Understanding Ineffective Events and Reducing Test Sequences for Android Applications

Ping Wang*‡, Jiwei Yan†, Xi Deng*‡, Jun Yan✉*†‡ and Jian Zhang✉*‡
* State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
† Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
‡University of Chinese Academy of Sciences, Beijing, China
Email: {wangping, yanjw, dengxi, yanjun, zj}@ios.ac.cn

***Abstract*—Monkey, which is integrated with the Android system, becomes the most widely used test input generation tool, owing to the simplicity, effectiveness and good compatibility. However, Monkey is based on coordinates of screen and oblivious to the widgets and the GUI states, which results in a great many ineffective events that have no contribution to the test. To address the major drawbacks, this paper parses the events of 200 test sequences generated by Monkey into human-readable scripts and manually investigate the effects of these events. We find three types of patterns on the ineffective events, including no-ops, single and combination of effect-free ones, and summarize them into ten rules for sequence reduction. Then, we implement a tool CHARD to match these patterns in real-world traces and prune the redundant events. The evaluation on 923 traces from various apps covering 16 categories shows that CHARD can process 1,000 events in a few seconds and identifies 41.3% events as ineffective ones. Meanwhile, the reduced sequence keeps the same functionality with the original one that can trigger the same behaviors. Our work can be applied to lessen the diagnose effort for record-and-replay, and as a preprocessing step for other works on analyzing sequences. For instance, CHARD can remove 72.6% ineffective events and saves 67.6% time of delta debugging in our experiments.**

***Index Terms*—Android Testing, Monkey, Event Trace Reduction, Record-and-replay**

I. Introduction

Android applications (apps) are becoming increasingly prevalent nowadays. The number of available apps in the Google Play [1] Store was most recently placed at 2.6 million apps in December 2018, after surpassing one million in July 2013 [18]. The most popular ones among them tend to be feature-rich, which puts forward higher requirements on testing. However, due to some characteristics of Android apps, such as the fragmentation of devices and the event-driven nature, it is exhausting to test them manually.

Heterogeneous automated testing tools for Android have been developed by practitioners and researchers. According to the thorough empirical study [3] by Choudhary et al., the simple tool Monkey [6] is a winner among the popular tools, while other complex techniques fail to outperform Monkey in ease of use, compatibility, code coverage, and fault detection.

However, Monkey has two major limitations: (1) a large number of events are redundant such that it usually takes an exceptional long test sequence as the price to reach high code coverage; (2) events generated by Monkey are low-level events which are hardly human-readable. These drawbacks increase the time cost to replay the test sequences when Monkey triggers a crash or reaches a special state. Moreover, developers are exhausted in inspecting the test trace during replaying, in order to understand the behaviors and low-level test events generated by the random test tool. Thus, reducing test sequences is important and essential.

Event trace reduction techniques are proposed to simplify test sequences. The basic idea is to set a target in advance and reduce target-irrelevant events using various techniques. Delta debugging (DD) [8] is a widely used algorithm, which uses an iterative process to repeat the reduction until it obtains a minimally sized subtrace. For Android test sequences, researchers use variants of DD to adapt to the characteristics of Android platform [4], [10]. Nonetheless, DD techniques are very time consuming, because they usually require re-executing the reduced sequences in the iteration process.

To reduce test sequences for Android applications in a faster way, we propose our approach based on the comprehension of events. According to the runtime details, such as system logs and the change of application state before and after a test event, we can infer the effectiveness of events. However, only the logs generated by Monkey is inadequate to understand what the event does. To understand the ineffective events better, we conduct a comprehensive manual study on 200 real traces each of which has 1,000 events, from 40 popular open-source apps. By manually reducing events repeatedly, we summarize the patterns of ineffective events, including no-ops, single and combination of effect-free ones, which can be leveraged to help the automatic test sequence reduction.

In this paper, based on the results of the above study, we extract execution information and build the sequence and event model. We define rules for patterns of ineffective events to reduce test sequences. For effect-free combination, we use extended finite state machines (EFSMs) to conduct the reduction. In order to deal with the nested combinations, we add a stack to the automaton. Additionally, we design a readable natural language format to describe each event for improving the comprehensibility.

To evaluate the proposed approach, we implement the approach into a tool CHARD (CompreHensibility And Re-

This work is supported by National Natural Science Foundation of China (Grant No. 61672505) and Key Research Program of Frontier Sciences, CAS, Grant No. QYZDJ-SSW-JSC036.

Duction) to analyze and reduce the test sequences generated by Monkey. CHARD is evaluated with two datasets, of which the first contains 890 test sequences from 74 Android apps covering 16 categories, and the other contains 33 test sequences with crashes collected from the related study [10]. The evaluation shows that CHARD can reduce test sequences effectively, correctly, efficiently.

The main contributions of this work are summarized as follows:

1) Through the manual study on test sequences of Monkey, we distinguish the removable events and summarize the patterns of the ineffective events;
2) Based on the above findings, we give event model and ten reduction rules to shorten them;
3) We implement the proposed techniques into a prototype tool CHARD and evaluate its effectiveness, correctness, and efficiency with real-world sequences.

The remainder of this paper is organized as follows. The Monkey and our empirical study are introduced in Section II. Then we present the ineffective patterns in Section III. In Section IV, we use a case study to show the execution of our approach. The evaluation of our tool CHARD is demonstrated in Section V. We discuss the related work and conclude our work in the last two Sections.

II. Event Sequences for Android Apps

In this section, we introduce Monkey sequence reduction problem and our findings on the types of ineffective events.

