# SimplyDroid: Efficient Event Sequence Simplification for Android Application*

Bo Jiang, Yuxuan Wu, Teng Li
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiangbo, wuyuxuan, liteng}@buaa.edu.cn

W.K. Chan†
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—To ensure the quality of Android applications, many automatic test case generation techniques have been proposed. Among them, the Monkey fuzz testing tool and its variants are simple, effective and widely applicable. However, one major drawback of those Monkey tools is that they often generate many events in a failure-inducing input trace, which makes the follow-up debugging activities hard to apply. It is desirable to simplify or reduce the input event sequence while triggering the same failure. In this paper, we propose an efficient event trace representation and the SimplyDroid tool with three hierarchical delta-debugging algorithms each operating on this trace representation to simplify crash traces. We have evaluated SimplyDroid on a suite of real-life Android applications with 92 crash traces. The empirical result shows that our new algorithms in SimplyDroid are both efficient and effective in reducing these event traces.**

***Index Terms*—Test case reduction, delta debugging, event sequence reduction, Android**

## I. Introduction

The mobile Internet industry has witnessed an explosive growth in recent years. Both the number and the complexity of mobile applications have increased rapidly. According to Gartner [33], the Android OS has taken 87.8% of the smart phone market share in the 3rd quarter of 2016. In Google Play [34], there are more than 2.6 million Android applications [38].

To increase the satisfaction of end-users, mobile application developers must improve the quality of their applications. Mobile testing is one important measure to achieve this goal. Different kinds of testing techniques [8] have been proposed to test Android applications, including fuzz testing, GUI traversal-based testing [3][17], and search-based testing [18]. Fuzz testing is represented by the family of Monkey tools, which includes the built-in Monkey tool of Android OS and its improved versions by third party. Owing to their simplicity, effectiveness, and wide applicability, the improved monkey tools are widely adopted by cloud-based mobile testing platforms [35][39][41].

A major limitation of these fuzz testing techniques is that they often generate a large number of input events before triggering a failure, which makes follow-up debugging tasks hard to apply. In such scenarios, simplifying input event sequence that triggers the same failure is desirable.

Delta debugging (DD) [27] has been applied to perform such test input reductions on traditional applications, Web applications, and compilers, etc. There are also research works on reducing input event sequences for Android applications with the emphasis on handling execution non-determinism based on the idea of DD [9]. Nonetheless, in general, DD techniques are slow to generate the reduced input. The problem with the DD strategy is that its partition strategy is unaware of the presence of interaction sessions of input events with end users, which may lead to large number of unsuccessful trials (i.e., fail to trigger failure) in the reduction process.

Hierarchical Delta Debugging [19] improves the efficiency of DD by revealing the hierarchical structures of test cases and applying DD according to such structures. We observe that an Android input event sequence can be structured hierarchically based on interaction sessions and sub-session with users.

Our insight into the events trace reduction problem is that a long input event sequence often contains sub-sequences representing small interactive sessions with the end users. These sessions of events forms a natural boundary for reduction as they can often be reduced together with high probability. We further observe that the relationship of input events is reflected in the hierarchical relationship of their corresponding GUI states. Thus, we seek to find the hierarchical relationships between input events by building and analyzing the GUI state hierarchy tree as trace representation.

Based on the trace reprsentation, we further propose an input event sequence simplification tool SimplyDroid, which contains a family of three test case reduction algorithms based on the notion of hierarchical delta debugging coined as Hierarchical Delta Debugging (HDD), Balanced Hierarchical Delta Debugging (BHDD), and Local Hierarchical Delta Debugging (LHDD) each operating on the above trace representation. The HDD algorithm is an adaption of the existing notion of hierarchical delta debugging to show the applicability of our trace representation. Both BHDD and LHDD are our new HDD algorithms that use the structural property of the trace representation to improve the reduction efficiency without significant loss of reduction effectiveness.

We have used 92 input traces with crash occurrences from 8 real-life Android applications with real and seeded faults to evaluate the SimplyDroid tool. The experimental results show

* This research is supported in part by NSFC (project no. 61772056), the Research Fund of the MIIT of China (project no. MJ-Y-2012-07), the RGC GRF of HKSAR (project nos. 111313, 11201114, 11200015, and 11214116), and the research fund of the State Key Laboratory of Virtual Reality Technology and Systems.
† Correspondence author

that all the three techniques have improved reduction efficiency over the classic DD technique without loss of effectiveness. Furthermore, LHDD is significantly more efficient than DD, HDD and BHDD, which makes it competitive to apply.

The contribution of this paper is three-fold. First, it presents a novel GUI state hierarchy tree representation to model an Android input event trace. Second, it proposes the first family of efficient HDD algorithms for simplifying Android input event traces. Third, it reports the first comprehensive experiment on 92 crash traces of 8 real-world Android applications that evaluates the effectiveness and efficiency of our family of HDD algorithms and trace representation.

The organization of the remaining sections is as follows. In Section II, we present a motivating example to illustrate our key ideas. In Section III, we present our tool SimplyDroid [1] as well as the family of input sequence reduction algorithms. Section IV reports a comprehensive experimental study, in which we have evaluated the effectiveness and efficiency of our family of HDD algorithms operating on our trace representation, followed by the related work in Section V. Finally, we conclude our work in Section VI.

## II. MOTIVATING EXAMPLE

This section presents a motivating example of our work.

