Minimizing GUI Event Traces

Lazaro Clapp
lazaro@stanford.edu

Osbert Bastani
obastani@cs.stanford.edu

Saswat Anand
saswat@cs.stanford.edu

Alex Aiken
aiken@cs.stanford.edu

Stanford University
Stanford CA, USA

ABSTRACT

GUI input generation tools for Android apps, such as Android's Monkey [13], are useful for automatically producing test inputs, but these tests are generally orders of magnitude larger than necessary, making them difficult for humans to understand. We present a technique for minimizing the output of such tools. Our technique accounts for the non-deterministic behavior of mobile apps, producing small event traces that reach a desired activity with high probability.

We propose a variant of delta debugging [36, 38], augmented to handle non-determinism, to solve the problem of trace minimization. We evaluate our algorithm on two sets of commercial and open-source Android applications, showing that we can minimize large event traces reaching a particular application activity, producing traces that are, on average, less than 2% the size of the original traces.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

testing; trace minimization; delta debugging; Android

1. INTRODUCTION

Test cases are time consuming to write, especially for applications dealing with rich graphical user interfaces (GUIs). Many properties can only be reliably tested once the program has reached a particular state, such as a specific screen or view in its GUI. Part of the challenge of GUI testing is in creating a sequence of user interactions that cause the program to reliably reach a target GUI state, under which the test one cares about can be performed. Automatically generating a sequence of GUI events to reach a particular point in the program is a difficult problem to solve in general.

In many cases, it is possible to reach the desired point in an application by randomly generating GUI events until the right view is displayed, or by capturing and replaying the GUI events generated by a user, or a human tester, interacting with the application. However, these randomly generated or tester-captured *event traces* generally contain more interactions than necessary to reach the desired state. Furthermore, traces generated by capture and replay of concrete user interactions might not be robust in the face of application non-determinism, and thus might break if the program changes behavior, even slightly, between executions. In modern mobile and web apps, which often include internal A/B testing logic and personalization, GUI non-determinism is a common obstacle for simple capture and replay systems.

In this paper, we:

- Present an algorithm based on delta-debugging [36, 38] to minimize a trace of GUI events. Our algorithm is robust to application non-determinism. When the application is deterministic, the output trace is 1-minimal, meaning no element of the trace can be removed without changing the behavior we care about [38].

- Minimization proceeds by checking whether subtraces of a trace still reach the desired state. This problem is highly parallelizable, but it is not obvious how to take maximal advantage of the information gained from each subtrace trial (see Section 4). We define the subtrace selection problem and provide an efficient heuristic, as well as an optimal solution based on the problem's characterization as a Markov Decision Process (MDP).

- We show the effectiveness of the above techniques in minimizing randomly generated traces for two representative datasets: one set of popular Android apps taken from the Google Play Store, and a set of open-source apps from the F-droid application repository.

Section 2 provides a brief overview of the problem of GUI trace minimization in the face of application non-determinism, the relevant characteristics of the Android platform, and a motivating example for our technique. Section 3 describes our trace minimization algorithm while Section 4 explores the problem of subtrace selection embedded in the algorithm's inner loop, including the characterization as a Markov Decision Process (4.3). Section 5 presents our results, contrasting the performance of different solutions to the subtrace selection problem. Finally, Section 6 briefly describes related work and Section 7 presents our conclusions.

FSE '16 November 13-19, 2016, Seattle, WA, USA

DOI: 10.1145/1235

2. PROBLEM OVERVIEW

The GUI of an Android application (also called an *app*), consists of a set of *activities*. Each activity corresponds to a top-level view (e.g., a page) of the user interface. Additionally, each app has one *main activity*, which is the first one invoked when the app is launched. In the course of interacting with an application's GUI, the user commonly transitions between different activities, as actions in one activity trigger another activity.

A GUI *event trace* is a sequence of user interface events such as screen taps, physical keyboard key presses and complete multi-touch gestures. One way to obtain such a trace is to record the actions of a user. Another option uses a random or programmatic input generation tool to generate the trace without human involvement. In our experiments, we took the second approach, using Monkey [13], a standard first-party random input generation tool for Android.

Monkey traces can be generated based only on the number and types of desired events, then replayed for a particular app. By generating multiple large random traces for each app under test, we are able to reach various activities within the app. These traces could conceivably serve as system tests for the app, especially if augmented by state checks or log monitoring. However, using automatically generated traces as tests can be problematic for the following reasons:

1. Because traces are randomly generated, the majority of the events do not trigger any interesting app behavior (e.g. they are taps on inactive GUI elements), or trigger behavior unrelated to the functionality we care about for a given test. Large random event traces are hard for humans to interpret, particularly if most of the trace is irrelevant for the desired behavior. The traces produced by Monkey typically consist mostly of irrelevant events.

2. Replaying a large trace is a time consuming process, as a delay must be introduced before sending each event to the app, to ensure previous events have been fully processed. In our experiments, we set this delay to 4 seconds, which we found necessary to allow for events that trigger network requests or other expensive operations to be fully handled by the app before the next event is triggered.

In summary, given a large event trace, we would like to find a minimal subtrace that triggers the same app behavior. In particular, we focus on subtraces that reach the same activity. If we assume that the app is deterministic in the sense that the same event trace always visits the same activities in the same order, then the algorithm from Section 3 extracts, from a given event trace that reaches a particular activity, a 1-minimal subtrace that reaches that activity.

Note that we use activity reachability as a proxy for uncovering user triggered behavior in the app. All the techniques and checks in this paper apply just as well if the behavior we seek to trigger involves reaching the execution point of a particular GUI widget callback or the point at which a particular web request is sent. We only require that we have a small finite set of behaviors that shall be triggered in the app, so that every system test is built on top of a minimal trace triggering that behavior.

2.1 Trace minimization example

One of the apps we use for our experiments (see Section 5) is Yelp's Eat24 (`com.eat24.app.apk`), a popular commercial food delivery app. To generate minimized traces that reach this app's activities, we run the app on an Android emulator and use Monkey to generate multiple GUI event traces. Each trace consists of 500 single-touch events at random coordinates within the app's portion of the screen. To capture non-determinism in the app, we replay each trace multiple times (we use 20 repetitions), clearing any local data or app state in the device between replays.

For a particular trace T we generated, the activity `LoginActivity` is always reached by replaying T on this app. `LoginActivity` is a relatively easy to reach activity for this particular app, as there are at least two ways to launch this activity immediately from the app's main activity: either by clicking on an item in the application's menu or on the lower portion of the app's home screen (which displays recommendations if the user is already logged in). The second method requires only a single GUI event: the click on the bottom of the screen. However, approximately 50% of the time when the app is launched from a clean install, it shows a dialog asking for the user's address and zip code. This dialog blocks interaction with the main activity and, in our set-up, automatically brings up Android's software keyboard. Dismissing the dialog is as simple as clicking anywhere in the screen outside the dialog and the virtual keyboard. However, this does not dismiss the keyboard itself, so the state of the screen is different than if the dialog had never appeared. After dismissing the dialog, it is still possible to navigate to the `LoginActivity` with one more click, but the area of the screen that must be clicked is different than if the dialog had never appeared at all.

Suppose now we wanted to manually select, out of the 500 events T, a minimal subtrace T' such that replaying the events of T' in order reaches the `LoginActivity` activity regardless of whether the app shows the location dialog or not. This subtrace must exist, since the original trace always reaches `LoginActivity` (among other activities), in both cases. However, we cannot simply select the single click subtrace that would launch `LoginActivity` if the dialog is not present, since it won't work when the dialog does appear. We have the same problem if we focus on picking the two clicks when the dialog appears, as such a trace does not necessarily work when the dialog is absent. For example, most clicks that would simply dismiss the dialog might also trigger different behavior if the dialog is missing by clicking on active GUI widgets underneath the dialog.

This sort of behavior is not exclusive to the EAT24 app. In fact, many Android apps behave non-deterministically, either because of internal A/B testing logic or just because of the non-determinism of the app's external environment. We could always manually write GUI testing scripts that are aware of the app's specific non-determinism and generate different GUI events when faced with different app states. However, as we will show, it is possible to automatically generate small subtraces that are robust against application non-determinism, while still treating the app itself as a black box. We require only a way to run subtraces on top of the app from an initial state multiple times and list the activities being reached.