A. Monkey Sequence Reduction Problem

Because of the event-driven characteristic of Android applications, the interactions between the users and a specific application k form a sequence of user events. In this sequence, each event depends on the execution of the previous one. We first give a definition of the user event sequence.

Definition 1. (User Event Sequence). A user event sequence

$$\mathcal{U}(k) = E_1 \cdots E_m$$

for an Android app k is a sequence composed by m $(m > 0)$ separate user events, each of which is an atomic operation that can not be split.

We then formalize a user event in $\mathcal{U}(k)$ as follows.

Definition 2. (User Event). A user event E is a tuple (src, des, act) that denotes an entry in the user event sequence $\mathcal{U}(k)$, where

- src and des are the statuses before and after the event is triggered. The status of app can be described as a triple (att, win, wdt), where att, win and wdt are the corresponding activity, window and widget respectively. In the sequence $\mathcal{U}(k)$, the des of an event is the src of next one.
- act denote the action of this event, which changes the status of the app k from src to des. The action is a tuple (obj, tp), where obj is an object to be operated which may be a widget item or a system key, and tp is the type of the operation that originates from Android reference, including `Touch`, `Motion`, `Rotation`, `Nav`, `Flip`, `Trackball`, `PinchZoom`, `MajorNav`, `Appswitch`, `SysOp`. [14]

Monkey, which is currently integrated with the Android system, is regarded as the current state-of-practice for automated software testing [13]. Monkey supports all the types of event actions declared in Definition 2, which will be translated into a set of low-level events during the execution. For example, a `Touch` event is composed by *ACTION_DOWN* and *ACTION_UP*. Thus, we introduce the definition of Monkey event sequence.

Definition 3. (Monkey Event Sequence). A Monkey event sequence $\mathcal{M}$ which has n low-level Monkey events can be formalized as:

$$\mathcal{M} = e_1 e_2 e_3 \cdots e_n$$

We observed that Monkey event sequence has natural boundaries between actions (refer to Section IV). Therefore, we can rearrange the n low-level events into m high-level user events E and form a user event sequence after conducting $\mathcal{M}$ on app k:

$$\mathcal{M} = (e_1 \cdots)^1 \cdots (\cdots e_n)^m = E_1 \cdots E_m = \mathcal{U}_M(k).$$

This rearrangement accomplishes the translation from a Monkey event sequence $\mathcal{M}$ to user event sequence $\mathcal{U}_M(k)$, which makes the test sequence more user-friendly.

Furthermore, we find that the translated user event sequence contains a large number of ineffective events (see Section II-B), which increases the test efforts. Thus, this user event sequence $\mathcal{U}_M(k)$ can be optimized into a new refined one $\mathcal{U}_M^R(k)$ by removing ineffective events.

In summary, our work in this paper is to solve the *Monkey sequence reduction problem*, i.e.,

$$\mathcal{M} \rightarrow \mathcal{U}_M^R(k).$$

The key challenge is how to understand user events and reduce the ineffective ones, which will be addressed in the rest of this Section.

B. Ineffective Event

Because Monkey does not automatically generate a script to record the test sequences, we define a script format to record the information. We enhance Monkey into Moneky$_{RR}$ to record script automatically and replay the test sequence in this script format. In the rest paper, we use Moneky$_{RR}$ to implement experiments, which can automatically record a test sequence by a script and replay the reduced script.

In order to investigate the properties of the test sequences generated by Monkey, we conduct an empirical study manually. We collect 40 apps covering 14 categories from a widely used website F-Droid [5], which are listed in Figure 1. Finally, we use Moneky$_{RR}$ to generate test sequences for each app and reduce them manually. According to our investigation, we find two types of ineffective events that should be removed, including no-ops and redundant events. The results of our empirical study are shown in Figure 1. The *#NE* and *#RE*

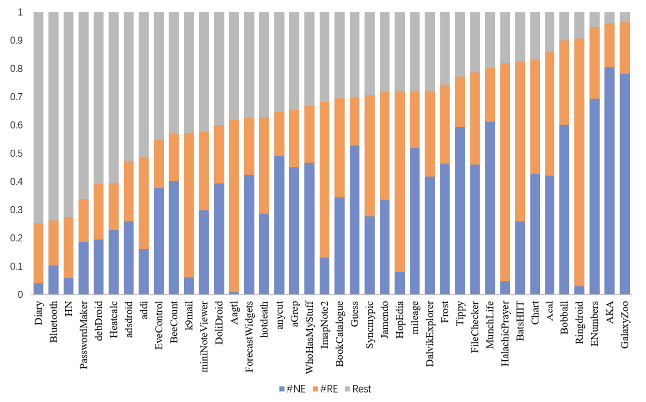


Fig. 1: The Distribution of Ineffective Events

represent the proportions of no-ops and redundant events, respectively.

1) No-ops Events: By matching each event with its behavior, we find that many events are no-ops events. A no-ops event E is an event that does not link to any widget or operates on a widget with an unacceptable type of event, i.e., the action $E.act$ is invalid. According to the Android response mechanism, these events are not intercepted or consumed, i.e., the Android system does not respond to them. The approach to identifying a no-ops event is to analyze whether the execution of this event triggers any differences in the status of activity or window and the attribute of widgets. If there is no difference, the event will be defined as a no-ops event.

After reducing no-ops events, the average rate of operational events is approximately 65.9%. There are several special cases. `Aagtl` is a map application with a full-screen map, leading that 99% events are operational. `AKA` is an app that needs to download sources firstly, and this action is often refused for the non-responsive server. The home page will be blank, which causes the extremely low operational rate 19.6%.