### A. GUI State Hierarchy Tree as Trace Representation

To understand the hierarchical structure of GUI input events, we have to build a partial GUI state hierarchy tree and map the input events to it. Here "partial" means the GUI hierarchy tree only reflects those parts of the GUI hierarchy related to the current input event sequence for simplification. Fig. **1** shows the GUI state hierarchy tree built from a real crash trace from the Android application *DalvikExplorer*.

When building the GUI state hierarchy tree, we use the hierarchical relationship of GUI states to identify the hierarchical relationship of their corresponding events. In this state hierarchy tree, each node with number *i* not only represents the event *i* in the crash trace, but also represents the GUI state before processing the event *i*. An edge in the GUI state hierarchy tree represents the parent-child (hierarchical) relationship between the two corresponding GUI states.

If a new node *n* with a state different from any nodes on the path from the root node to its previous node, then the node *n* is defined as a child node of its previous node; otherwise, the node *n* is defined as a new (and right) sibling node of the equivalent node having the same state as *n*. In Fig. **1**, within the state hierarchy tree, we use the same color to represent equivalent states (sibling) and the last node without any index to stand for state "crash". With the GUI hierarchy tree construction process, new events will always grow at the rightmost sub-tree. Thus, the crash node (i.e., the last event of the event sequence) is always the rightmost node at the bottom level of the rightmost sub-tree of the whole tree. Furthermore, the parent and the ancestors of the crash node are always the ***last*** nodes in their corresponding levels.

[1] The SimplyDroid tool is open source at https://github.com/gongbell/SimplyDroid

We have built an enhanced Monkey tool to log the application GUI states during testing. The definition of GUI state can have different granularities. In the current version of SimplyDroid, we use an Activity ID to represent a GUI state, which is a lightweight solution. Using a finer level of GUI state (e.g., GUI structure) definition may lead to different reduction results, whose tradeoff is interesting to explore in the future.

Therefore, our crash traces contain not only the sequence of input events but also the corresponding GUI states. More specifically, our crash trace of events is in the form $\langle e_1, e_2, \ldots, e_n \rangle$, and it corresponds to the GUI state trace $\langle s_1, s_2, \ldots, s_{n-1}, s_n,$ *crash state*$\rangle$. Upon receiving event $e_i$, the state $s_i$ transits to $s_{i+1}$.

The GUI state hierarchy tree is constructed as follows. For each state $s_i$ and event $e_i$ with the same number *i*, a new node *n* is constructed. Then node *n* is compared with each node (state) *m* from root to its previous node $s_{i-1}$ in turn. If $s_i$ is equal to any node *m*, node *n* is added as a sibling node of *m*. Otherwise, *n* is added as a child node of node $s_{i-1}$. The tree construction algorithm will be presented in Section III.

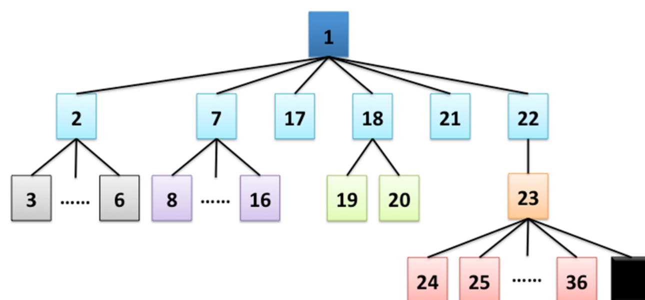

Fig. 1 GUI state hierarchy tree of an exemplified trace

### B. The Reduction Process of Classic DD

The Delta Debugging (DD) algorithm makes no use of the hierarchical relationships of GUI states [27]. It partitions an input event sequence into subsequences with equal size, and performs reduction with increasingly finer granularity. For the exemplified crash trace of length 36, in our experiment, the DD algorithm used 38 trial executions by spending 3 minutes and 15 seconds before getting the final reduced test input sequence ⟨1, 22, 23, 36⟩. We observed that many of these 38 trial executions generated invalid traces, which wasted efforts.

### C. The Reduction Process of HDD

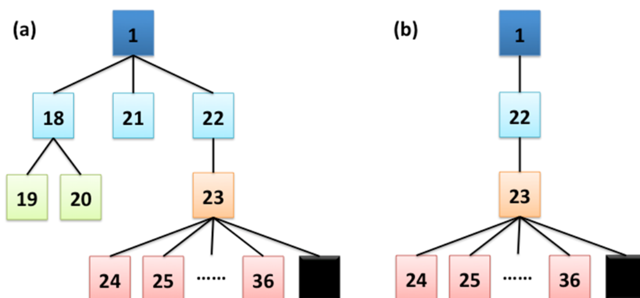

Fig. 2 Intermediate reduction attempts of HDD

The idea of HDD is to perform the reduction from a higher level to a lower level on a GUI state hierarchy tree. In this way, the sub-trees of a node can be reduced together, leading to

higher reduction efficiency. Within each level, the reduction strategy of HDD is similar to DD. Fig. 2 shows the reduction attempts of HDD within level 1. While processing level 1, HDD equally partitions the node set (2, 7, 17, 18, 21, 22) shown in Fig. 1 into two parts (2, 7, 17) and (18, 21, 22). It tries to remove one part or the other. As a result, the result for the first round of reduction is shown in Fig. 2(a). We can see that those nodes (2, 7, 17) and their sub-trees are all eliminated in one round. Then the node set (18, 21, 22) becomes the input for the next iteration of reduction. This node set is again partitioned into two parts (18, 21) and (22) for reduction. The result for the second round of reduction is shown in Fig. 2(b). The reduction at level 1 stops because there is only one node numbered 22 remained. Then the reduction at levels 2 and 3 continue iteratively. Finally, HDD used 6 trial executions by spending 64 seconds to get the same reduction result ⟨1, 22, 23, 36⟩ as DD. We can see from the reduction process that HDD is more efficient than DD.

