it used to drive a testing session. To get a trace would require non-trivial modifications to Monkey. Before jumping into this effort, we performed an experiment to determine whether Monkey is even capable of generating replayable traces—if Monkey cannot generate replayable traces, there is no point in the modification.

In this experiment, we used a script to generate traces with partial information from Monkey and checked if those traces could be replayed. The script injects user actions at the rate of m actions per second, collecting branch coverage and screen abstraction after injecting every n actions. The script picks the value of m from the set {1, 2, 5, 10, 20, 100} and value of n from {2, 10, 50, 100, 200}. For each pair of values for m and n, the script runs Monkey until it injects 2000 actions. By combining the sequence of actions reported by Monkey with the collected coverage information, the script can generate traces that have coverage and screen information...
after every \( n \) actions (instead of having the information after every event.) We call such traces \textit{partial traces}.

Using this script, we collected three partial traces for each possible value of \( m \) and \( n \) using the same random seed and compared the partial traces. If the partial traces do not match, this indicates that Monkey cannot generate a replayable trace. We performed the experiment using ten apps with three different random seeds.

The results of this experiment showed that Monkey passes the test for four apps when \( n = 2 \) and \( m = 2 \). For the other six apps, Monkey fails the test even when injecting one action per second. At this speed Monkey becomes useless in practice because its power comes primarily from its ability to inject many actions quickly. It will take too long to generate a sufficiently good test suite using Monkey at this speed. Therefore, we have concluded that using Monkey is not viable for generating initial replayable test suite.

Why is Monkey testing highly non-deterministic? We found that Monkey injects actions asynchronously—that is, Monkey injects an action without checking whether the previously injected action has been fully handled. This allows Monkey to inject an order of magnitude more actions than testing tools that synchronously inject actions, but this also makes Monkey highly non-deterministic. For example, we noticed that if actions are injected while the app is unresponsive, those actions are dropped. Because the period of unresponsiveness varies from execution to execution, the number of dropped actions varies across executions.

5.2 Evaluation of DetReduce

Table 2 and Table 3 show the results of applying DetReduce to the test suites generated by the SwiftHand and Random algorithms, respectively. Each table has four parts. The first part shows the following information about the test suites to be minimized: total branch coverage (\( \#br \)), total screen coverage (\( \#s \)), total number of transitions (\( \#act \)), and total number of traces (\( \#tr \)) of each initial test suite. The second and third parts of the table show information about the test suites generated after running the first and second phases, respectively, of DetReduce. The fourth part shows important statistics summarizing the experiment results: the running time of each phase of the algorithm (\( t_{p1} \) and \( t_{p2} \)), the execution time (\( t_e \)) of the resulting reduced regression test suite, and the ratio of the execution time of the resulting regression suite to the execution time of the original test suite in percentage (\( t_e/t \)). We make the following observations from the data shown in the tables.

- **RQ1**: The execution time of the reduced test suites (\( t_e \)) is several orders of magnitude shorter than that of the original test suites (8 hours). This shows that DetReduce is highly effective in minimizing the test suites for the benchmark apps. Regarding the sizes of test suites, phase 1 of DetReduce removes 91.27% of transitions (median) and 90.5% of restarts (median). Phase 2 of DetReduce further removes 33.07% of transitions and 31.81% of restarts from the test suites obtained after Phase 1. These two phases of DetReduce cumulatively remove 93.84% of transitions and 93.52% of restarts. We also found that the rate of reduction is higher for test suites generated from Random. This is because these test suites have lower test coverage and more redundancies.

- **RQ2**: The running time of the algorithm is within a factor of 6X of the execution time of the original test suites generated by the test generation algorithms. More than half of the running time was spent in detecting and eliminating loops in phase 1 (note that DetReduce spends no time removing redundant traces because such traces do not require any execution). The time spent in phase 1 is reasonable because the phase searches for a minimized test suite while eliminating redundant loops from each trace. Note that these experiments employed a conservative parameter (ten) for the number of re-executions to perform to check trace replayability, and the running cost of DetReduce can be further reduced by setting this parameter to eight.

- **RQ3**: Despite using an approximate method for checking if a trace is replayable, the minimized test suites nonetheless cover the most of the original branch and screen coverage. DetReduce fails to provide 100% coverage for amoney, explorer, ttable, and vlc. We manually analyzed the reasons for the missing branches and screens, and determined that non-replayable traces were not fully removed while generating the original test suites before phase 1 of DetReduce.