2) Redundant Events: After filtering out the no-ops events, we find that there are many other events that can be further reduced without sacrificing the effectiveness of the whole test sequence. We say one or more events $E_i \cdots E_j$ $(j \geq i)$ are redundant if they contain at least one valid action while their executions keep the status of the app ($E_i.src = E_j.des$). The redundant events are also regarded as ineffective events for they have no contribution to the code coverage or do not change the environment. By investigation, we classify the redundant events into two categories: **single effect-free events** $(i = j)$ and **effect-free combinations** $(i < j)$.

The single effect-free event contains rejected event and optional event. A rejected event is an event which is abandoned in Monkey. Due to its setting, Monkey only allows testing in the package of the apps under test. Thus, activities out of the specific packages are rejected to be opened by Monkey. For example, when the `HOME` button is triggered or `BACK` on the `MainActivity` will lead to jumping to a target beyond the scope of testing. Optional events denote a set of special system events, which are redundant to most of the apps while being essential in some kind of apps. For instance, the operations of volume buttons are unimportant for most applications, which are cut out by default. However, some applications overwrite the functions of the volume button, e.g., e-book uses them to turn pages.

An effect-free combination contains two or more effective events, of which the last one completely counteracts the effect of the former. For example, an event in the sequence opens a new window, while the next one closes the window by clicking the back button on the phone without impact on the other events. In this case, these two events compose an effect-free combination Contrary Pair and should be removed in general. Though each event in the pattern is effective, the combination of these events is redundant in most cases. According to the experience of manual reduction, we summarize the combinations in Table I. After reducing ineffective events, the *Rest* represents the effective events, whose rate is approximately 34.1%.

C. *Significance Levels of Events*

During the manual record-reduce-replay process, we find that reducing different event affect differently on the change of a test sequence. For example, if we directly delete a window-opening event, which will influence its following events, it will cause a wrong state. That is to say, the following events will be triggered in the wrong window. Thus, the result of executing the reduced sequence will be completely different from the original one. In contrast, when a no-ops event is removed, the reduced sequence has the same runtime behavior as before.

Thus, according to the differences that the deletion could make, we define four levels of the significance:

TABLE I: Types of Ineffective Events

Type	Name	Description
Single Effect-free Event	Rejected event	`HOME` button
		press `BACK` button on the `MainActivity`
	Optional Event	`VOLUME-UP` button, `VOLUME-DOWN` button, `CALL` button
Effect-free Combination	Contrary Pair	open a dialog window, and cancel it immediately or close it by click invalid region
		open an activity, and back immediately
		select one item of checkboxs, and disable it
	Last Effective	select an item in a group of radios, and select another immediately
		click a text box with no input, and click it again
	Text Slide	open a window of long text, slide the text without log and exit in the end
	Hidden Menu	click window to open hidden menu bar, select one option to open a window, then close it

- Essential – The execution of the event changes the activity or window, bringing about complete changes in the current layout.
- Major – The event triggers functionalities of the application.
- Minor – The event operates widgets or Android system.
- Trivial – The event is no-ops.

The levels help developers to judge whether to delete the event or not and customize the different reduction strategies.

For describing significance level, we define a function ℓ : $E \to \{$*Essential, Major, Minor, Trivial*$\}$. Each level has its criteria showed in Table II. If an event satisfies more than one criterion, the higher level will be chosen.

III. Patterns for Ineffective Events

We define ten patterns to represent the ineffective events, which should be removed. Every pattern has a corresponding rule to delete it. There are three types of ineffective events that need to be removed: no-ops events, single effect-free events, and effect-free combinations.

A. *No-ops Events*

The events in level Trivial are no-ops events. Thus, we define the following pattern.

- **Pattern 1.1** $E \vDash \ell(E) = Trivial$.

According to the criteria in Table II, we can obtain the significance level for each event. If an event is Trivial, i.e., satisfies Pattern 1.1, this event will be removed.

B. *Single Effect-free Events*

Single effect-free events include rejected events and optional events. For the two types of rejected events in Table I, we define two patterns to identify them.

- **Pattern 2.1** $E \vDash \mathit{act.tp} \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\}$, $\mathit{act.obj} = \texttt{HOME}$.
- **Pattern 2.2** $E \vDash \mathit{act.tp} \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\}$, $\mathit{act.obj} = \texttt{BACK}$, $\mathit{src.act} = MainActivity$.

For optional events, the best way to identify whether the events are effective is to enable users to configure them. We also have a default rule to judge by the level of events, which is safe but imprecise. If the level of an optional event is Major or Essential, this optional event may be useful in the app and should not be deleted.

- **Pattern 2.3** $E \vDash \mathit{act.tp} \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\}$, $\mathit{act.obj} = \texttt{CALL}$, $\ell(E) = Minor$.

TABLE II: Criteria of Event Significance Level

$\ell(E)$	Criteria
Essential	$E.act.tp \in \{Rotation, Appswitch, PinchZoom\}$ $\vee E.des.win \neq E.src.win$ $\vee E.des.att \neq E.src.att$
Major	$GetLog(E) \neq \texttt{null}$
Minor	$E.act.tp \in \{SysOp, MajorNav, Nav, Trackball\}$ $\vee$ $E.act.obj \neq \texttt{null}$
Trivial	other

- **Pattern 2.4** $E \vDash \mathit{act.tp} \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\ \}$, $\mathit{act.obj} \in \{\texttt{VOLUME\_UP}, \texttt{VOLUME\_DOWN}\}$, $\ell(E) = Minor$.