Figure 1 shows the execution of a 3 event trace – the minimal subtrace obtained by the technique in this paper

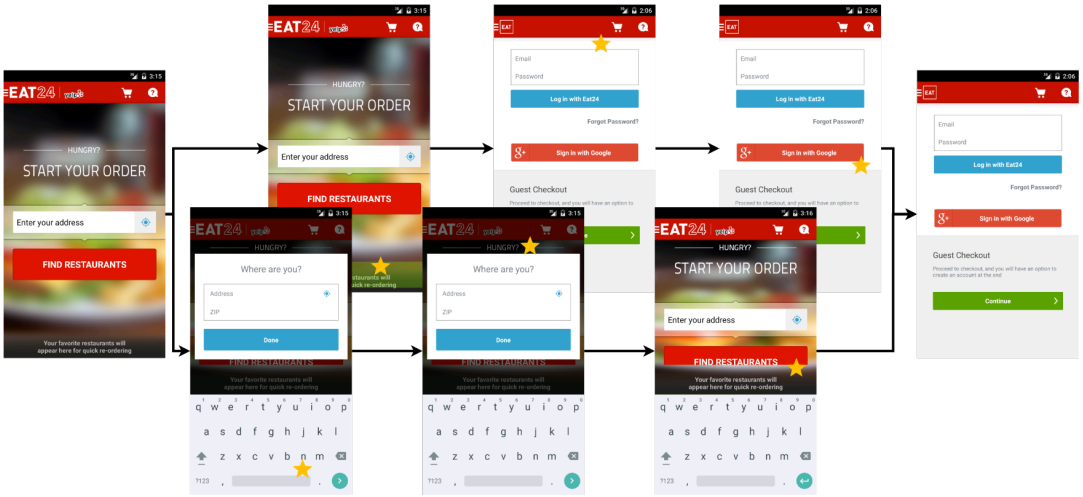


Figure 1: Reaching `LoginActivity` on `com.eat24.app.apk`

– that accomplishes our goal without checking at any point the state of the application's GUI. If there is no location dialog, the first event in the trace triggers the direct transition to `LoginActivity` by clicking on the bottom of the screen. The second and third events click inactive areas of the `LoginActivity` GUI layout, having no effect in this case. If the dialog appears, the first click hits a portion of the virtual keyboard layout, typing a whitespace character into the location dialog. The second click immediately dismisses the dialog without using the whitespace character. Finally, the third click happens in the new location of the panel that launches `LoginActivity`, without dismissing the keyboard[1]. Thus, whether or not the dialog appears, the script will reach `LoginActivity` and stop there. It is worth noting that at no point is our technique aware of the dialog; it only knows that this script reaches `LoginActivity` with high probability over multiple runs of the app.

3. MINIMIZATION ALGORITHM

In this section we present our trace minimization algorithm and discuss its basic properties. Our algorithm is based on Zeller's delta debugging algorithm (see [38]), reformulated for our problem and augmented to deal with application non-determinism.

Intuitively, delta debugging starts with a large input sequence that passes a particular test oracle and attempts to remove part of the sequence at every step, such that the remaining subtrace still passes the oracle. This is repeated until we have a 1-minimal subtrace that is accepted by the oracle. A *1-minimal* subtrace with property P satisfies P, but removing any single element does not satisfy P.

Our version of delta debugging takes as input an Android app A, a trace T (which is an ordered sequence $T = \{e_0, e_1, ...\}$ of GUI events) and a target activity a within A. A subtrace T' of T is a subsequence of T. The algorithm has access to a test oracle $O(A, T, a)$, which consists of starting a new Android emulator, installing A, running trace T on A and verifying that a was launched during that execution. The oracle returns either 0 (the target activity a was not reached) or 1 (activity a was reached). Because of application non-determinism, our test oracle may accept

[1]Transitioning between activities does dismiss the keyboard, so we don't need to do so explicitly.

```
   globals : O, n, nr, st
1  def ND3MIN(A,T,a):
2      return MinR(A,T,a,n);
3  def MinR(A,T,a,k):
4      size ← len(T)/k;
5      for i in range(0,k):
6          T_i ← T[i * size : (i+1) * size]; T̄_i ← T \ T_i;
7      T_sub ← get_passing(A,{∀i ∈ [0,k). T_i},a);
8      if T_sub ≠ None:
9          return MinR(A,T_sub,a,n);
10     T_compl ← get_passing(A,{∀i ∈ [0,k). T̄_i},a);
11     if T_compl ≠ None:
12         return MinR(A,T_compl,a,max(k − 1,2));
13     if k < len(T):
14         return MinR(A,T,a,min(2k,len(T)));
15     return T;
16 def get_passing(A,S,a):
17     if ∃T_c ∈ S. passes(A,T_c,a):
18         return T_c;
19     return None;
20 def passes(A,T,a):
21     s ← 0;
22     for i in range(0,nr):
23         s ← s + O(A,T,a);
24     return s ≥ st;
```

Figure 2: `ND3MIN`: Non-Deterministic Delta Debugging MINimization.

a trace with some probability p, and reject it with probability $1 - p$. Fixing A and a, we define the probability $P_T = Pr[O(A, T, a) = 1]$, which is the trace's underlying probability of reaching a when run on app A.

Figure 2 shows the pseudo-code for the general form of our trace minimization algorithm (`ND3MIN`). Besides A, T and a, the algorithm uses 4 global parameters: the oracle O, a starting number n of subtrace candidates to test, a number of times to run each candidate (nr) and the success threshold required to accept it (st).

For the algorithm to select a particular subtrace T' at the end of any minimization step, calling $O(A, T', a)$ nr times results in the oracle returning 1 at least st times. This requirement is enforced by function `passes()` in line 20. We say that a trace is *successful* if it passes the check in `passes()`. Function `get_passing()` in line 16 takes a set S of subtraces and selects a subtrace $T_c \in S$ such that `passes(`A,T_c,a`)` returns true. It returns None iff no such subtrace exists in

set S. Note that `get_passing()` specifies no order in which traces are passed to `passes()`. We assume oracle calls are expensive but we have the ability to make multiple oracle calls in parallel. In Section 4, we use the flexibility in the definition of `get_passing()` to minimize the number of rounds of (parallel) calls to O required by our algorithm.

`ND3MIN()` calls the recursive function `MinR()` which uses the two helper functions described above to implement our version of delta debugging. `MinR()` follows the classic structure of delta debugging. First (lines 4–6), it partitions trace T in k subtraces T_i of contiguous events of roughly equal size (starting with $k = n$ on the first recursive call). It also generates the complement of each subtrace T_i, defined as $\overline{T_i} \leftarrow T \setminus T_i$. It then proceeds in four cases:

Case #1: If any candidate subtrace T_i is successful, the algorithm selects that T_i as its new current trace T, and calls itself recursively with $k = n$ (lines 7–9).

Case #2: Otherwise, if any complement subtrace $\overline{T_i}$ is successful, the algorithm selects that $\overline{T_i}$ to be the next T. If $k \geq 3$, $k = k - 1$ before the next recursive call, otherwise it is set to 2 (lines 10–12). Reducing k by 1 ensures that the candidate subtraces T_i generated in the next call will be a subset of those in this call, so the algorithm will keep reducing to complement subtraces until $k = 2$ or no $\overline{T_i}$ is successful. Note that when $k = 2$, both the set of candidate subtraces and that of complements are identical ($\overline{T_0} = T_1$ and $\overline{T_1} = T_0$).

Case #3: If k is smaller than the number of events left in T, we double the number of partitions, up to $k = \text{len}(T)$ (lines 13–14). Note that when we reach $k = \text{len}(T)$, this implies that on the next recursive call, every subtrace consists of a single event.

Case #4: Otherwise, return T as our minimized subtrace

Ideally, we would like to show that, given a probability (lower) bound p_b, if $P_T \geq p_b$ and $T_{min} = \texttt{ND3MIN}(A,T,a)$ then, with high probability, T_{min} is a 1-minimal subtrace of T such that $P_{T_{min}} \geq p_b$. Unfortunately, this is not possible. To see why, imagine $\exists T' \subsetneq T_{min}$, $P_{T'} = p_b - \delta$. As $\delta \to 0$, the number of checks required to distinguish T' from a subtrace which fulfills $P_{T'} \geq p_b$ grows without bound. If T' can be obtained from T_{min} by removing a single event, then testing T_{min} for 1-minimality must take unbounded time. We consider instead the following property:

Definition 1. *A trace T is* approximately 1-minimal *with respect to the bound p_b and distance ϵ, if $P_T \geq p_b - \epsilon$, and any subtrace T' which can be obtained by removing a single event from T is such that $P_{T'} < p_b$.*

We would like to bound the probability that `ND3MIN` returns an approximately 1-minimal trace with bound $p_b = st/nr$ and distance ϵ. We would first like to show that if $P_T \geq p_b$ and $T_{min} = \texttt{ND3MIN}(A,T,a)$, then, with high probability, $P_{T_{min}} \geq p_b - \epsilon$ for a small ϵ. Given a single independent subtrace T', if the check $X = \texttt{passes}(A,T',a)$ returns true, then the probability that $P_{T'} \geq p_x$, for a given p_x, is:

$$\tilde{p} = Pr[P_{T'} \geq p_x \mid X] \tag{1}$$

$$= \int_0^1 Pr[P_{T'} \geq p_x \mid X, P_{T'} = p] \cdot f_{P_{T'}|X}(p)dp \tag{2}$$

$$= \int_{p_x}^1 f_{P_{T'}|X}(p)dp \tag{3}$$

where $f_{P_{T'}|X}(p)$ is the probability density of $P_{T'}$ given X. Note that $Pr[P_{T'} \geq p_x \mid P_{T'} = p]$ is 1 if $p \geq p_x$, and 0 otherwise. By Bayes' theorem:

$$\tilde{p} = \frac{\int_{p_x}^1 Pr[X \mid P_{T'} = p] \cdot f_{P_{T'}}(p)dp}{Pr[X]} \tag{4}$$

$$= \frac{\int_{p_x}^1 Pr[X \mid P_{T'} = p] \cdot f_{P_{T'}}(p)dp}{\int_0^1 Pr[X \mid P_{T'} = p] \cdot f_{P_{T'}}(p)dp} \tag{5}$$