- **RQ4**: In order to check how DetReduce affects the fault-detection capability of test suites, we collected exceptions raised while executing each test suite. We identified seven distinct exceptions based on their stacktrace. All survived after applying DetReduce. Note that DetReduce does not consider exceptions to be part of the test coverage it tries to preserve.

- **RQ5**: We measured how many re-executions were required to identify each non-replayable trace created during our experiments. The following table summarizes the results.

| \( n \) |  \( T \) | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 | 64000 | 128000 | 256000 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 19043 | 10956 | 120 | 86 | 35 | 35 | 27 | 14 | 3 | 0 | 0 |

The first column (\( T \)) shows the total number of traces created during the experiments, and the remaining columns show the number of non-replayable traces that required \( n \) re-executions for detection. The results show that all non-replayable traces were detected within the first eight iterations. The results also show that 37.5% of traces attempted during the experiments were replayable traces, suggesting that DetReduce is good at selecting candidate traces in our benchmarks.

5.3 Splicing and the Number of Fragments in Traces

**RQ6**: To understand the effect of bounding the number of trace fragments in phase 2 of our algorithm, we measured the relationship between the bound and the likelihood of finding a replayable trace, and the average number of trace fragments in a trace generated by splicing. For these measurements, we used four relatively complex benchmark apps.

5.3.1 Bounding the Number of Fragments and the Replayability of Traces. We measured the correlation between the bound on the number of fragments and the possibility of finding a replayable trace, and the average number of trace fragments in a trace generated by splicing. For these measurements, we used four relatively complex benchmark apps.
would decrease the possibility of finding replayable traces, and the

Table 3: Test reduction result using DetReduce on SwiftHand traces

<table>
<thead>
<tr>
<th>app</th>
<th>initial test suite</th>
<th>phase 1</th>
<th>phase 2</th>
<th>running time in hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>acar</td>
<td>4270 173 1508 1483</td>
<td>3242</td>
<td>1116.8</td>
<td>0.90</td>
</tr>
<tr>
<td>ameno</td>
<td>2955 114 1304 1350</td>
<td>2955</td>
<td>114.08</td>
<td>1.13</td>
</tr>
<tr>
<td>amoney</td>
<td>4481 159 1321 779</td>
<td>4481</td>
<td>159.28</td>
<td>1.14</td>
</tr>
<tr>
<td>astrid</td>
<td>6201 170 10532 560</td>
<td>6201</td>
<td>170.18</td>
<td>1.14</td>
</tr>
<tr>
<td>amone</td>
<td>5098 102 1084 1515</td>
<td>5098</td>
<td>102.15</td>
<td>1.14</td>
</tr>
<tr>
<td>drome</td>
<td>2387 47 13511 785</td>
<td>2387</td>
<td>47.72</td>
<td>1.14</td>
</tr>
<tr>
<td>emobile</td>
<td>1554 213 12651 782</td>
<td>1554</td>
<td>213.14</td>
<td>1.14</td>
</tr>
<tr>
<td>explore</td>
<td>6647 105 7559 703</td>
<td>6647</td>
<td>105.14</td>
<td>1.14</td>
</tr>
<tr>
<td>mone</td>
<td>889 76 13697 1003</td>
<td>889</td>
<td>76.98</td>
<td>1.14</td>
</tr>
<tr>
<td>money</td>
<td>4966 62 13703 806</td>
<td>4966</td>
<td>62.01</td>
<td>1.14</td>
</tr>
<tr>
<td>santy</td>
<td>978 186 12375 764</td>
<td>978</td>
<td>186.14</td>
<td>1.14</td>
</tr>
<tr>
<td>todo</td>
<td>2651 126 13028 1516</td>
<td>2651</td>
<td>126.12</td>
<td>1.14</td>
</tr>
<tr>
<td>vlc</td>
<td>2341 60 11978 770</td>
<td>2341</td>
<td>60.79</td>
<td>1.14</td>
</tr>
<tr>
<td>whos</td>
<td>230 15 12857 727</td>
<td>230</td>
<td>15.19</td>
<td>1.14</td>
</tr>
<tr>
<td>xmp</td>
<td>2079 50 11326 761</td>
<td>2079</td>
<td>50.17</td>
<td>1.14</td>
</tr>
</tbody>
</table>

The results obtained using DetReduce demonstrate that phase 2 of DetReduce is more effective in reducing the number of traces without affecting the coverage. Our hypothesis was that increasing the number of fragments would decrease the possibility of finding replayable traces, and the results confirm this hypothesis for the four apps.

Table 4: The frequency of replayable traces.