### D. The Reduction Process of BHDD

BHDD is based on the insight that many GUI state hierarchy trees in real-world scenarios are imbalanced: sub-trees with more nodes tend to stay at one side. These trees may make HDD to spend more execution trails on reducing small node sets. BHDD takes the size of whole sub-tree rooted at each node into consideration, and thus has the potential to reduce more nodes in each round of execution trial than HDD.

As shown in Fig. 3, we label the size of the sub-tree for each node at level 1. The partition strategy of BHDD tries to make the total number of nodes in each partition as close as possible. Thus, the first round of partition for BHDD generates node sets (2, 7, 17, 18) and (21, 22). The former has a total of 19 nodes and the latter has a total of 16 nodes. In this way, for an imbalanced tree, BHDD may reduce more events in each round. For the example trace, BHDD used 43 seconds with 6 trial executions to get the same reduction result as DD.

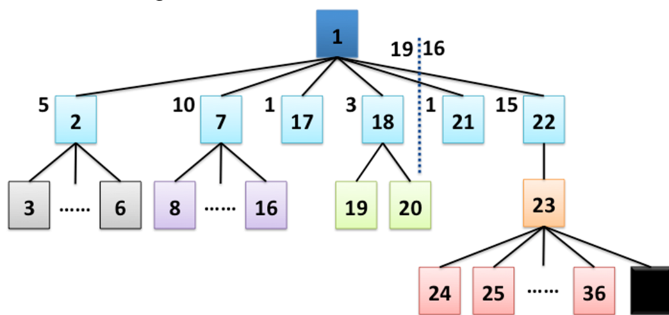

Fig. 3 Partition strategy of BHDD

### E. The Reduction Process of LHDD

As discussed in Section II A, the parent and the ancestors of the crash node are always the ***last*** nodes (e.g., node 22 and 23) in their corresponding levels. We can see from the GUI hierarchy tree of the example that these ancestor nodes of the crash node (e.g., node 22 and 23) are critical because at these nodes, the application under test transits to the next level of GUI state closer to the crash node. Based on this observation, the LHDD algorithm adopts a heuristic: reducing the sequence of events as long as the transition from ***the last node of one level*** to the ***first node of its next level*** is preserved. For the exemplified GUI tree in Fig. 1, we need only to check whether the transition ⟨*22,23*⟩ is successful at (blue) level 1 (Fig. 4(b)) and the transition ⟨*23, 24*⟩ is successful at (orange) level 2. At the bottom level, LHDD checks for crash occurrence.

In this way, we effectively convert the reduction process from a global optimization problem into a "partial" local optimization problem. By "local", we mean that the reduction process needs not check the sub-tree of the last node, and by "partial" we mean that the subtree of nodes other than the last node should still be checked. For example, when reducing at level 1 (blue nodes), LHDD does not include the subtree rooted at node 22 for checking (i.e., local), but includes the subtrees rooted at 2, 17 and 18 for checking (i.e. partial local). In our experience, the subtree of the last node is often large in size. In such scenarios, intuitively, LHDD saves a lot of time.

To further optimize the local reduction process, LHDD adopts another heuristic before performing the DD reduction at each level. Since the reduced event sequence at current level must include the last node, it incrementally selects nodes (events) with number equal to power of 2 from the last node to front for trial execution and stops when it finds the first sub-sequence of events that can still reach the first node in the next level. As shown in Fig. 4(a), the pre-selection process will try event sequence ⟨22⟩, ⟨21, 22⟩, and ⟨17, 18, 19, 20, 21, 22⟩ in turn until the event sequence is accepted by the local reduction criteria. Finally, LHDD used only 16 seconds with 4 trial executions to get the same reduction result, which is much more efficient than the previous three DD algorithms.

## III. Our Efficient Event Trace Reduction Framework

In this section, we first present the overall design of our event trace reduction system. Then we present three test case reduction algorithms: HDD, IHDD, and LHDD.

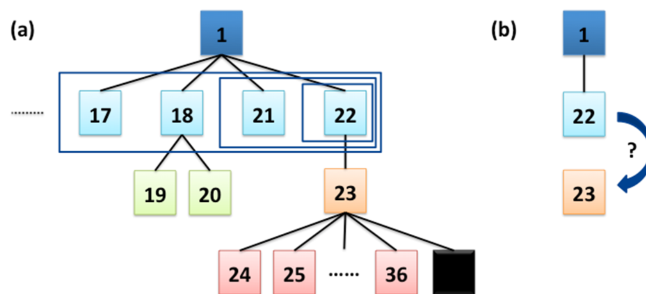

Fig. 4 Pre-selection and local reduction strategy of LHDD

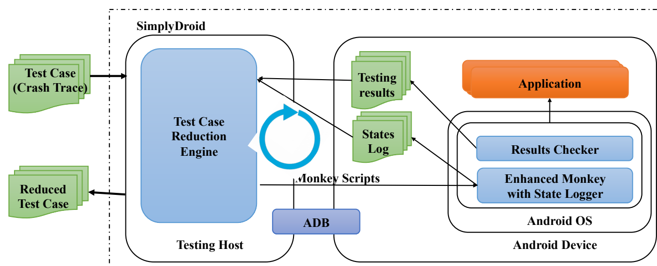

Fig. 5 Design of SimplyDroid

### A. Design of SimplyDroid

The overall design of *SimplyDroid* is shown in Fig. **5**. The input of *SimplyDroid* is the original test case (i.e., the original crash trace), while the output is a finally reduced test case. The *SimplyDroid* tool consists of 3 cooperating modules (colored blue): the test case reduction engine, the result checker, and the enhanced Monkey with application state logger. The test case reduction engine is the core of the SimplyDroid tool, which realizes the sets of test case reduction algorithms.