If an event satisfies any item in Pattern 2, this event is single effect-free and will be removed.

C. *Effect-free Combinations*

We use extended finite state machine (EFSM) [2] to describe each type of the combinations in Table I.

Definition 3. (Extended finite state machine). The extended finite state machine is a tuple $M = (Q, \Sigma, I, V, C, \Lambda)$, where

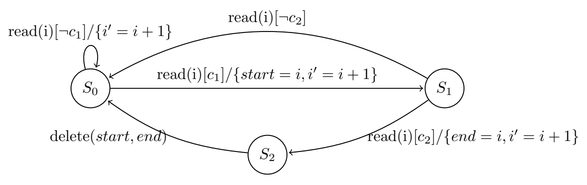
- Q is a finite set of states,
- Σ is a finite set of operations (also called events),
- $I \subset Q$ is the set of initial states,
- V is the set of state variables and every variable is a global variable,
- C is a finite set of conditions,
- Λ is a finite set of transitions. A transition is $q \xrightarrow{e[g]/a} q'$, where $q, q' \in Q$, $e \in \Sigma$, $g \in C$ is a condition and a is an action. A variable x affected in the transition specification $q \xrightarrow{e[g]/a} q'$ will be denoted x' in state q'. All parts of a transition label are optional.

According to the progress of event reduction, we define two operations in Σ and two variables in V. The `read(i)` will input E in the test event sequence to the EFSM. The `delete(m,n)` will delete the subsequence from event labelled m to event labelled n. Variable $start$ and end record the position of a pair or subsequence that need to be deleted in the original test event sequence. $\Sigma = \{\texttt{read(i)}, \texttt{delete(m,n)}\}$. $V = \{start, end\}$. The four EFSMs are shown in the Figure 2.

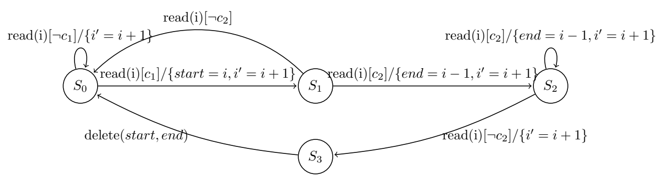
As we can see in Figure 2(a), there are two conditions c_1 and c_2. The c_1 is the condition that an event has the property of the first event in Contrary Pair, and the c_2 is the condition that an event and its former one compose Contrary Pair. We conclude and keep these conditions for each property in a collection. There are three actions, $start = i$ ($end = i$) is used to record the start (end) event that may need to be deleted, while $i' = i + 1$ indicates that the value of i plus one after this transition. Especially, in the transition from S_1 to S_0 on the condition $\neg c_2$, there is no action $i' = i + 1$, for keeping the current event unchanged, in case that this event has another property of Contrary Pair that may match the next event.

Thus, we use the four extended finite state machines in Figure 2 to conduct reduction for effect-free combinations respectively. In a few complex cases, the Contrary Pair may be nested by itself. We use the idea of the pushdown automaton to deal with these cases. A pushdown automaton [17] is a finite state machine with an infinite stack. The stack helps to reduce the complexity of the algorithm to deal with the nested Contrary Pair. Thus, we declare a stack and define the operations of the stack on our extended finite state machine in Figure 3.

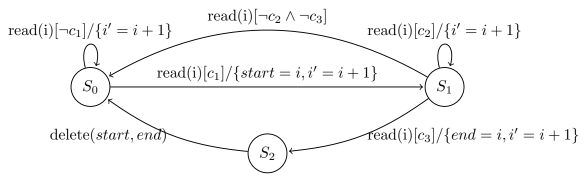
S_0 is the initial state. The initial stack is empty. The condition c is that the top event of the stack matched the


read(i)[$\neg c_1$]/{$i' = i+1$}
read(i)[$\neg c_2$]
read(i)[c_1]/{$start = i, i' = i+1$}
S_0
S_1
delete($start, end$)
read(i)[c_2]/{$end = i, i' = i+1$}
S_2

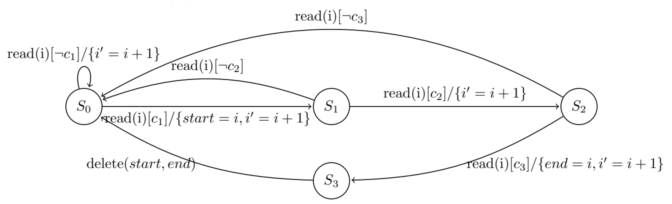
(a) Pattern 3.1 for Contrary Pair


read(i)[$\neg c_2$]
read(i)[$\neg c_1$]/{$i' = i+1$}
read(i)[c_2]/{$end = i-1, i' = i+1$}
read(i)[c_1]/{$start = i, i' = i+1$}
read(i)[c_2]/{$end = i-1, i' = i+1$}
S_0
S_1
S_2
delete($start, end$)
read(i)[$\neg c_2$]/{$i' = i+1$}
S_3

(b) Pattern 3.2 for Last Effective


read(i)[$\neg c_2 \wedge \neg c_3$]
read(i)[$\neg c_1$]/{$i' = i+1$}
read(i)[c_2]/{$i' = i+1$}
read(i)[c_1]/{$start = i, i' = i+1$}
S_0
S_1
delete($start, end$)
read(i)[c_3]/{$end = i, i' = i+1$}
S_2

(c) Pattern 3.3 for Text Slide


read(i)[$\neg c_3$]
read(i)[$\neg c_1$]/{$i' = i+1$}
read(i)[$\neg c_2$]
read(i)[c_2]/{$i' = i+1$}
S_0
S_1
S_2
read(i)[c_1]/{$start = i, i' = i+1$}
delete($start, end$)
read(i)[c_3]/{$end = i, i' = i+1$}
S_3

(d) Pattern 3.4 for Hidden Menu

Fig. 2: Patterns for Effect-free Combinations

current event E, i.e., they are Contrary Pair that should be removed. If E satisfies c, S_1 is reached and the top event in the stack will be popped. Otherwise, S_2 is reached and the current event will be pushed into the stack. After running the automaton, the nested Contrary Pairs are removed. The events in the stack are the left events.