<table>
<thead>
<tr>
<th>app</th>
<th>#replayable traces (out of 2000 samples per app)</th>
<th>#replayable traces (out of 1000 samples per app)</th>
</tr>
</thead>
<tbody>
<tr>
<td>acar</td>
<td>159 119 96 94 80 70 47 54 47</td>
<td>25 14 22 37 24 42 48 73 689</td>
</tr>
<tr>
<td>astrid</td>
<td>189 108 93 64 51 41 22 16 19 14</td>
<td>29 3 15 16 10 13 23 38 70 785</td>
</tr>
<tr>
<td>cnote</td>
<td>186 117 77 73 53 31 22 31 16 11</td>
<td>12 4 11 2 7 8 10 6 13 929</td>
</tr>
<tr>
<td>emobile</td>
<td>199 169 139 116 96 93 62 41 50 57</td>
<td>25 16 20 23 24 59 33 59 88 685</td>
</tr>
</tbody>
</table>

constructed via splicing. We then sampled 200 traces from the set of 20,000 traces. Finally, we checked how many of the sampled traces are replayable by executing each trace ten times. Table 4 shows the results. The first column shows the name of the app and the rest of the columns show the number of replayable traces for each k. Our hypothesis was that increasing the number of fragments would decrease the possibility of finding replayable traces, and the results confirm this hypothesis for the four apps.

5.3.2 Number of Fragments in Traces Generated by Splicing

Even if traces containing many trace fragments tend to be non-replayable, we would not need to bound the number of fragments during phase 2 of DetReduce if most of the traces that can be

Table 5: The frequency of traces composed of k fragments.

<table>
<thead>
<tr>
<th>app</th>
<th>#traces composed of k fragments (out of 1000 samples per app)</th>
</tr>
</thead>
<tbody>
<tr>
<td>acar</td>
<td>25 14 22 37 24 42 48 73 689</td>
</tr>
<tr>
<td>astrid</td>
<td>29 3 15 16 10 13 23 38 70 785</td>
</tr>
<tr>
<td>cnote</td>
<td>12 4 11 2 7 8 10 6 13 929</td>
</tr>
<tr>
<td>emobile</td>
<td>25 16 20 23 24 59 33 59 88 685</td>
</tr>
</tbody>
</table>

constructed from Qr contain a small number of fragments. Our hypothesis was that, without a proper bound, a significant number of traces generated from Qr would contain many fragments, making them non-replayable with high probability. Therefore, phase 2 of DetReduce would spend considerable amount of time checking the replayability of non-replayable traces. In order to validate our hypothesis, we constructed 1000 traces composed of most at random by sampling random paths from Qr, and checked the number of traces that are replayable in each sampled trace. Table 5 shows the results. The first column shows the name of the app and the rest of the columns show the number of traces composed of k fragments for each k between 1 and 10. The results show that there are many more traces composed of a large number of fragments than traces composed of fewer fragments. Consequently, if we perform splicing without bounding the number of fragments, we are more likely to get traces composed of a large number of fragments. The results of the previous experiment (Section 5.3.1) suggest that such traces are likely to be non-replayable. This validates our hypothesis that phase 2 of DetReduce will not scale if the number of trace fragments is unbounded.

5.4 Threats to Validity

We used a limited number of benchmark apps to evaluate DetReduce, so it is possible that our results do not generalize. To address this issue, we carefully selected the benchmark apps, and the details of the selection process are explained in Section 5.1.1.

The selection of the test generation algorithms could potentially bias the evaluation results. Specifically, the results obtained from a single algorithm cannot determine whether the results can be generalized to the other test generation algorithms. To address this issue, we used both SwiftHand and Random algorithms. We could not use Monkey because it cannot generate replayable traces, as explained in Section 5.1.3. The results obtained using Random show
that DetReduce is not an artifact that only works with SwiftHand. However, the results are still not strong enough to decisively conclude that DetReduce can effectively reduce test suites generated from an arbitrary test generation tool.

Finally, in the evaluation, we checked whether the exceptions raised by the original test suites are also raised by the test suites reduced by DetReduce. However, this is a limited evaluation. A more robust evaluation would involve injecting artificial faults into the benchmark apps and measuring the number of injected faults detected before and after the test suite reduction. We did not take this approach because of the difficulty of injecting faults into the binary of an Android app.