By definition of $X = \texttt{passes}(A,T',a)$:

$$Pr[X \mid P_{T'} = p] = \sum_{i=st}^{nr} \binom{nr}{i} \cdot p^i (1-p)^{nr-i} \tag{6}$$

Note that if we assume a discrete probability distribution, we can replace the integrals above by sums over p, and the density functions $f_{P_{T'}}(p)$ by probabilities $Pr[P_{T'} = p]$. Using this approximation and plugging the parameters $nr = 20$ and $st = 18$, as well as the (discrete) prior probability distribution $Pr[P_{T'} = p]$ obtained experimentally (see Section 5), we have $Pr[P_{T'} \geq 0.85 \mid \texttt{passes}(A,T',a)] > 0.95$. So, selecting $p_b = 0.9$ and $\epsilon = 0.05$ would at first seem like an option to prove a bound on the probability of $P_{T_{min}} \geq p_b - \epsilon$. Unfortunately, most executions of our algorithm perform a large number of calls to `passes()`, and the error accumulates rapidly. After just 20 calls, the naive bound on $Pr[P_{T_{min}} \geq 0.85]$ in our example would become $0.95^{20} \approx 0.36$. Bounding the error in our algorithm more tightly is non-trivial. Instead, when running our experiments, we perform a final independent check, calling $\texttt{passes}(A,T_{min},a)$ one last time on the final output of our algorithm. In Section 5 we observe that this final check passes often. For the minimized traces where this final check succeeds, we can indeed say that $P_{T_{min}} \geq 0.85$ with probability > 0.95, as per the example above, which uses our experimental parameters.

We can now get a bound on the probability of the second requirement in the definition of approximate 1-minimality:

Lemma 1. *If $T_{min} = \texttt{ND3MIN}(A,T,a)$, then the probability $\widehat{p}$ that there exists no subtrace T', obtained by removing a single event from T_{min}, such that $P_{T'} \geq p_b$ is:*

$$\widehat{p} = \left(1 - \frac{\int_{p_b}^1 \left(1 - \sum_{i=\mathrm{st}}^{\mathrm{nr}} \binom{\mathrm{nr}}{i} \cdot p^i (1-p)^{\mathrm{nr}-i}\right) \cdot f_{P_{T'}}(p)dp}{\int_0^1 \left(1 - \sum_{i=\mathrm{st}}^{\mathrm{nr}} \binom{\mathrm{nr}}{i} \cdot p^i (1-p)^{\mathrm{nr}-i}\right) \cdot f_{P_{T'}}(p)dp}\right)^l$$

where l is the number of events in T_{min}.

Proof. Consider only the last call to the recursive procedure `MinR()`, which returns T_{min}. Note that `MinR()` returns T_{min} only in Case #4 which executes only when cases #1 to #3 are not satisfied. Thus, for the last execution of `MinR()`, Case #3 must have been skipped, which means

$k \geq \text{len}(T_{min})$. Since $k \geq \text{len}(T_{min})$, the set $\{\forall i \in [0,k).\, \overline{T_i}\}$ contains every subtrace which can be obtained by removing a single event from T_{min}. Because Case #2 was also not satisfied, we know that calling `get_passing()` on this set at line 10 returns **None**. By definition, this is equivalent to calling `passes()` on each $\overline{T_i}$ and having it return false.

Taking $Y_i = \neg$`passes(`A,$\overline{T_i}$,a`)`, we have:

$$Pr[Y_i \mid P_{\overline{T_i}} = p] = 1 - \sum_{i=st}^{nr} \binom{nr}{i} \cdot p^i (1-p)^{nr-i} \quad (7)$$

And, using steps analogous to equations 1 to 5 above:

$$\widehat{p_i} = Pr[P_{\overline{T_i}} \geq p_b \mid Y_i] \quad (8)$$

$$= \frac{\int_{p_b}^{1} Pr[Y_i \mid P_{\overline{T_i}} = p] \cdot f_{P_{\overline{T_i}}}(p) dp}{\int_{0}^{1} Pr[Y_i \mid P_{\overline{T_i}} = p] \cdot f_{P_{\overline{T_i}}}(p) dp} \quad (9)$$

which is the probability $Pr[P_{\overline{T_i}} \geq p_b]$ for each $\overline{T_i}$ given the behavior of `passes()` the algorithm must have observed.

Finally, the probability that no $\overline{T_i}$ is such that $P_{\overline{T_i}} \geq p_b$ is given by $\widehat{p} = \prod_{\overline{T_i}} (1 - \widehat{p_i})$ which expands to the formula for $\widehat{p}$ given in the lemma's statement, since the formula for $\widehat{p_i}$ is the same for every T_i, and there are $l = k = \text{len}(T_{min})$ such subtraces. □

4. TRACE SELECTION

Recall that the method `get_passing()` takes a set S of subtraces of T, and must return a subtrace $T_c \in S$ such that calling the oracle $O(A, T_c, a)$ nr times would succeed st times. If no such $T_c \in S$ exists, `get_passing()` must be able to determine that and return **None**. Calls to oracle O are time consuming, so we wish to minimize the number of such calls that execute non-concurrently. We assume that we have a maximum of m instances of O which can be run in parallel. In our implementation, these represent individual instances of the Android emulator.

For each call to `get_passing()`, we have a set of n traces $S = \{T_0, ..., T_{n-1}\}$. We define a schedule as an array $sch = [v_0, ..., v_{n-1}]$ such that $\sum_{j=0}^{n-1} v_j \leq m$. A step for `get_passing()` consists on generating a new schedule sch, running v_j oracle calls $O(A, T_j, a)$ for each $j \in [0, n-1]$ and capturing the results. Since the total number of calls to O is at most m, the calls corresponding to a single step of `get_passing()` can be executed in parallel. We accumulate the results of all steps before the current one as pairs (s_j, f_j), where s_j is the number of successes seen so far for T_j (i.e. 1 was returned by $O(A, T_j, a)$) and f_j the number of failures (0 was returned). We can see that given the definition of `get_passing()`, we never gain anything from running a single T_j more than nr times, so we forbid this, and thus $\forall j.\, s_j + f_j \leq nr$. We seek to minimize the number of steps in each call to `get_passing()` before we can either identify a T_j which satisfies `passes()` (i.e. $\exists j.\, s_j \geq st$) or we have concluded that no such T_j can exist. We note that, given the previous constraints, if ever $f_j > nr - st$, T_j cannot be a trace that satisfies `passes()`.

We give 3 strategies for minimizing the number of steps of `get_passing()`. Section 5 compares them empirically.

4.1 Naive Scheduling

There are two obvious ways to generate the schedule sch at every step of `get_passing()`, which depend very little on the observed pairs (s_j, f_j).

The first method is to schedule all nr executions for each trace one after the other, so at every step $v_j = min(nr - s_j - f_j, m - \sum_{k=0}^{k<j} v_k)$. This strategy goes from $j = 0$ to $n-1$ and greedily tries to add another execution of T_j to sch until doing so would either mean that more than nr executions of T_j have been scheduled over all steps of `get_passing()` or would push the schedule beyond the limit of m calls to O. A common sense optimization is, at every step, to return immediately if a T_j with $s_j \geq st$ has been found, and ignore any T_j with $f_j > nr - st$ for scheduling. The worst-case for this strategy happens when no trace in S satisfies `passes()`. The following is a particular example of this greedy strategy in action with $nr = 20$, $st = 18$, $n = 3$ and $m = 15$. We represent each step as a transition between two lists of pairs (s_j, f_j) $\forall j = 1, 2, 3$, representing the accumulated results before and after the step. Each step is also annotated with the corresponding schedule sch:

$$[(0,0),(0,0),(0,0)] \xrightarrow[15,0,0]{} [(13,2),(0,0),(0,0)]$$
$$[(13,2),(0,0),(0,0)] \xrightarrow[5,10,0]{} [(17,3),(3,7),(0,0)]$$
$$[(17,3),(3,7),(0,0)] \xrightarrow[0,0,15]{} [(17,3),(3,7),(15,0)]$$
$$[(17,3),(3,7),(15,0)] \xrightarrow[0,0,5]{} [(17,3),(3,7),(19,1)]$$

Another naive strategy is to schedule traces in a round–robin fashion. Each step scans $j = 0$ to $n-1$ multiple times, adding an additional invocation of T_j if $s_j + f_j + v_j \leq nr$ and $f_j \leq nr - st$. It stops when the schedule is full (m calls scheduled) and proceeds to run sch. Again, we stop as soon as there is nothing else to run or we have found a T_j with $s_j \geq st$. The worst-case for this strategy happens when all traces succeed with high probability. We repeat the example above with the round–robin strategy:

$$[(0,0),(0,0),(0,0)] \xrightarrow[5,5,5]{} [(4,1),(2,3),(5,0)]$$
$$[(4,1),(3,2),(5,0)] \xrightarrow[8,0,7]{} [(11,2),(2,3),(12,0)]$$
$$[(11,2),(2,3),(12,0)] \xrightarrow[7,0,8]{} [(17,3),(2,3),(19,1)]$$

Both of these naive strategies are similar in that they are likely to do plenty of unnecessary work in the average case. We chose the round–robin variant as our baseline for comparison, since it performs better in the case in which is common for many of the traces in S to fail the oracle often and st is close to nr (this matches our scenario).