D. *Readable Natural Language Interpreter*

To improve the readability of the reduced test sequence, we use readable natural language to describe an atomic event. The natural language format is defined as follows:

Line $i - j$ is a $\langle$*type*$\rangle$ event that $\langle$*action*$\rangle$, causing $\langle$*result*$\rangle$, level $\langle$*level*$\rangle$.

Line $i-j$ represents the certain event in the Monkey script. The $\langle$*type*$\rangle$ is the type for the event. The $\langle$*action*$\rangle$ describes the behaviour of the event, e.g., clicking a button named *OK* or pressing *BACK* button. The $\langle$*result*$\rangle$ describes the change from src to des, which lists the changes caused by this event, such as opening an Activity or triggering functions of the app. The $\langle$*level*$\rangle$ will be replaced by the value $\ell(E)$.

According to this description, developers having knowledge of this application could repeat the test case manually and even find the exceptional events according to the result of events.

IV. Case Study

To better introduce our approach, we use a case study to show the execution and results of our tool.

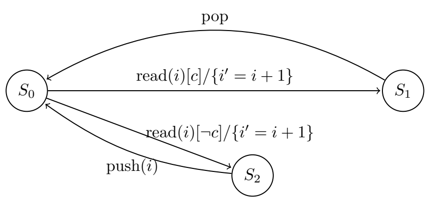


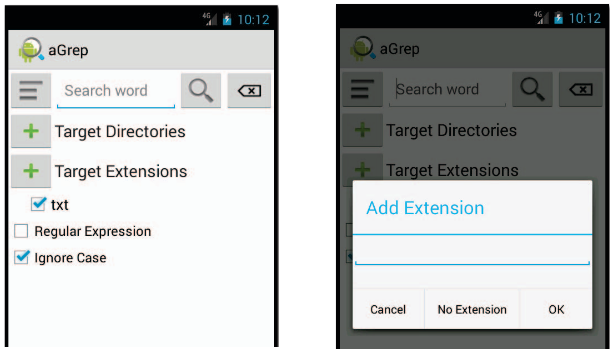
Fig. 3: Pattern 3.5 for the nested Contrary Pair

aGrep is an open-source text search Android application, which is available from Google Play and has more than ten thousand downloads. Its entry page is shown in Figure 4(a). When the user clicks on the button "Target Extensions", the page will jump to Figure 4(b). We use Monkey to generate 100 events on it and display part of the result in the following list.

```
1  :Switch: #Intent;action=android.intent.action.MAIN
       ;category=android.intent.category.LAUNCHER;
       launchFlags=0x10200000;component=jp.sblo.
       pandora.aGrep/.Settings;end
2  Sleeping for 300 milliseconds
3  :Sending Touch (ACTION_DOWN): 0:(461.0,240.0)
4  :Sending Touch (ACTION_UP): 0:(454.263,229.428)
5  Sleeping for 300 milliseconds
6  :Sending Touch (ACTION_DOWN): 0:(268.0,145.0)
7  :Sending Touch (ACTION_UP): 0:(268.951,140.663)
8  Sleeping for 300 milliseconds
9  :Sending Touch (ACTION_DOWN): 0:(140.0,181.0)
10 :Sending Touch (ACTION_UP): 0:(142.734,179.520)
11 Sleeping for 300 milliseconds
12 :Sending Key (ACTION_DOWN): 4
13 :Sending Key (ACTION_UP): 4
14 Sleeping for 300 milliseconds
15 :Sending Touch (ACTION_DOWN): 0:(356.0,443.0)
16 :Sending Touch (ACTION_UP): 0:(344.819,440.863)
17 Sleeping for 300 milliseconds
18 :Sending Key (ACTION_DOWN): 5
19 :Sending Key (ACTION_UP): 5
```

As we can see, this test sequence is poorly human-readable. According to our study, the events in line 3 and 4 consist of an atomic event that touches on the blank area, which is a no-ops event. The following two events in line 6-10 achieve clicking the same text box twice and compose an effect-free combination named Last Effective in which only the last event should remain. Lines 18 and 19 consist of an event that will open the phone call page, which is an optional event and should be deleted from this app. There are also many effect-free combinations named Contrary Pair. For example, two events that one after another click on the same checkbox.