6 RELATED WORK

GUI test minimization. ND3MIN [7] is the most closely related work to our research. It is a GUI test minimization algorithm for Android apps, based on delta-debugging [48]. We compare ND3MIN and DetReduce in three aspects: a) They have different goals: ND3MIN aims to minimize each test case in isolation while keeping the final activity. DetReduce tries to minimize a whole test suite while keeping the branch and screen coverage of the test suite. b) Handling non-determinism: ND3MIN aims to tolerate non-deterministic behaviors occurring during the execution of an app. On the contrary, DetReduce is designed to actively detect and avoid non-deterministic behaviors during the process of minimization. c) Running time: ND3MIN is a variation of delta-debugging, whose worst case time complexity is O(n^2) where n is the size of input test case [48]. This could make the algorithm fail to scale since the cost of performing each test run is expensive in GUI testing. ND3MIN uses up to 50 hours to minimize a test case composed of 500 actions. DetReduce is capable of handling a test suite composed of more than 10,000 transitions in less than 30 hours. We could not perform an empirical comparison because the implementation of ND3MIN was not available to us.

Hammoudi et al. [17] also proposed a delta-debugging based test minimization algorithm. Unlike ND3MIN and DetReduce, their work aims to minimize manually written test cases for web applications. Their results showed that the execution time of the minimized test cases are on average 22% shorter than that of the original test cases. This shows that there is room to minimize even manually written test cases. Since they used relatively small test cases composed of less than 150 actions, it is hard to say if their delta-debugging approach would scale on a large GUI test case.

Test minimization in general. Test suite reduction techniques [18, 19, 21, 22, 24, 30, 41, 42, 46, 50] automatically reduce the size of a test suite without losing the coverage of the test suite. Unlike our work, these techniques assume that test suites consist of already compacted test cases; these techniques do not focus on reducing the size of each test case. They only focus on selecting a small set of test cases from a test suite. The first part of the first phase of DetReduce, where we remove redundant traces, can be seen as a test suite reduction technique. In the context of GUI testing, McMaster and Memon [30] proposed call-stack history as a metric for reducing GUI test suites. We might be able to reduce more redundant traces by adopting this technique. However, it is possible that removing too many traces at the first phase of DetReduce might negatively affect the capability of the second phase of DetReduce.

Delta-debugging [47, 48] is probably the most widely-known test minimization technique. We found it difficult to use delta-debugging to minimize a large GUI test suite because of the cost of running the test suite. It is often possible to accelerate delta-debugging by exploiting domain specific knowledge. For example, hierarchical delta-debugging (HDD) [35] works on structured texts, such as XML, by first performing delta-debugging on top-level structures, then gradually moving into substructures. This allows HDD to significantly reduce the time required to reduce structured texts compared to the original delta-debugging. A similar idea has been used in DEMi [40] to minimize test cases for a distributed system. However, we have yet to find a way to make delta-debugging scale better on GUI test suites.

Prior to our work, Groce et al. [15] proposed cause-reduction, a test reduction technique combining delta-debugging and greedy test-case selection. There are two notable differences between cause-reduction and DetReduce. First, cause-reduction is comparable to phase 1 of DetReduce, and it does not have a component corresponding to phase 2. Second, cause-reduction uses delta-debugging as a component. On the contrary, DetReduce uses highly domain-specific components such as loop-elimination and splicing.

Automated Android GUI testing techniques. In this paper we used SwiftHand [5] to generate test suites. One can use any automated GUI testing technique, such as A3E [2], Dynodroid [26], AppsPlayground [38], or Mobiguitar [1], to generate initial test suites. One may argue that test minimization might not be necessary in the future if automated testing techniques continue to improve. Automated GUI testing techniques are indeed becoming better in maximizing test coverage and finding bugs in a limited period of time [8, 28]. However, recent studies [17, 46] suggest that even test cases and test suites created by human experts need to be compacted. Therefore, we predict that GUI test suite minimization techniques will remain useful, even though automated GUI testing techniques continue to improve.

A recent survey [6] compares the performance of several automated testing tools for Android apps. Their results suggest that Monkey outperforms other more sophisticated tools in terms of maximizing coverage in a limited period of time. However, we observed that it is difficult to replay test cases generated by Monkey. Even if Monkey finds a bug, it might be difficult to reproduce the bug or minimize the sequence of actions obtained from Monkey [43].

GUI test scripts [13, 14, 39] and record-and-replay tools [10, 14, 16, 20, 36] are means to generate reusable test cases reflecting human knowledge. These tools complement our approach. One can use these tools either to generate a set of test cases to be minimized, or to add more test cases to already minimized test suites.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants CCF-1409872 and CCF-1423645.

REFERENCES

DetReduce: Minimizing Android GUI Test Suites for Regression Testing

ICSE ’18, May 27–June 3, 2018, Gothenburg, Sweden