4.2 Heuristic: Exploration followed by greedy

The naive algorithms don't exploit all of the information contained in the pairs (s_j, f_j) in deciding what to do next. Clearly, if we have a trace T_{j_1} for which $(s_{j_1}, f_{j_1}) = (5, 0)$, and a trace T_{j_2} for which $(s_{j_2}, f_{j_2}) = (3, 2)$, then T_{j_1} is significantly more promising than T_{j_2}, and we should try checking it first. Conversely, it should be possible to discard T_{j_2} by scheduling it for execution only a few more times; adding 15 copies of T_{j_2} to the next schedule, while allowed, would likely be a waste. We use these observations to propose a heuristic that, at every step: a) tries to confirm traces that seem likely to be successful, and b) tries to discard traces that seem likely to be unsuccessful, in that order of priority.

Before scheduling the first step, we have $\forall j.\, (s_j, f_j) = (0, 0)$. Since we have no information, we simply schedule the traces in S in a round–robin fashion. This gives us at least some information about every T_j. In every round after that, we follow the algorithm outlined in Figure 3.

```
   globals : c
1  def schedule(S,[(s_j, f_j)]):
2      ∀j. v_j ← 0;
3      ∀j. p_j ← s_j/(s_j + f_j);
4      sort S by p_j, s_j desc;
5      S_c ← [T_j ∈ S | p_j ≥ c];
6      S_d ← ∅;
7      S_f ← [T_j ∈ S | p_j < c];
8      while Σ_j v_j < m:
9          if S_c ≠ ∅:
10             T_k ← remove_first(S_c);
11             x_k ← min(nr − s_k − f_k, ⌈(st − s_k)/p_k⌉);
12             if x_k ≤ m − Σ_j v_j:
13                 v_k ← x_k;
14             elif x_k ≤ m:
15                 S_d ← S_d ∪ [T_k];
16             else:
17                 v_k ← m − Σ_j v_j;
18         elif S_d ≠ ∅:
19             Schedule from S_d by round–robin.;
20         elif S_f ≠ ∅:
21             T_k ← remove_first(S_f);
22             y_k ← min(nr − s_k − f_k, ⌈(nr−st+1)−f_k / (1−p_k)⌉);
23             v_k → min(y_k, m − Σ v_j);
24         else:
25             Schedule from original S by round–robin.;
```

Figure 3: Trace selection heuristic.

We first compute $p_j = s_j/(s_j + f_j)$ for each j, the empirical success probability of T_j so far. We sort S in descending order, first by p_j and then by s_j. Then we partition the sorted array S into two sections: S_c contains the traces such that $p_j \geq c$ for a certain constant c ($c = 0.8$ in our implementation) and S_f the rest. We assume that traces in S_c are likely to succeed after nr calls to O, while traces in S_f are likely to fail, and we predict accordingly. While there are traces in S_c, and our schedule is not full, we remove T_k from S_c in order. We compute x_k in line 11, which is the expected number of runs of T_k needed to get to the point where $s_k = st$. If we can schedule that many runs, we do so (line 13). If we can't schedule x_k runs of T_k in this step, but we can do it in the next step, we move T_k to S_d, a list of 'deferred' traces from S_c. We do this so that if we can pack the expected number of runs for multiple traces in S_c we do so, even if those aren't the traces with the highest p_j. If x_k is too large to schedule in any single step, then we just schedule as many copies of T_k as we can.

Only when S_c is empty, we revisit the traces in S_d and schedule them in round–robin fashion. If there are no traces in S_d then we begin removing T_k from S_f in order. We compute y_k in line 22 for these traces, which is the expected number of runs of T_k needed to get to the point where $f_k = nr - st + 1$ (at which time we can declare that trace as failing `passes()`). We could run the traces in S_d in order of increasing empirical success probability, which would allow us to discard some of them more quickly, but this doesn't reduce the expected number of steps for `get_passing()`, since we need to discard all traces before we can return **None**. We run them in order of the decreasing probability instead, as this will allow us to more quickly correct course in the rare case in which we have misclassified a passing trace as being in S_d: after running a few more copies of the trace, instead of discarding it, we would observe its empirical probability increasing, and we reclassify it into S_c on the next step.

If there is space left in the schedule after scheduling S_c, S_d and S_f as described, we add additional runs of the traces in S in a round–robin fashion, that is, copies beyond the expected number of executions required to 'prove' or 'disprove' each T_j, but without running any T_j more than nr times.

We show the execution of our heuristic on our same example from the previous two techniques:

$$[(0,0),(0,0),(0,0)] \xrightarrow[5,5,5]{} [(4,1),(2,3),(5,0)]$$

$$[(4,1),(2,3),(5,0)] \xrightarrow[2,0,13]{} [(6,1),(2,3),(18,0)]$$

4.3 Solving trace selection as an MDP

We can formulate the problem of trace selection as a Markov Decision Process (MDP), which allows us to solve for the optimal strategy, for given values of the parameters n, m, nr, st.

A Markov Decision Process is a tuple $(\mathbb{S}, \mathbb{A}, P, \gamma, R)$, where: $\mathbb{S}$ is a set of states, $\mathbb{A}$ is a set of actions and $P : \mathbb{S} \times \mathbb{A} \to \{Pr[\mathbb{S}]\}$ is a map from every state and action pair to a probability distribution over possible state transitions. $R : \mathbb{S} \to \mathbb{R}$ is the reward function, which associates a value with reaching each state. Finally, $\gamma \in [0, 1]$ is called the discount factor.

To execute an MDP, we start from some initial state $\sigma_0 \in \mathbb{S}$, then choose an action $a_0 \in \mathbb{A}$. The MDP then transitions to a state σ_1 chosen at random over the probability distribution $P(\sigma_0, a_0)$. The process is then repeated, selecting a new a_i for each σ_i and choosing σ_{i+1} from the distribution $P(\sigma_i, a_i)$. The value of the execution of the MDP is then $\sum_i \gamma^i R(\sigma_i)$, which is the reward of each state visited, multiplied by γ^i. The discount factor γ is used to decrease the reward of reaching "good" states, in proportion to how late in the execution these states are reached.

Given a policy $\pi : \mathbb{S} \to \mathbb{A}$, which is simply a mapping from states to actions, we calculate the value of the policy as the expected value $V^\pi(\sigma_0) = E[\sum_i \gamma^i R(\sigma_i)]$ where σ_0 is the initial state, and for every $i \geq 0$, $a_i = \pi[\sigma_i]$ and σ_{i+1} is chosen from $P(\sigma_i, a_i)$. Solving an MDP is equivalent to finding a policy π that maximizes $V^\pi(\sigma_0)$.

We encode trace selection as an MDP as follows:

- $\mathbb{S} = \{[(s_0, f_0), ..., (s_{n-1}, f_{n-1})]\}$ is the set of possible combinations of observed values of (s_j, f_j) for each $T_j \in S$. The initial state is $\sigma_0 = [(0,0), ..., (0,0)]$
- $\mathbb{A} = \{[v_0, ..., v_{n-1}] \mid \sum_{j=0}^{n-1} v_j \leq m\}$ is the set of possible schedules *sch*.
- $P(\sigma_i, a)[\sigma_j]$ where $\sigma_i = [(s_{i0}, f_{i0}), ...]$, $a = [v_0, ...]$ and $\sigma_j = [(s_{j0}, f_{j0}), ...]$ with $\forall k.\, (s_{jk} + f_{jk}) - (s_{ik} + f_{ik}) = v_k$ is:

$$\prod_{k=0}^{n-1} \sum_{p} \binom{v_k}{s_\delta} p^{s_\delta} (1-p)^{(v_k - s_\delta)} Pr[P_{T_k} = p]$$

  with $s_\delta = s_{jk} - s_{ik}$. $P(\sigma_i, a)[\sigma_j]$ is 0 if $\exists k.\, (s_{jk} + f_{jk}) - (s_{ik} + f_{ik}) \neq v_k$.
- $R(\sigma')$ is -1 for every state σ', and $\gamma = 1$

We make the reward negative and the discount factor 1, since we wish to find only a policy that minimizes the number of reached states, which is equivalent to minimizing the number of steps in `get_passing()`. Any state containing (s_j, f_j) with $s_j \geq st$ for any j, as well as any state where $f_j > nr - st$ for every j, is a terminal state of the MDP: once such a state is reached, the execution of the MDP ends.