The reduction process is iterative, where the test case reduction engine repeatedly sends reduced Monkey events to the application under test. The enhanced monkey in turn sends the inputs in the script to the application under test. We also enhanced Monkey to record the state of the application (i.e., Activity ID) upon sending each event. At the end of a test script execution, the result checker checks the results by comparing exception messages and the stack traces outputted by Monkey. Then, the test results and the GUI state log is feedback to the reduction engine to continue the delta debugging cycle. Finally, the reduced test case is outputted.

### B. Building GUI State Hierarchy Tree

As shown in Table 1, the algorithm builds the GUI state hierarchy tree from the crash trace containing the input events and states. As discussed in Section II, the crash trace of states is in the form <$s_1, s_2, \ldots, s_n$, *crash state*> where $s_i$ represents states. Note there is a corresponding events trace <$e_1, e_2, \ldots, e_n$> where each event $e_i$ triggers the transition from state $s_i$ *to* $s_{i+1}$. The algorithm builds a new node for each state $s_i$ (line 1 to 3). Then it is compared with the nodes (states) from its previous node $s_{i-1}$ to the root node (line 4 to 11). If $s_i$ is equal to any node, it is added as a sibling node to it. Otherwise, it is added as a child node of its previous node (line 12 & 13).

TABLE 1. THE GUI HIERARCHY TREE CONSTRUCTION ALGORITHM

```
        procedure build(S)
        Input S : the sequences of activity state logged.
        Output r : the root node of GUI tree built.
        begin
1         n0 = new Node()    //n0 is an empty node
2         for( i : 1 to length(S) ) {
3           ni = new node(si)    //the state of node ni is si
4           ns = ni-1 //start comparison from previous node
5           while( ns != n0 ) {
6             if( ni.state == ns.state ){//find a node with same state as ni
7               set ni as a sibling node of ns
8               break
9             }
10            ns = ns.parent  //walk up the tree,
11          } //end while
12          if( ns == n0 ) { //no node in the path to root have same state
13            set ni as a child node of ni-1
14          }  //end if
15        } //end for
16        return n0 //return the root node
        end
```

### C. Hierarchical Delta Debugging Algorithm (HDD)

As shown in Table 2, the HDD algorithm is a realization of the idea of existing hierarchical delta debugging but operating on our tree-based representation of a trace.

TABLE 2 THE HDD ALGORITHM

```
        procedure hdd(E, r)
        Input E : the sequence of events logged.
        Input r : the root node of the GUI state hierarchy tree.
        Output Er : the simplified event sequence.
        begin
1         M.add( r )  //M is a global vector storing the reduced events
2         nodeseq = empty //nodseq stores the reduced nodes at each level
3         N = empty // N stores the sequence of nodes in next level
4          add to N the child nodes of each node in M
5           do{
6             if (N is not empty ) {
7               inlevel_hdd( N, 2 )  //reduction in current level
8               for(each node ni in N) {//M now contains the reduced events
9                 if( M.contain( ni ) ) { //update nodeseq with M
10                  nodeseq.insert(ni)
11                }
12                else {
13                  remove node ni and its subtree from the tree rooted at r}
14              } //end for
15            } // end if (N is not empty )
              N.clear() //clear N to be empty
16            add to N the child nodes of each node in M //update N
17          }while ( N is not empty );
          //note Er is updated in checkEvents on triggering bug
18        return Er
19      end

        procedure inlevel_hdd(N0, p)
        Input N0 : the sequence of nodes need to simplify at current level
        Input p : the number of partition in this simplification.
        begin
20        if( length( N0 ) < p )
21          return
22        //partition N0 into p subsequences N1, N2, ... Np
23        [N1, N2, ... Np] = partition(N0, p )
24        evtseq = nodes2events( nodeseq )  //map nodes (states) to events
25        for( each node nj in group Np ) { //test Np first
26          evtseq.insert(eventsInSubTree(nj))  //inserts events in subtree
27        } //end for
28        if(checkEvents(evtseq) ) { //execute evtseq for checking
29          M = Np //record current reduction result
30          inlevel_hdd( Np, 2 ) //continue finer reduction
31          return  //successful reduction from Np, return
32          }
33        for( i : p-1 to 1 ) {
             //check the complement of the other p-1 partition
34          evtseq = nodes2events( nodeseq )
35          for( each node nj in N1 to Np except Ni ) {
36            evtseq.insert(eventsInSubTree( nj ) )
37          }
38          if( checkEvents( evtseq ) ) {
39            M = merge( N1 to Np except Ni )
40            inlevel_hdd( M, p-1 )
41            return; //successful reduction from complements
42            }
          } //end for
          //fail at current granularity, reduction at finer granularity
43        inlevel_hdd (Np, min( 2*p, length( Np ) ) )
44        return
45      end

46      procedure checkEvents(E0)
47      Input E0 : the sequence of events need to execute on the application.
48      Output res : whether this sequence can reproduce the crash.
49      begin
50          if  E0 can trigger crash on execution {
51             Er = E0 //update Er on successful triggering crash
52            return true } // end if
53          return false  //Events cannot trigger crash
54      end
```