We convert the original 100 events into 43 atomic events totally, with 18 no-ops and 11 redundant. After analysis, the



(a) (b)

Fig. 4: Application aGrep

test sequence can be reduced to 14 events. We can correctly replay the reduced event trace saving 65% time. The readable report is shown in the following list.

```
line 1-1 is a Appswitch event that opens the entry
        page of {jp.sblo.pandora.aGrep}, causing open
        activity (.Settings), level Essential.
line 9-10 is a Touch event that clicks on EditText
        with the text "Searchword", level Minor.
line 12-13 is a SysOp event that KEYCODE_BACK,
        causing close keyboard, level Essential.
line 15-16 is a Touch event that clicks on
        CheckBox with the text "IgnoreCase", causing
        trigger program execution and generate logs,
        level Major.
```

V. Evaluation

To evaluate the proposed approach, we set up the following research questions:

- RQ1: (Effectiveness): How many events can be reduced for various apps?
- RQ2: (Correctness): Will the reduction influence the execution result?
- RQ3: (Efficiency): To what extent can it save the time on reduction for the crashed test sequences?

A. Implementation

We implement a tool coined as CHARD (CompreHensibility And ReDuction) to achieve the proposed approach, and build the automatons in Java language.

The overall design of our system is shown in Figure 5. We use tool InsDal [11] to instrument all the methods of the app under test. On Android devices, Moneky$_{RR}$ takes this instrumented app as input and generates a test script and log file, which is invoked via ADB (Android Debug Bridge). On PCs, in *Rule*, there are ten ineffective patterns for reduction. In *Configuration*, users can configure these reduction rules. In *Event Comprehension*, test sequence model is built by analyzing the information collected from the log files. In *Test Sequence Reduction*, in the guide of *Rule* and *Configuration*, the foregoing kind of ineffective events, including no-ops events and redundant ones, will be removed from the test sequence. The outputs of the system are the reduced test

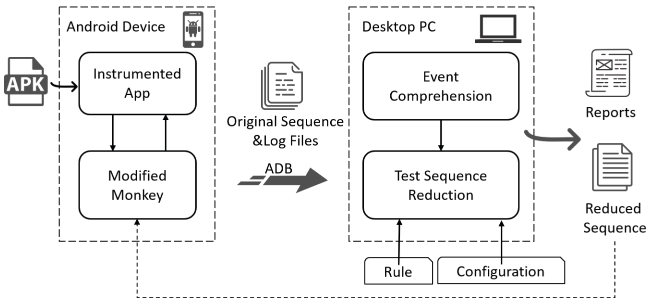


Fig. 5: Overview of Our Approach

sequence, which can be reused via Monkey, and the reports including the readable test sequence and the progress of reduction.

B. RQ1: Effectiveness

In the study for RQ1 and RQ2, we collected a set of 74 apps from F-Droid covering 16 categories, aiming to test the effectiveness and correctness under various situations whatever kind it may be. We list categories and statistics of sizes in Figure 6. The average of size is around two thousand KB. In addition, 35 of the 74 apps are available on the Google Play store. We also collected such information as the rating of each app, and the numbers of downloads. The ratings range from 3.6 to 4.8. The average number of downloads is 50,000.

After data sets selection, we use Moneky$_{RR}$ to generate 10 sequences for each app with 1000 events. Among them, there are 132 sequences from 35 apps that abort due to error. Crashes are caused by 8 kinds of exception, which are listed in Table III. Column *AppC* and column *SeqC* represent the count of apps and the count of sequences with this type of exception respectively. Because 4 apps and 2 sequences have more than one type of errors, the first two values in row sum are larger than 35 and 132. According to the previous experiments [10], not all of the crashes are repeatable. Thus, we repeat each sequence 3 times to confirm its stability and display the number of successful ones in column *Repeat*.

In addition, in order to check our approach could deal with long sequences, we pick 15 stable apps to generate 10

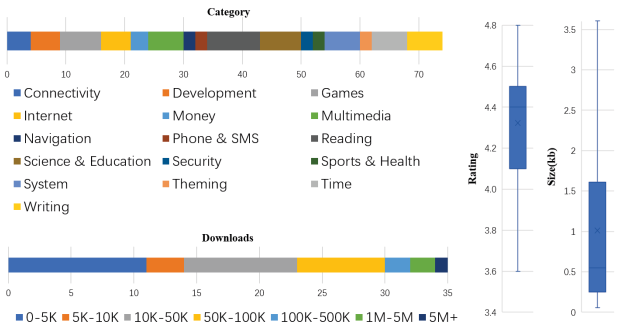


Fig. 6: Statistics of Apps in Study for RQ1 and RQ2

TABLE III: Statistics of Sequences with Bugs

Type	AppC	SeqC	Repeat
ActivityNotFoundException	2	7	4(5/4/5)
ArithmeticException	1	7	5(6/6/7)
IllegalArgumentException	3	16	13(16/13/16)
NoClassDefFoundError	2	14	12(14/12/14)
NullPointerException	8	26	22(25/22/25)
NumberFormatException	2	10	10(10/10/10)
OutOfMemoryError	1	4	2(2/5/5)
Native crash: Segmentation fault	25	50	2(11/16/10)
Sum	44	134	70(89/88/92)

sequences for each app with 10,000 events. In total, we obtain 890 test sequences in two scales.

Finally, we use CHARD to conduct experiments on these test sequences with default configurations, using all the 10 rules. After execution of CHARD, a sequence can be interpreted to a natural language report which describes the events in detail and can be reduced and saved in a new script. For the test sequences, statistics show that 45% events are level Major that trigger some functionalities of the application, 13% events are Essential that change the current window of the app, 24% events are Minor, most of which are input events on the keyboard, 18% events are Trivial, all of which are no-ops events. After reduction, all of the events in level Trivial are deleted according to Pattern 1.1. The deleted events in other levels are classified by types of ineffective events. The average rates of different types of ineffective events are shown in Figure 7. The left pie-chart shows that 58.7% events are left and 18% events are no-ops. The right pie-chart shows the types of other 23.3% ineffective events in detail.