Note too that the precise definition of $P(\sigma_i, a)[\sigma_j]$ requires knowledge of $Pr[P_{T_k} = p]$ for each subtrace T_k. But we have no way of precisely knowing this distribution. In practice, if we have a prior (discrete) probability distribution $Pr[P_{T'} = p]$ over the set of all possible subtraces, we can approximate:

$$Pr[P_{T_k} = p] = \sum_p \left(\binom{s_k + f_k}{s_k} p^{s_k} (1-p)^{f_k} \right) Pr[P_{T'} = p]$$

In Section 5, we approximate the prior by running `ND3MIN()` with naive round–robin trace selection on a few (app, trace, activity) tuples. We use that experimentally discovered prior to approximate $P(\sigma_i, a)[\sigma_j]$.

There are methods for solving an MDP in the general case, but they require iterating multiple times over the set $\mathbb{S}$ and calculating the expected value $V^\pi(\sigma')$ of each state based on the value of other states, until a fix-point is reached. In our formulation, the number of states, enumerated naively, is:

$$|\mathbb{S}| = \left(\sum_{k=0}^{k \leq nr} (k+1) \right)^n = \left(\frac{(nr+1)(nr+2)}{2} \right)^n$$

since we can construct $k+1$ pairs (s_j, f_j) with $s_j + f_j = k$. For $n = 5$ ($nr = 20$), this gives us over 6×10^{11} states.

We can significantly optimize our solution for this particular MDP in two ways, however: by getting rid of the fix-point iteration requirement (changing it to a single pass over $\mathbb{S}$), and by reducing the size of $\mathbb{S}$ itself.

First, we observe that in our MDP we can never visit the same state twice. In fact, our MDP is a DAG, where each state must transition to one in which the sum of the elements of the tuples (s_j, f_j) has a higher value. This type of MDP is called a finite horizon MDP [17].

We now restrict all candidate policies π to include only schedules for which $\sum_{j=0}^{n-1} v_j = m$ exactly, except in the case in which doing so would violate the constraint $\forall j.\, s_j + f_j \leq nr$. In the latter case, every π always chooses to schedule as many runs of each T_j as needed to reach nr. We note that this last action must always lead to a final state.

Every state $\sigma' = [(s_0, f_0), ..., (s_{n-1}, f_{n-1})]$ reachable from the initial state σ following any π with the above restrictions, is either a final state or is such that $\sum_{j=0}^{n-1}(s_j + f_j) \mod m = 0$. We then define I, the state's iteration, such that $I[\sigma'] = \left(\sum_{j=0}^{n-1}(s_j + f_j)\right)/m$ for every non-final state. $I[\sigma']$ is always an integer. We assign all final states to iteration $I[\sigma'] = \lceil \frac{nr \times n}{m} \rceil$. We note that for every $\sigma_i \in \mathbb{S}$, π as above, and $\sigma_j \in P(\sigma_i, \pi[\sigma_i])$ we have $I[\sigma_j] > I[\sigma_i]$.

Given the partition of $\mathbb{S}$ into subsets $\mathbb{S}_i = \{\sigma \in \mathbb{S} \mid I[\sigma] = i\}$ induced by I, we can solve the MDP by visiting each $\mathbb{S}_i$ once in reverse order of i and computing the optimal $\pi[\sigma']$ and $V^\pi(\sigma')$ for each $\sigma' \in \mathbb{S}_i$. We do not need to iterate to reach a fix-point, since

$$V^\pi(\sigma') = R(\sigma') + max_{a \in \mathbb{A}} \sum_{\sigma''} \left(P(\sigma', \pi[\sigma'])[\sigma''] \right) V^\pi(\sigma'')$$

depends only on the values $V^\pi(\sigma'')$ of states reachable from σ' by actions in π. Since all such states satisfy $I[\sigma''] > I[\sigma']$, their value has already been calculated when visiting a previous $\mathbb{S}_{j>i}$. The time to solve the MDP by this method is $O(|\mathbb{A}|\,|\mathbb{S}|)$. Note that within a single $\mathbb{S}_i$, the problem of computing $V^\pi(\sigma')$ for the states in $\mathbb{S}_i$ is highly parallelizable.

We can further reduce $|\mathbb{S}|$ by merging states which are isomorphic in our model. First, we can coalesce all final states into two: a success state σ_s whenever $s_j \geq st$ for any j, and a failure state σ_f when $f_j > nr - st$ for every j. Furthermore, if state σ' has $f_j > nr - st$ for any j, this is equivalent (for the purposes of `get_passing()`) to the same state after replacing (s_j, f_j) with (X, Y) with $Y > nr - st$. We pick a single state in this equivalence set as representative, taking care to choose one which preserves the invariant $\sum_{j=0}^{n-1} s_j + f_j \mod m = 0$ and falls into the latest possible iteration. We can also reorder the tuples (s_j, f_j) in every state, since the order of the subtraces in S does not affect the result of `get_passing()`, and the generated policy π will be identical, up to a reordering of $a' = \pi[\sigma']$. After applying all optimizations, for $n = 5$, $m = 15$, $nr = 20$ and $st = 18$, our algorithm must examine $|\mathbb{S}'| = 729,709$ states in order to compute the optimal strategy π.

After pre-computing π for a particular set of parameters n, m, nr, st, we can use it to generate a schedule sch at each step of the trace selection process. Section 5 compares the number of steps required by `get_passing()` when using this strategy versus our heuristic from Section 4.2.

5. RESULTS

This section presents our empirical evaluation and results. Section 5.1 describes our experimental setup and presents the data discussed in the rest of the section. Section 5.2 explores the size of the minimized subtraces and the number of calls to `get_passing()` performed by the algorithm. Section 5.3 contrasts the performance of the different trace selection methods of Section 4. Finally, Section 5.4 explores the prevalence of application non-determinism in our dataset.

Our tests were performed in an Amazon EC2 cloud environment, in which we ran $m = 15$ Android emulator instances, each on its own virtual machine, as our test oracles. As mentioned before, the rest of the parameters used for `ND3MIN()` are $n = 5$, $nr = 20$ and $st = 18$.

5.1 Datasets

We evaluate our event trace minimization approach on two different datasets: one composed of 7 applications from the Google Play store selected among the top 20 most popular apps in their category across different app categories (**gplay**), and another of 4 redistributable open-source applications from the F-droid open-source application repository (**fdroid**). For each application, we generated 10 random Monkey traces, of 500 events each, restricting ourselves to tap events exclusively. Although our approach can potentially be used to minimize traces with any type of events, reaching activities requiring events other than taps, such as structured keyboard input or complex touch gestures, often produces an unfeasibly large original trace when performing random testing alone.

We ran these traces on the corresponding application and recorded the activities reached during their execution. For the Google Play apps, we arbitrarily selected an average of 4 such activities per app, with the minimum being 3 activities. For the F-Droid dataset, we selected an average of 5.25 activities per app, with a minimum of 3. For each app A, and activity a, we took a trace T in our 10 generated traces which reached a with $P_T \geq 0.9$ (observed over 20 trace replays). In the cases where many traces reached a with equal probability, we picked one at random.

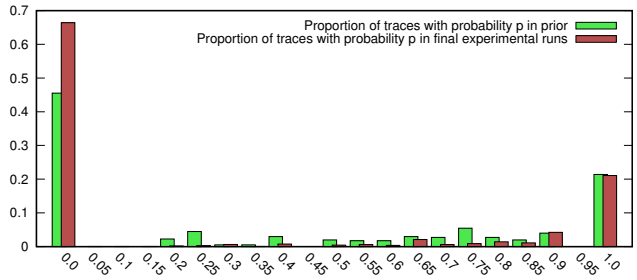


Figure 4: Prior probability distribution of subtraces.

We first ran `ND3MIN()` on nine activities from two apps (`com.eat24.app` and `com.duolingo`) of the **gplay** set, using the naive (round–robin) trace selection strategy of Section 4.1. This produced calls to our test oracle O with 402 distinct subtraces in total. We used the oracle responses obtained from this preliminary experiment to generate a probability prior $Pr[P_{T'} = p]$ given an unknown T' and a probability p. We restricted ourselves to only two apps and nine activities for generating this prior for two reasons. First, minimizing traces under the naive strategy is a time consuming process, compared with either the heuristic or MDP methods. Second, we want to show that the prior computed for a few apps generalizes over unrelated Android applications, meaning that computing this prior is a one time cost, leading to a solution for the MDP model that can be applied without changes to minimizing traces for unknown apps.

Figure 4 shows this prior probability distribution. As observed in Section 4.3, we can use this prior to approximate the transition probabilities used in the MDP for computing the optimal policy for trace selection. To check the quality of this prior, we also plot the probability distribution as estimated by examining all queries to the test oracle performed by the rest of the experiments in this section (from a total of 3490 subtraces). This distribution shows more traces as always failing ($P_{T'} = 0$) and fewer traces as having low but non-zero probability. Otherwise, the distribution looks very similar to our prior, which increases our confidence in using said prior as our estimate of $Pr[P_{T'} = p]$.