This algorithm starts by invoking *hdd()*. Owing to one-one correspondences between input events and nodes (states) in the tree, the algorithm simplifies the nodes (states) level by level from top to bottom (line 1 to 17). Within each level, the in-level node simplification procedure ***inlevel_hdd()*** is called (line 7). $M$ is a global vector storing the current successfully reduced events, which is updated in ***inlevel_hdd()*** (line 29 and 39). The ***nodseq*** stores the reduced nodes at current level and is updated by $M$ (line 10). And, $N$ stores the next level of nodes for reduction. The ***inlevel_hdd()*** realizes the idea of hierarchical delta debugging within one level until a simplest node sequence is found. It partitions the event sequences (line 23), checks the last partition, and performs recursive reduction if successful (line 30). Otherwise, it tries to reduce the complement of other partitions (line 33 to 41) with recursive calls. If it still fails, it performs finer level reduction (line 44). The procedure ***checkEvents()*** is responsible for execute event sequences, check whether the crash is triggered and update $E_r$, the final simplified event sequence (line 46 to 54).

There are three subroutines whose implementation is omitted for brevity, which we explain as follows: (1) *partition()* (line 23), whose function is to partition node sequence into subsequences with equal number of nodes. (2) *nodes2events()* (line 24, 34) maps the sequence of nodes to the corresponding sequence of events. (3) *eventsInSubTree()* (line 26, 36), whose function is to return all events in the subtree of a node.

### D. Balanced Hierarchical Delta Debugging Algorithm (BHDD)

The algorithm of BHDD optimizes the HDD algorithm by changing the subroutine *partition()* (i.e., line 23 of HDD algorithm). The *partition()* of BHDD is to divide node sequence in the current level into partitions with equal number of nodes by counting the number of nodes in their sub-trees. We omit its detailed implementation for brevity.

### E. Layed Hierarchical Delta Debugging Algorithm (LHDD)

As shown in Table 3, the algorithm ***lhdd()*** is based on ***hdd()*** except with two optimizations. The first optimization is to add a process of pre-selection (line 7 to 22), within which more nodes are selected until the selected sequence of events that can trigger a transition to next level of Activity is found. The second optimization is to only checks the successful transition from the last node in the **local** level to the first node in the next level of Activity in ***checkEvents***(). This saves the execution of the events in the subtree of the last node.

The procedure ***inlevel_hdd()*** of LHDD is the same as that of HDD algorithm, so we omit it for brevity. However, there are some procedures called in ***inlevel_hdd()*** that is changed for LHDD algorithm, which we detail them here. First, the function ***eventsInSubTree()*** for LHDD is the same as that for HDD except that if a node is the last node, it adds only the node itself without adding events in its subtree. Second, the function ***partition()*** for LHDD is the same as that of BHDD by considering of number of nodes of the subtrees, except that the size of the last node in $N$ is counted as 1. The difference in these two procedures essentially reflects the local reduction logic: the sub-tree of the last node in the current level is not considered during reduction.

TABLE 3 THE LHDD ALGORITHM

```
     procedure lhdd(E, r)
     Input E : the sequence of events logged.
     Input r : the root node of the GUI state hierarchy tree.
     Output E_r : the simplified event sequence.
     begin
1      M.add(r)   //M is a global vector storing the reduced events
2      nodeseq = empty //nodseq stores the reduced nodes at each level
3      N = empty // N stores the sequence of nodes in next level
4      add to N the child nodes of each node in M
5       do{
6          if (N is not empty) {
7           preLen = 1  //perform preselection, start at 1
8            // get the current preselected node sequence N_s
9           N_s = N[length(N) – preLen, length(N) – 1]
10          while( preLen < length(N) ) {
11             evtseq = nodes2events ( nodeseq )   // to event sequence
12             for( each node n_i in group N_s ) {
13                //add sub-tree of selected nodes except the last node
14                evtseq.insert(eventsInSubTree( n_i ) )
                } //end for
15             if( checkEvents( evtseq ) ) {//preselection is successful
16                 break;  //exit preselection
17               }
18             else { //try pre-selection with double size
19                preLen = min( 2* preLen, length(N) )
20                N_s = N[length(N) – preLen, length(N) – 1]
21             }
22          } // end of while, preselections stops
23         inlevel_hdd( N_s, 2 ) //performing in level reduction
24         for( each node n_i in N ) {
25           if( M.contain( n_i ) ) { //update nodseq with M
26              nodeseq.insert( n_i )
27             }
28             else {
29              remove node n and its subtree from GUI tree rooted at r
30             }
31          } //end for
32        } //end if (N is not empty)
33       N.clear() //clear N to be empty
34        add to N the child nodes of each node in M //update N
35     }while( N is not empty ); //reduce at all levels
36     //E_r is updated in checkEvents on triggering bug
37     return E_r
38   end

40   procedure checkEvents(E_0)
41   Input E_0 : the sequence of event need to execute in the application.
42   Output res : whether this sequence can trigger the crash at last level
     or transit to next activity successfully at middle level
43   begin
44     if( E_0 contains the last event of E ) {  //last level, testing for crash
45       if  E_0 can trigger crash {
46           E_r = E_0 //update E_r with the sequence E_0
47           return true;
48         }
49       }
50     else  { //not last level, testing for transition
51          If  E_0 can transit to the first node in the next level of Activity
52             return true; }
53    return false; //events can neither trigger crash nor transition
54   end
```

Finally, the procedure ***checkEvents()*** called in both ***lhdd()*** *and* ***inlevel_hdd()*** is also changed (line 40 to 54), which differentiates local check at intermediate levels and the final check in the last level. For the local check, the algorithm only checks the successful transition to the next Activity. For the final check in the last level, the algorithm checks whether a crash is triggered.