According to the results, the rules that we defined could reduce 41.3% events. Our approach has good performance on sequence reduction in various apps covering 16 categories in two scales.

C. RQ2: Correctness

Due to the non-determinism of Android apps, executing the same sequence twice might result in different results. To exclude these situations, we have to pick stable sequences to prove the correctness of our approach. For the sequences without crashes, it is difficult to choose a standard criterion to check their consistency. Thus, we use 70 sequences mentioned in Table III, which are stable and trigger crashes in the end. The 70 sequences with crashes are from 16 apps. We use

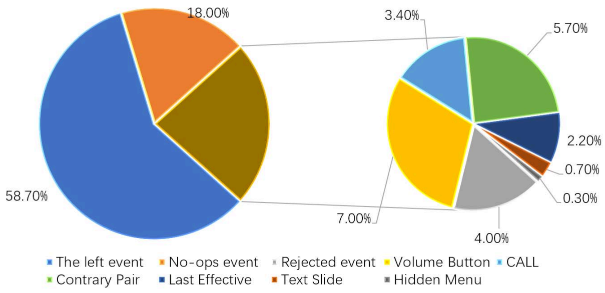


Fig. 7: The Distribution of Different Types

CHARD to reduce them and replay the reduced script by Monkey. Then, we record the results of the execution and compare with the former results. The statistics are shown in Figure 8. The 66 green dots represent the sequences that replay the former crash successfully. In contrast, the 4 red dots are failed. These sequences are failed because crashes are related to execution time. On the one hand, the attributes of some widgets are changeable with time, such as text and location. For example, `Acal` has ads to interfere in the layout. The changeable widgets in `BatHIIT` are used for timing. On the other hand, the time for network loading is uncertain, such as music player `Jamendo`.

The event numbers in the sequences before and after reduction are shown in Figure 8. The average reduction rate is around 65.5%. In conclusion, our approach deletes the ineffective events effectively and could keep the key operations at the same time.

D. RQ3: Efficiency

As we mentioned earlier, delta debugging (DD) has been applied to perform event trace reduction. Jiang et al. [10] proposed a tool named SimplyDroid to enhance the DD algorithm to simplify crash traces generated by Monkey. The reduced scripts are the shortest sequences to trigger the same crash. However, due to the iterative reduction progress, it costs a long time. Our approach, test sequence reduction based on comprehension, could help to preprocess the test sequences and reduce the time of delta debugging.

First, we repeated the experiments of 92 crash traces of SimplyDroid. 40 traces they used come from variants of apps, which cannot be obtained. In the rest 52 traces, though we use the same configuration, due to the different devices and special characteristics of apps, 19 original scripts cannot trigger crashes. We remove these traces and pick up 33 successful crash traces.

Second, we use CHARD to deal with these 33 traces of 4 apps, with all the reduction rules. The results are shown in Figure 9. The `original` represents the original count of the atomic event. The `reduced` represents the count of the left atomic events after preprocessing by CHARD. The average reduction rate is 72.6%, and all of these traces replay crash

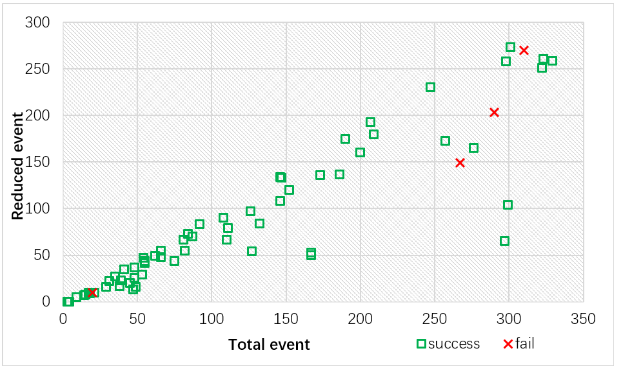


Fig. 8: Results of Reducing and Replaying

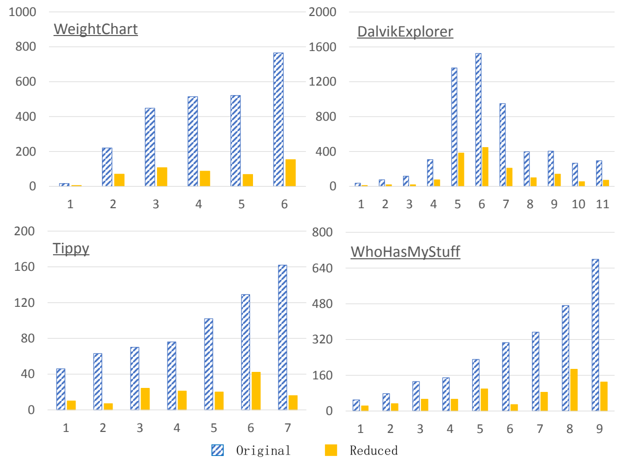


Fig. 9: Number of Events Before and After Reduction

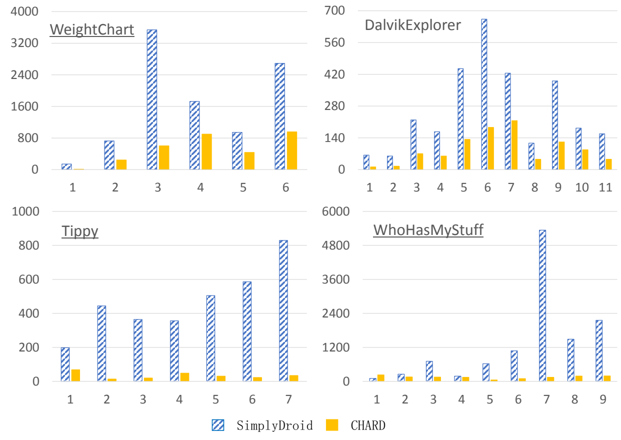


Fig. 10: Comparison of Execution Time

successfully. We record the time of the execution. The longest execution time is 2395 milliseconds to deal with the sixth sequence of `DalvikExplorer`. CHARD takes an average of 1307 milliseconds to deal with 1000 events.