For each dataset (**gplay** and **fdroid**) we then ran `ND3MIN()` on each tuple (A, T, a) in the set, under two different configurations for trace selection (see Section 4):

- One using the heuristic in Section 4.2 exclusively for every invocation of method `get_passing()`.
- One using the pre-computed optimal policy (Section 4.3) when `get_passing()` receives a set S with 2 to 5 subtraces (the values of n for which we are able to compute the optimal policy in reasonable time[2]). In this configuration, `get_passing()` defaults to using the same heuristic of the previous configuration, whenever `get_passing()` is passed $n \geq 6$ subtraces.

On average 66% of the steps executed in the MDP based trace selection case are steps in which `get_passing()` was invoked with fewer than 6 subtraces, and thus uses the optimal policy, based on solving the MDP. The remaining 34% fall back to using the same heuristic of Section 4.2.

[2]Solving the MDP using 4 2-vCPU EC2 VMs takes 8 min for $n = 3$, 43 min for $n = 4$ and 83 hours for $n = 5$.

For the applications in the Google Play dataset, Table 1 shows the size of the minimal subtrace obtained by our minimization algorithm for each target activity, together with the number of steps and wall-clock time that our algorithm took to extract it in each configuration. Table 2 shows the analogous information for the F-Droid open-source apps. The performance of the naive method on the 2 apps of the **gplay** set, used to compute the $P_{T'}$ priors, is listed in Table 3.

5.2 Size of minimized traces, steps and check

The column labeled `events` in our tables specifies the number of events in the minimized trace T_{min} as generated by our algorithm. Recall that our input traces are 500 events long in each experiment. The average length of the minimized traces is 4.57 for the **gplay** dataset using the heuristic trace selection, 4.18 using the MDP-based version. For the **fdroid** dataset, these numbers are 3.05 and 2.9, respectively. The fact that only a few events are needed in each case to reach the desired activity shows the value of minimization. Our minimized traces are smaller than the original Monkey traces by an average factor of roughly 100x.

The column labeled `steps` counts the number of times the method `get_passing()` generated a schedule and called our 15 test oracles in parallel. Equivalently, `steps` is the maximum number of sequential calls to each of the test oracles required during our minimization algorithm. To make sure the trace is suitable for minimization, our implementation first runs the original trace T 20 times, and aborts running if the oracle doesn't accept T in at least 15 of those calls. We include the two steps required for this check in our count.

The column labeled `check` contains a triplet of the form $c\ (p_1/p_2)$. After our minimization algorithm has finished, we run the resulting trace 20 additional times, and record as c the number of times it succeeds. We use this number to calculate $p_1 = Pr[P_{T_{min}} \geq 0.85]$ and p_2, the probability that, if $P_{T_{min}} \geq 0.85$, then T_{min} is approximately 1-minimal, as defined by Lemma 1. For p_1, we use a formula analogous to that of Section 3, but taking into account the exact number of successful oracle queries:

$$p_1 = \frac{\sum_{p \geq 0.85} \binom{nr}{c} \cdot p^c (1-p)^{nr-c} Pr[P_{T'} = p]}{\sum_p \binom{nr}{c} \cdot p^c (1-p)^{nr-c} Pr[P_{T'} = p]}$$

For the probability prior required for these calculations, we use the same prior from Figure 4 used by the MDP trace selection method. All probabilities in the tables are truncated, not rounded, as we wish to obtain a lower bound. As observed in Section 3, since the error of `passes()` accumulates through multiple trace reductions in our algorithm, our final $P_{T_{min}}$ can fall below p_b, so it is not always true that $c \geq 18$. The vast majority of our minimized traces fullfil $P_{T_{min}} \geq 0.9$, and all but one succeed over 50% of the time.

Note that for the same (app, trace, activity) tuple, our algorithm sometimes produces minimized traces of different sizes when using different trace selection strategies. This can happen for one of two reasons. First, different trace selection strategies cause delta debugging to pick different subtraces during recursive invocations of the `MinR()` method, which can guide the algorithm towards discovering different 1-minimal solutions. A 1-minimal solution does not imply the returned trace is of minimum length among all possible successful subtraces, and an input trace can contain multiple distinct 1-minimal subtraces that reach the desired activity

Table 1: Results for the Google Play apps

			heuristic				opt:mdp			
Application	Key	Activity	events	steps	time	check	events	steps	time	check
com.eat24.app	1-1	SplashActivity	0	4	4:34:01	20 (0.99/1.0)	0	4	4:17:47	20 (0.99/1.0)
	1-2	HomeActivity	0	4	3:45:04	20 (0.99/1.0)	0	4	5:48:18	20 (0.99/1.0)
	1-3	LoginActivity	3	18	13:27:12	20 (0.99/0.94)	3	17	12:22:16	20 (0.99/0.94)
	1-4	CreateAccountActivity	5	51	35:17:52	19 (0.94/0.91)	5	45	32:33:16	19 (0.94/0.91)
com.duolingo	2-1	LoginActivity	0	4	3:34:23	20 (0.99/1.0)	0	4	4:19:35	20 (0.99/1.0)
	2-2	HomeActivity	2	21	11:11:34	20 (0.99/0.96)	2	15	12:51:19	20 (0.99/0.96)
	2-3	WelcomeFlowActivity	2	18	12:06:37	20 (0.99/0.96)	2	16	13:11:39	20 (0.99/0.96)
	2-4	SkillActivity	19	73	40:46:06	18 (0.69/0.72)	16	62	51:01:40	12 (0.00/0.76)
	2-5	LessonActivity	25	87	50:16:46	11 (0.00/0.65)	23	110	87:56:06	11 (0.00/0.67)
	2-6	FacebookActivity	2	64	52:14:13	20 (0.99/0.96)	7	85	48:02:17	20 (0.99/0.88)
com.etsy.android	3-1	HomescreenTabsActivity	0	4	3:18:15	20 (0.99/1.0)	0	4	3:27:18	20 (0.99/1.0)
	3-2	CoreActivity	3	28	21:27:39	20 (0.99/0.95)	3	71	43:27:56	20 (0.99/0.95)
	3-3	DetailedImageActivity	8	50	38:55:17	15 (0.11/0.87)	7	35	26:31:25	7 (0.00/0.88)
com.ted.android	4-1	SplashScreenActivity	0	4	5:27:51	20 (0.99/1.0)	0	4	4:11:50	20 (0.99/1.0)
	4-2	MainActivity	1	13	8:28:01	20 (0.99/0.98)	1	10	7:04:49	20 (0.99/0.98)
	4-3	TalkDetailActivity	7	31	17:45:40	19 (0.94/0.88)	7	24	15:59:37	20 (0.99/0.88)
	4-4	VideoActivity	15	57	39:02:54	19 (0.94/0.77)	11	32	22:27:27	18 (0.69/0.82)
	4-5	BucketListInfoActivity	5	13	13:45:54	20 (0.99/0.91)	5	11	10:26:14	19 (0.94/0.91)
com.zhiliaoapp.musically	5-1	SignInActivity	2	16	13:52:34	20 (0.99/0.96)	2	16	15:04:38	20 (0.99/0.96)
	5-2	OAuthActivity	1	13	14:32:24	20 (0.99/0.98)	1	14	13:55:28	20 (0.99/0.98)
	5-3	TermOfUsActivity	1	12	12:53:04	20 (0.99/0.98)	1	13	12:44:12	20 (0.99/0.98)
com.pandora.android	6-1	SignUpActivity	1	18	21:12:14	20 (0.99/0.98)	1	14	16:38:10	20 (0.99/0.98)
	6-2	SignInActivity	1	13	18:32:25	20 (0.99/0.98)	1	14	15:48:11	20 (0.99/0.98)
	6-3	ForgotPasswordActivity	8	72	61:41:23	18 (0.69/0.87)	3	49	47:00:46	17 (0.43/0.94)
com.google.android .apps.photos	7-1	LicenseActivity	6	46	51:34:31	20 (0.99/0.90)	5	47	40:44:01	18 (0.69/0.91)
	7-2	LicenseMenuActivity	5	50	51:35:34	19 (0.94/0.91)	5	46	39:18:00	18 (0.69/0.91)
	7-3	SettingsActivity	3	27	34:18:45	20 (0.99/0.94)	2	36	31:02:26	20 (0.99/0.96)
	7-4	PhotosAboutSettingsActivity	3	34	32:21:50	20 (0.99/0.94)	4	34	28:26:12	20 (0.99/0.93)
Average			4.57	30.18	24:34:17		4.18	29.86	23:48:40	
Median			2.5	19.5	18:09:03		2.5	16.5	15:53:54	