TABLE 4 SUBJECT PROGRAMS

| Subject | Subject Description | Program Version | Android Version | Fault Type | Device | Exception Type | Number of crash traces | Crash Trace Length |
|---|---|---|---|---|---|---|---|---|
| Yahtzee | Game | 1.1 | 2.3.3 | Mutant | Mi5 | ArithmeticException | 17 | 100-214 |
| K9mail | Mail | 5 | 2.3.3 | Mutant | Mi5 | NullPointerException | 23 | 150-621 |
| DalvikExplorer | System information Viewer | 3.4 | 4.1.2 | Real | Mi1 | ActivityNotFound & OutOfMemoryError | 11 | 36-1525 |
| WeightChart | Weight recorder | 1.0.4 | 2.3.3 | Real | Mi1 | ActivityNotFound | 8 | 16-765 |
| Ringdroid | Ringtone editor | 2.6 | 4.1.2 | Real | Mi1 | RuntimeException | 6 | 314-885 |
| Tippy | Calculator | 1.1.3 | 2.3.3 | Mutant | Mi5 | ArithmeticException | 9 | 31-162 |
| SyncMyPic | Photo synchronizer | 0.15 | 2.3.3 | Mutant | Mi5 | ArithmeticException | 8 | 27-462 |
| WhoHasMyStuff | Item lending helper | 1.0.7 | 4.4.2 | Mutant | Mi5 | RuntimeException | 10 | 50-679 |

## IV. EXPERIMENT AND RESULTS ANALYSIS

This section presents our experiment and data analysis.

### A. Research Questions

RQ1: Are HDD, BHDD, and LHDD effective to reduce the size of Android input event sequence?

RQ2: Are HDD, BHDD, and LHDD efficient when performing test case reduction?

RQ3: If LHDD is efficient, which of the two optimizations contributes more to its performance improvement.

### B. Experimental Setup

We used a Lenovo laptop V4400 as our testing host. The laptop was equipped with Intel i7 4702 and 16GB memory. The operating system was Windows 10 and the Integrated Development Environment was Eclipse. We used the Monkey tool as our testing engine. On the phone model Mi1 the interval between sending two events was set as 500ms and on Mi5 the interval was set as 800ms to make sure that there was enough time for each event to be processed.

### C. Subject Program and Crash Traces

We have selected 8 real-life subject programs for our experimental study. The descriptive statistics of the 8 subject programs is summarized in Table 4. We have listed the program name, a brief description of the application, the application version, the Android OS version, the type of fault within the application, the device hardware model used to run the tests, the types of exceptions captured upon crash, the number of crash traces for each application, and the range of the length of the traces in the table. For example, *Yahtzee* was a mobile game application whose program version is *1.1*. The application was running on Mi5 phone with Android OS version 2.3.3. We injected mutant faults within the application to generate 17 crashes of type *ArithmeticException*, and the number of events within the crash trace ranges from 100 to 214. The other subjects can be interpreted similarly.

All the 8 subjects were all real-life mobile application that had evolved for many years. Furthermore, the released versions were often quite stable while unstable commit versions were often hard to acquire. As a result, finding crash traces in application releases with Monkey was not an easy task. For DalvikExplorer, WeightChart, and Ringdroid, we were lucky enough to find 25 crash traces with real faults. For the other 5 subjects, we had to manually create mutants by injecting faults frequently found in mobile applications. We inserted, removed or modified statements to inject faults. We got another 95 crash traces from the mutant programs.

Our algorithm only worked on deterministic traces, so we examined these 120 crash traces to check whether they can replay stably. We manually executed each crash trace 3 times. Then we compared their exception messages and stack traces in the output to confirm reproducibility. We removed 19 traces, which cannot be replayed by Monkey stably due to non-determinism in its execution. We removed a subject called *Sanity* and its 7 crash traces because replaying it stably requires resetting its data before each execution, which was not supported by the current implementation of SimplyDroid. We removed 2 versions of *WhoHasMyStuff* because the interaction with the application may sometimes activate the soft input keyboard, which changed the layout of the GUI. As a result, the coordinate-sensitive Monkey cannot replay them. Finally, the 92 crash traces left were all used in our data analysis.

To facilitate identification of the GUI state hierarchy, we had also enhanced Monkey tool to log the application GUI states (i.e., Activity ID in this work) during testing. Therefore, our crash traces contained not only the sequence of input events but also their corresponding GUI states.

### D. Experimental Procedure

We also realized the classic Delta-Debugging (DD) algorithm as a benchmarking technique for comparison. We performed test case reduction on all the 92 crash traces with DD, HDD, BHDD, and LHDD algorithms. We logged the size of the reduced test cases as well as the time for reduction for each technique on each trace. Since there are two major optimizations in the LHDD algorithm (i.e,, the pre-selection and the local reduction), we wonder which optimization contributes more in its performance improvement. So we turned off the pre-selection optimization in the LHDD (i.e., using the local reduction optimization only) to form a new technique LHDD-NoPre for comparison with LHDD. We also used LHDD-NoPre to perform test case reduction on the 92 crash traces and log their reduction results.

### E. Experimental Results and Analysis

In this section, we present our experimentation results followed by detailed results analysis for each research question.

*1) Answering RQ1:* In this section, we would like to know whether our proposed HDD techniques are effective to reduce the size of Android input event sequence.