Third, we used SimplyDroid to reduce these scripts pre-processed by CHARD and recorded the time. The results are shown in Figure 10. The *SimplyDroid* represents the original time used by SimplyDroid. The *CHARD* represents the sum of time used by CHARD and SimplyDroid. Our approach could help *SimplyDroid* save 67.6% time on average.

Finally, we analyze the reduced scripts generated by SimplyDroid to obtain the rate of different level events. We find that level Essential accounts for 83% of the events that lead to the crash. And the rest are 11.5% level Major and 5.5% level Minor. It's no surprise that events of level Trivial contain nothing to the crashes.

In this study, the efficiency of our approach can reflect perfectly. CHARD could reduce the sequences in a shorter time and guarantee the correctness. In addition, the significance levels we defined show the reasonableness by the statistics from the final scripts.

VI. Related Work

Besides Monkey, there are many test input generation tools based on random testing strategy. Dynodroid [12] based on Monkey has more sophisticated strategies and could trigger effective events only. PUMA [16] is also based on the same random exploration as Monkey, and integrates a generic UI Automator to obtain the information of the layout. Both of them reduce a lot of redundant events. We improve Monkey from another aspect. We will not change the original strategy of Monkey and improve the usability by reducing the test sequence after execution.

Event trace reduction techniques include delta debugging and program slicing. Towards the extremely long test cases with crashes generated by Monkey, SimplyDroid [10] uses delta debugging (DD) [21] to simplify Android input event traces. It enhances the DD algorithm and contributes to adjusting DD with Android. However, the DD can be used on crashed traces only, and the iteration is time-consuming. Our approach can pre-process these traces and increases the efficiency of DD significantly. Program slicing [7] [20] [22] is another main debugging technique, used in traditional programs. EFF [23] aims to reduce the event trace for UNIX by using dynamic slicing technique. JSTrace [19] is a tool that uses a novel dynamic slicing technique and reduces the event trace of the web application, aiming to effectively cut down error reproduction time. However, event trace reduction focuses on the minimal subset that triggers the same error. Our approach can keep the same behaviors of the apps, even though no error occurs.

There are also some works related to reproducing errors and reduction based on rules, aiming to help developers diagnose the result and bug reports. AppDoctor [9] is a system for efficiently and effectively testing apps under various systems and user actions. It uses heuristic rules to reduce the event sequence. CRASHSCOPE [15] tests apps by using systematic exploration and generates a human-readable report showing the steps to reproduce crashes. Our tool CHARD uses heuristic rules to reduce and generate readable reports together.

VII. Conclusion

For the testing of Android apps, we focus on the widely used tool Monkey, which randomly generates low-level events without regard to the state of the app. By investigation, we find that many events of Monkey are ineffective ones. Thus, the test reduction is a promising technique for improving the quality of the test sequences generated by Monkey. We conduct a two-stage study on the traces for testing real-world apps. For various apps, the approach can effectively reduce more than 40% events in the traces. In particular, for crash traces, the approach removes over 65% events and guarantees the replay of the crash in normal situations. In addition, our approach can be a pre-processing step for delta debugging, and save 67.6% time for it. Our tool CHARD can be used to rearrange the original test cases for replaying by removing the ineffective events.

References

[1] Google play. https://play.google.com/store?hl=en. The Google Play application market.

[2] V. S. Alagar and K. Periyasamy. *Specification of Software Systems*. Graduate Texts in Computer Science. Springer, 1998.

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

[4] L. Clapp, O. Bastani, S. Anand, and A. Aiken. Minimizing GUI event traces. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 422–434, 2016.

[5] F-droid. https://f-droid.org/.

[6] Google. Android monkey. http://developer.android.com/tools/help/monkey.html.

[7] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. 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, Proceedings*, pages 303–321, 1999.

[8] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 333–344, 2015.

[9] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 18:1–18:15, 2014.

[10] B. Jiang, Y. Wu, T. Li, and W. K. Chan. Simplydroid: efficient event sequence simplification for Android application. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 297–307, 2017.

[11] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. Insdal: A safe and extensible instrumentation tool on dalvik byte-code for Android applications. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 502–506, 2017.

[12] 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.

[13] 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.

[14] K. Mao, M. Harman, and Y. Jia. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 94–105, 2016.

[15] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, pages 33–44, 2016.

[16] C. Qian, L. Huang, M. Cao, J. Xie, and H. So. PUMA: an improved realization of MODE for DOA estimation. *IEEE Trans. Aerospace and Electronic Systems*, 53(5):2128–2139, 2017.

[17] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[18] Statista. Number of available applications in the google play store from december 2009 to december 2018. https://www.statista.com/statistics/266210/number-of-availableapplications-in-the-google-play-store/.

[19] J. Wang, W. Dou, C. Gao, and J. Wei. Fast reproducing web application errors. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 530–540, 2015.

[20] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

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

[22] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 319–329, 2003.

[23] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 81–91, 2006.