Table 2: Results for the F-Droid apps

			heuristic				opt:mdp			
Application	Key	Activity	events	steps	time	check	events	steps	time	check
com.evancharlton .mileage	8-1	Mileage	0	4	3:09:23	20 (0.99/1.0)	0	4	3:06:46	20 (0.99/1.0)
	8-2	VehicleStatisticsActivity	2	17	8:35:44	20 (0.99/0.96)	2	15	8:48:25	20 (0.99/0.96)
	8-3	TotalCostChart	3	25	12:46:05	20 (0.99/0.95)	3	25	13:16:33	20 (0.99/0.95)
	8-4	FillupInfoActivity	10	84	41:21:14	20 (0.99/0.84)	10	85	36:39:12	20 (0.99/0.84)
	8-5	FillupActivity	0	4	3:12:27	20 (0.99/1.0)	0	4	3:31:56	20 (0.99/1.0)
	8-6	FillupListActivity	2	16	9:09:12	19 (0.94/0.96)	2	21	10:58:12	19 (0.94/0.96)
	8-7	MinimumDistanceChart	3	15	7:46:58	20 (0.99/0.95)	3	14	11:07:33	20 (0.99/0.95)
	8-8	AverageFuelEconomyChart	3	14	7:39:26	20 (0.99/0.95)	3	14	8:09:53	20 (0.99/0.95)
de.delusions.measure	9-1	MeasureTabs	0	4	3:35:59	20 (0.99/1.0)	0	4	3:23:09	20 (0.99/1.0)
	9-2	MeasureActivity	0	4	3:12:24	20 (0.99/1.0)	0	4	2:56:53	20 (0.99/1.0)
	9-3	BmiTableActivity	2	17	8:52:07	20 (0.99/0.96)	2	26	11:32:35	20 (0.99/0.96)
	9-4	BmiCalcActivity	4	48	22:47:29	20 (0.99/0.93)	3	27	13:01:08	20 (0.99/0.93)
org.liberty.android .fantastischmemo	10-1	AnyMemo	0	4	3:20:17	20 (0.99/1.0)	0	4	3:12:32	20 (0.99/1.0)
	10-2	OptionScreen	3	16	8:18:16	20 (0.99/0.95)	3	14	8:06:41	20 (0.99/0.95)
	10-3	AlgorithmCustomizationScreen	5	61	36:57:50	20 (0.99/0.91)	5	51	30:44:08	20 (0.99/0.91)
	10-4	StudyActivity	2	17	10:29:14	20 (0.99/0.96)	2	17	9:40:59	20 (0.99/0.96)
	10-5	CardEditor	6	49	24:30:42	18 (0.69/0.90)	6	35	16:52:17	20 (0.99/0.90)
	10-6	SpreadsheetListScreen	3	29	19:02:10	20 (0.99/0.95)	3	40	24:33:01	20 (0.99/0.95)
org.totschnig .myexpenses	11-1	CalculatorInput	0	4	3:24:06	20 (0.99/1.0)	0	4	3:19:23	20 (0.99/1.0)
	11-2	MyExpenses	3	52	26:12:57	20 (0.99/0.95)	3	43	21:40:33	20 (0.99/0.95)
	11-3	ExpenseEdit	13	70	32:22:06	16 (0.23/0.80)	11	66	36:05:57	17 (0.43/0.82)
Average			3.05	26.38	14:07:55		2.9	24.61	13:22:16	
Median			3	17	8:52:07		3	17	10:58:12	

and have different lengths. Because of the probabilistic nature of our algorithm, it is also possible that the trace returned by `ND3MIN()` is not truly 1-minimal, especially if it contains a subtrace which reaches the activity with a probability very close to 0.9, which our algorithm might have trouble classifying either way.

We can observe this situation when contrasting the minimized traces discovered by the naive and heuristic trace selection methods for `CreateAccountActivity` in `com.eat24.app`. The heuristic method produces a 5 event minimized trace that passes 19 out of 20 oracle calls in its final check, while the naive method returns a 4 event subtrace, which is actually a subtrace of the one returned by the heuristic case, but which only passes 17/20 checks. We re-ran both traces 300 times, which suggests the underlaying probability of the 5 event trace is ≈ 0.89 and that of the 4 event trace is ≈ 0.84.

5.3 Performance comparison

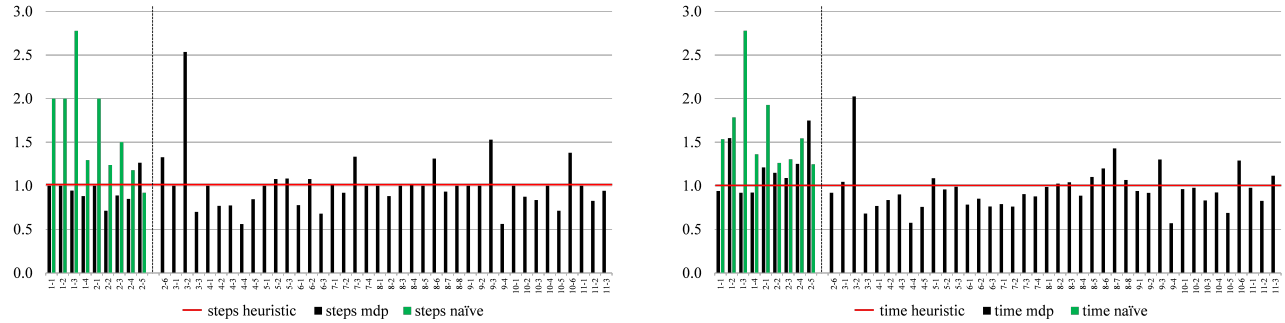
Figure 5 plots the number of steps and wall-clock time for each experiment, comparing the naive, heuristic and MDP-based trace selection methods. We normalize in each experiment to the value obtained in the heuristic case, since we only have the performance of the naive method for the limited set of (app, trace, activity) tuples in Table 3. Thus the red line at 1 represents the performance of our heuristic, and the bars represent the performance of the naive and MDP methods relative to that of the heuristic. The dashed vertical line separates the experiments for which we have data on the naive method from those for which we don't.

We note that the configuration using the MDP policies doesn't always outperform our heuristic. This is not unreasonable, since: a) the MDP method only guarantees to minimize the number of steps in `get_passing()`, but it might pick different subtraces than other methods, thus failing to minimize the number of steps over the whole algorithm, b) even within a call to `get_passing()` we are approximating the prior of $P_{T'}$ based on previous experiments, which could lead to non-optimal results if the distribution of underlying trace probabilities is very different in the particular app under test, versus the set used to compute the prior, and c) since the oracle is non-deterministic, the number of steps our algorithm must perform, even when using the same trace selection method, varies across runs. Our results do indicate, however, that using the heuristic method for trace selection is a reasonable option, which performs similarly to solving the MDP formulation and avoids the expensive pre-computation for every value of n.

Two outliers in our plots bear explaining. In experiment 2-5, both our heuristic and the MDP based method seem to under-perform the naive method on the number of steps. However, this is also a case in which the naive method produces a larger trace (30 events) than either the heuristic (25

Table 3: Performance for naive trace selection

			naive			
Application	Key	Activity	# events	# steps	exec. time	check
com.eat24.app	1-1	SplashActivity	0	8	7:00:20	20 (0.99/1.0)
	1-2	HomeActivity	0	8	6:41:37	20 (0.99/1.0)
	1-3	LoginActivity	3	50	37:22:56	20 (0.99/0.95)
	1-4	CreateAccountActivity	4	66	48:01:59	17 (0.43/0.93)
com.duolingo	2-1	LoginActivity	0	8	6:53:19	20 (0.99/1.0)
	2-2	HomeActivity	2	26	14:07:33	20 (0.99/0.96)
	2-3	WelcomeFlowActivity	2	27	15:47:21	18 (0.69/0.96)
	2-4	SkillActivity	15	86	62:55:47	15 (0.11/0.77)
	2-5	LessonActivity	30	80	62:41:16	14 (0.04/0.59)
Average			6.2	39.9	29:03:34	
Median			2	27	15:47:21	



(a) Proportion of mdp and naive steps versus heuristic **(b) Times of mdp and naive versus heuristic**

Figure 5: Trace selection performance comparison

events) or the MDP based method (23 events), so this outcome can be explained as the result of the naive method having stopped the minimization process earlier than the other two. In experiment 3-2 the MDP based method performs much worse than our heuristic while producing a trace of identical size. In this case, the heuristic found a trace consisting of 3 consecutive events of the original trace, while the MDP based method found a trace of non-consecutive events. The structure of delta debugging is such that it generally makes faster progress towards a contiguous subsequence than towards a non-contiguous one.

The average running time of our minimization algorithm using the heuristic approach is 24:34 hours for the activities in the **gplay** set (median: 18:09 hours) and 14:08 hours for the **fdroid** set (median: 8:52 hours). Using the MDP based method, we have an average of 23:49h and median of 15:54h for the **gplay** set, and an average of 13:22h and median of 10:58h for **fdroid** apps. Thus, our approach fits comfortably in the time frame of software processes that can be run on a daily (i.e., overnight) or weekly basis.

We tested the sequence of time measurements for all apps (**gplay** and **fdroid**) under the Wilcoxon Signed-Rank Test [32] and found a mean rank difference between the heuristic and MDP based methods with $p \approx 0.08$, which is not quite enough to be considered statistically significant. We do not have enough samples under the naive method to compare it against the other two under a similar test, but it can be seen from Figure 5 that this method often significantly underperforms compared to the other two.