TABLE 5 LENGTH OF CRASH TRACE (TEST CASE) AFTER REDUCTION

| Subjects | Yahtzee | | | | | | | | | | | | | | | | | K9mail | | | | | | | | | | | | | | | | | | | | | | | DalvikExplorer | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 |
| Original | 168 | 196 | 153 | 214 | 131 | 100 | 101 | 110 | 121 | 121 | 126 | 128 | 111 | 134 | 123 | 107 | 110 | 171 | 156 | 587 | 216 | 621 | 269 | 184 | 193 | 275 | 216 | 366 | 266 | 217 | 261 | 272 | 215 | 220 | 202 | 154 | 216 | 150 | 269 | 180 | 36 | 75 | 117 | 307 | 950 | 1525 |
| DD | 8 | 7 | 7 | 8 | 7 | 7 | 9 | 7 | 7 | 7 | 9 | 7 | 8 | 7 | 7 | 7 | 7 | 6 | 5 | 7 | 4 | 4 | 4 | 4 | 7 | 5 | 4 | 5 | 5 | 8 | 5 | 5 | 6 | 5 | 4 | 4 | 5 | 7 | 8 | 5 | 4 | 4 | 4 | 4 | 4 | 4 |
| HDD | 7 | 9 | 10 | 9 | 8 | 8 | 8 | 7 | 8 | 9 | 7 | 10 | 9 | 9 | 7 | 10 | 7 | 10 | 6 | 5 | 7 | 5 | 5 | 4 | 8 | 6 | 6 | 7 | 5 | 9 | 5 | 8 | 6 | 4 | 8 | 4 | 6 | 4 | 10 | 7 | 4 | 4 | 4 | 4 | 4 | 4 |
| BHDD | 8 | 9 | 10 | 8 | 8 | 8 | 7 | 7 | 8 | 8 | 8 | 10 | 11 | 10 | 8 | 8 | 8 | 7 | 5 | 5 | 7 | 5 | 5 | 4 | 12 | 5 | 7 | 6 | 4 | 11 | 6 | 6 | 6 | 4 | 5 | 5 | 5 | 6 | 9 | 10 | 4 | 4 | 4 | 4 | 4 | 4 |
| LHDD | 7 | 8 | 7 | 8 | 7 | 8 | 7 | 7 | 8 | 8 | 8 | 7 | 8 | 7 | 7 | 9 | 8 | 7 | 7 | 6 | 6 | 9 | 6 | 4 | 6 | 6 | 5 | 17 | 6 | 11 | 10 | 9 | 6 | 5 | 5 | 5 | 9 | 6 | 7 | 13 | 4 | 4 | 4 | 4 | 4 | 4 |
| LHDD -NoPre | 7 | 8 | 7 | 8 | 7 | 9 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 10 | 9 | 5 | 6 | 8 | 7 | 6 | 6 | 6 | 4 | 15 | 13 | 12 | 8 | 8 | 5 | 4 | 4 | 4 | 11 | 4 | 7 | 12 | 4 | 4 | 4 | 4 | 4 | 4 |

| Subjects | DalvikExplorer | | | | | WeightChart | | | | | | | | Ringdroid | | | | | | Tippy | | | | | | | | | SyncMyPic | | | | | | | | WhoHasMyStuff | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 |
| Original | 1359 | 397 | 404 | 265 | 294 | 16 | 46 | 521 | 514 | 239 | 765 | 448 | 220 | 885 | 449 | 550 | 314 | 646 | 706 | 70 | 76 | 63 | 31 | 102 | 129 | 162 | 46 | 99 | 45 | 426 | 89 | 154 | 177 | 462 | 27 | 44 | 132 | 78 | 306 | 353 | 472 | 50 | 217 | 679 | 149 | 231 |
| DD | 4 | 4 | 9 | 4 | 4 | 9 | 5 | 5 | 9 | 5 | 5 | 5 | 5 | 8 | 8 | 7 | 7 | 7 | 7 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 2 | 4 | 4 | 2 | 6 | 6 | 5 | 6 | 6 | 5 | 6 | 6 | 5 | 6 |
| HDD | 4 | 4 | 11 | 4 | 4 | 7 | 9 | 8 | 8 | 7 | 8 | 19 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 4 | 4 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 4 | 4 | 6 | 9 | 4 | 3 | 6 | 6 | 5 | 7 | 6 | 5 | 6 | 7 | 5 | 6 |
| BHDD | 4 | 4 | 11 | 4 | 4 | 9 | 5 | 8 | 8 | 7 | 8 | 18 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 4 | 4 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 4 | 4 | 6 | 9 | 4 | 3 | 6 | 6 | 5 | 6 | 6 | 5 | 6 | 7 | 5 | 6 |
| LHDD | 4 | 4 | 13 | 4 | 4 | 5 | 6 | 8 | 9 | 7 | 8 | 14 | 9 | 7 | 7 | 7 | 6 | 8 | 7 | 4 | 4 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 4 | 4 | 5 | 9 | 4 | 3 | 6 | 6 | 5 | 6 | 6 | 5 | 6 | 7 | 5 | 6 |
| LHDD -NoPre | 5 | 4 | 12 | 4 | 4 | 5 | 7 | 7 | 11 | 10 | 14 | 21 | 10 | 7 | 7 | 7 | 6 | 7 | 7 | 4 | 4 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 4 | 4 | 6 | 8 | 4 | 3 | 6 | 6 | 5 | 6 | 6 | 5 | 6 | 7 | 5 | 6 |

We have shown the length of the crash traces after reduction by each technique in Table 5. For each trace, we show its number, the original length of the crash trace before reduction and the length of the reduced crash trace for DD, HDD, BHDD, and LHDD, each in one row. Since there are too many traces (92 in total) to be shown, we have to wrap around the results at number 47 within the subject *DalvikExplorer*. We also listed the results of LHDD without pre-selection (LHDD-NoPre) in the last row for comparison purpose, which we will discuss when answering RQ3.