5.4 Effects of application non-determinism

One final question regarding the trace minimization problem is on whether or not handling application non-determinism is truly a significant problem. As we discussed in Section 2, some Android applications present non-deterministic behavior under the same sequence of GUI events, motivating the need for running each event trace multiple times during minimization and estimating trace probabilities. However, if this is a problem that occurs only rarely, it might be that the techniques presented in this paper are not often required. We argue that application non-determinism is in reality a common problem, as can already be somewhat discerned from the fact that the check columns of tables 1 and 2 often show traces as succeeding in reaching the target activity less than 20 times in 20 runs.

To explore the impact of application non-determinism for trace minimization, we took the output minimized traces of our MDP condition for the **gplay** dataset, and attempted to further minimize them by using traditional non-probabilistic delta-debugging (i.e. by following the algorithm in [38] or, equivalently, by running `ND3MIN()` with $nr = st = 1$). In 8 out of 28 cases (29%), this produced a further reduced trace. We then ran each of these resulting traces an additional 20 times. Table 4 contrasts the size and reliability of the traces minimized under the original MDP based non-determinism aware condition, with that of the result of further applying traditional delta debugging to these traces. As we can see, these resulting traces succeed in reaching the target activity much less frequently than the originally minimized traces. Thus, it is clear that for traces that succeed non-deterministically, it is important to take into account their corresponding success probabilities during minimization. This likely becomes more significant the more steps delta debugging takes, as we can see by looking at the cases of the table above in which the minimal trace obtained by the MDP strategy is larger than 5 events.

6. RELATED WORK

Many tools exist for generating GUI event traces to drive Android apps. These tools, sometimes collectively called 'monkeys', are commonly used to automatically generate coverage of an app's behavior or to drive app execution as part of a dynamic analysis system.

Dynodroid [20] improves on the standard Android Monkey by automatically detecting when the application regis-

Table 4: Effect on application non-determinism in our dataset

Application	Key	Activity	non-deterministic mdp		+ deterministic DD	
			# events	check	# events	check
com.eat24.app	1-3	LoginActivity	3	20/20	2	6/20
	1-4	CreateAccountActivity	5	19/20	3	15/20
com.duolingo	2-4	SkillActivity	16	12/20	11	4/20
	2-5	LessonActivity	23	11/20	15	1/20
com.etsy.android	3-2	CoreActivity	3	20/20	2	12/20
com.ted.android	4-4	VideoActivity	11	18/20	10	1/20
com.google.android .apps.photos	7-2	LicenseMenuActivity	5	18/20	4	18/20
	7-4	PhotosAboutSettingsActivity	4	20/20	3	18/20
Average			8.75	17.25/20	6.25	9.38/20

ters for system events and triggering those events as well as standard GUI events. It also provides multiple event generation strategies which take the app context into account and it allows the user to manually provide inputs when exploration is stalled (e.g. at a login screen). Tools such as GUIRipper [1]/ MobiGUITAR [2], AppsPlayground [24], ORBIT [34], SwiftHand [9], and A^3E [4] dynamically crawl each app while building a model which records observed states, allowed events in each state, and state transitions. The generated model is used to systematically explore the app. PUMA [15] provides a general framework over which different model-based GUI exploration strategies can be implemented. ACTEve [3] is a concolic-testing framework for Android, which symbolically tracks events from the point in the framework where they are generated, up to the point at which the app handles them. EvoDroid [21] generates Android input events using evolutionary algorithms, with a fitness function designed to maximize coverage. Brahmastra [5] is another tool for driving the execution of Android apps, which uses static analysis and app rewriting to reach specific components deep within the app.

A recent survey paper by Choudhary et al. [10] compares many of the tools above and seems to suggest that the standard Android Monkey is competitive in coverage achieved in a limited time. One explanation is that Monkey compensates for what it lacks in sophistication by maintaining a higher rate of event generation. Of course, this unexpected result could also be the effect of comparing research prototypes against an industry standard tool, which performs more robustly even while using less sophisticated techniques. In either case, the effectiveness of the standard Monkey to achieve high coverage, along with the relative noisiness of the traces produced, justifies our focus on trace minimization.

In addition to general test input generation tools for Android, many dynamic analysis tools include components that drive exploration of the app being analyzed (e.g. [18, 23, 19, 25]). Input fuzzers can also generate application inputs, albeit restricted to testing for particular scenarios, such as inter-app communication errors [27] or invalid data handling [35], rather than aiming at GUI exploration.

Besides minimizing traces generated by Monkey-style tools, our approach is also applicable to minimizing recorded user-interaction traces. Tools such as RERAN [11], Mosaic [14] and VALERA [16] could be used to record the actions of a human tester as an input event trace for our method.

In addition to automated input generation tools, GUI-aware tests for Android applications tend to be encoded as testing scripts in frameworks such as Selendroid [30], Robotium [26], Calabash [33] or Espresso [12]. These frameworks allow scripting specific interaction scenarios with an Android app and adding checks to generate an effective application test suite. A promising line of future work is to automatically transform automatically-generated and minimized execution traces into test scripts expressed in any of these frameworks, as a way to provide a starting point for test suite writers.

Regarding our specific technique, the core of our algorithm is based on delta debugging. Delta debugging is a family of algorithms for sequence minimization and fault isolation, described originally by Zeller et al. [36, 38]. This technique has been extended to many scenarios [37, 8, 22, 6]. In particular, the work by Scott et al. [29, 28], extends delta debugging to minimize execution traces which trigger bugs within non-deterministic distributed systems. They run standard delta debugging over the traces of external (user triggerable) events. Then, to check each subtrace of external events, they instrument the system under test and explore the space of possible interleavings of internal events. By contrast, we treat non-deterministic Android applications in a blackbox manner and rely on modeling the probability of success of traces of external events, independently of the internal workings of the system being tested. For the specific case of minimizing GUI event traces in Android applications, an important source of non-determinism turns out to be responses from network services outside our control, justifying the need for a blackbox approach. In other scenarios, the tradeoff between both approaches likely depends on the complexity of the internals of the system being tested and the 'success' probability of the original trace to be minimized.

A Markov Decision Process is a standard model within the reinforcement learning literature (see e.g. [17, 31]). MDPs are used to solve a variety of problems across multiple domains, including optimizing resource consumption in mobile phones [7]. To the best of our knowledge, we are the first to apply them to the problem of trace minimization in testing.

7. CONCLUSIONS

We have presented the problem of minimizing large GUI event traces as a first step towards producing scripted test cases from the output of random input testing tools, such as Android's Monkey. We have shown a delta debugging extension that handles non-determinism, which we have shown is a pervasive issue in app behavior. We have also presented two strategies for efficient trace selection. Evaluation of our trace minimization method shows that the resulting traces are 55 times smaller while still reaching the same activity with high probability.

8. ACKNOWLEDGMENTS

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Additionally, this work was partially supported by the Amazon Web Services (AWS) Cloud Credits for Research program.

9. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 258–261, 2012.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[3] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 59, 2012.

[4] 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 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 641–660, 2013.

[5] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 1021–1036, 2014.

[6] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 221–231, 2011.

[7] T. L. Cheung, K. Okamoto, F. M. III, X. Liu, and V. Akella. Markov decision process (MDP) framework for optimizing software on mobile phones. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 11–20, 2009.

[8] J. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 11th International Symposium on Software Testing and Analysis, ISSTA 2002, Rome, Italy, July 22-24, 2002*, pages 210–220, 2002.

[9] W. Choi, G. C. 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 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 623–640, 2013.

[10] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 429–440, 2015.

[11] L. Gomez, I. Neamtiu, T. Azim, and T. D. Millstein. RERAN: timing- and touch-sensitive record and replay for Android. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 72–81, 2013.

[12] Google. Espresso - https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[13] Google. UI/Application exerciser monkey - https://developer.android.com/tools/help/monkey.html.

[14] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented Android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 215–224, 2015.

[15] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, pages 204–217, 2014.

[16] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 349–366, 2015.

[17] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res. (JAIR)*, 4:237–285, 1996.

[18] K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin. AMC: verifying user interface properties for vehicular applications. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, pages 1–12, 2013.

[19] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: detecting and characterizing ad fraud in mobile apps. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 57–70, 2014.

[20] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 224–234, 2013.

[21] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 599–609, 2014.

[22] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151, 2006.

[23] S. Nath, F. X. Lin, L. Ravindranath, and J. Padhye. SmartAds: bringing contextual ads to mobile apps. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, pages 111–124, 2013.

[24] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: automatic security analysis of smartphone applications. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, pages 209–220, 2013.

[25] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, pages 190–203, 2014.

[26] Robotium. Robotium - https://github.com/robotiumtech/robotium.

[27] R. Sasnauskas and J. Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA 2014, San Jose, CA, USA, July 22, 2014*, pages 1–5, 2014.

[28] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 291–309, Santa Clara, CA, 2016. USENIX Association.

[29] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 395–406, 2014.

[30] Selendroid. Selendroid - http://selendroid.io/.

[31] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 28. MIT press, 1998.

[32] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[33] Xamarin. Calabash - http://calaba.sh/.

[34] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 250–265, 2013.

[35] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android apps with intent-filter tag. In *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, Vienna, Austria, December 2-4, 2013*, page 68, 2013.

[36] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999*, pages 253–267, 1999.

[37] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pages 1–10, 2002.

[38] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.