When compared with the original crash trace length, all the techniques achieve significant test case reduction rates. Except for a few outlier cases, most of these techniques achieve reduction rates of more than 90%. Furthermore, the actual length of crash traces after reduction is small for most of the cases. Thus we can answer RQ1 that the HDD techniques are effective to reduce the size of crash.

There are 3 crash traces (No. 28 and No. 40 of K9mail, No. 58 of WeightChart, in grey) where the family of HDD techniques generate larger reduced test case than DD technique. We have inspected these 3 crash traces carefully and found that this is due to the imprecise logging of application states in our enhanced Monkey. For K9mail, a popup window interferes with the current Activity; for WeightChart, a System Activity misleads our logger. We will improve our state logger in future work.

Despite these exceptional cases, the family of HDD techniques (HDD, BHDD, LHDD) in general has statistically comparable reduction effectiveness as the DD technique. We have performed *ANOVA* test to check whether the 4 different techniques (DD, HDD, BHDD, LHDD) have significant difference from each other. The *ANOVA* return a p-value more than 0.1, which cannot reject the null hypothesis at 0.05 significance level. Therefore, there is no significant difference among these 4 techniques in terms of reduction effectiveness.

*2) Answering RQ2:* In this section, we would like to know whether the family of HDD techniques is efficient such that developers need not to wait for a prolonged period to get a reduced test case.

The test case reduction time for the 8 subject programs are shown in Fig. **6** to Fig. **13**. Within each figure, the x-axis shows the crash traces number and the y-axis shows the test case reduction time in seconds. For some programs, the difference in reduction time between different crash traces is so big that we have to show the results in the log scale for y-axis.

We can see from the plots that in general the LHDD algorithm performs the best while the DD algorithm performs the worst. In fact, for majority of crash traces, the time saving of the LHDD technique compared to the DD technique is significant. For example, for crash trace No. 5 of *WeightChart*, DD takes 10 hours (36223 seconds) whereas LHDD takes only less than 7 minutes (413 seconds). When averaged over the 92 crash traces, HDD, BHDD, and LHDD save 4.6, 9.8, and 16 minutes over DD per trace, respectively. In total, HDD, BHDD, and LHDD save around 7, 15 and 25 hours of debugging time over DD on all 92 crash traces, respectively.

TABLE 6 SUMMARY OF TEST CASE REDUCTION EFFICIENCY

| > | DD | | HDD | | BHDD | |
|---|---|---|---|---|---|---|
| | Count | Percentage | Count | Percentage | Count | Percentage |
| HDD | 55 | 60% | \ | \ | \ | \ |
| BHDD | 75 | 81% | 71 | 77% | \ | \ |
| LHDD | 86 | 93% | 84 | 91% | 74 | 80% |

The reduction efficiency from DD to HDD and then to BHDD and finally LHDD appears to increase gradually. For each pair of techniques, we have computed the number and percentage of crash traces where the former uses less test case reduction time (in other words, better ">") than the later as shown in Table 6. HDD is more efficient than DD on 55 out of 92 cases. The percentage of crash traces where BHDD is more efficient than DD and HDD are 81% and 77%, respectively. LHDD is more efficient than DD, HDD, and BHDD on 93%, 91%, and 80% of the crash traces, respectively.

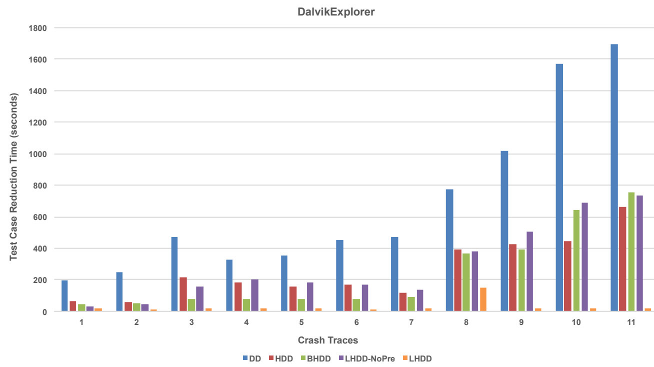


Fig. 6 Test Case Reduction Time for DalvikExplorer

Fig. 7 Test Case Reduction Time for WeightChart

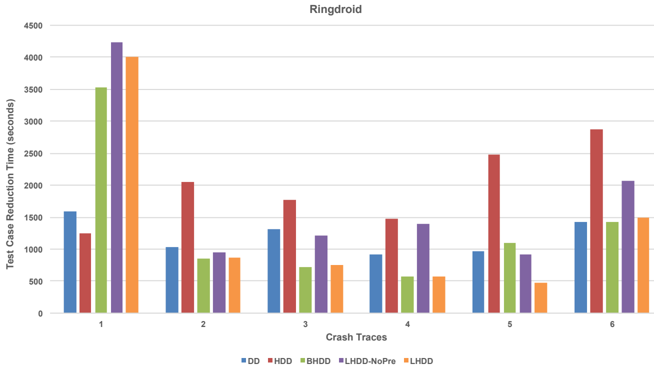


Fig. 8 Test Case Reduction Time for RingDroid

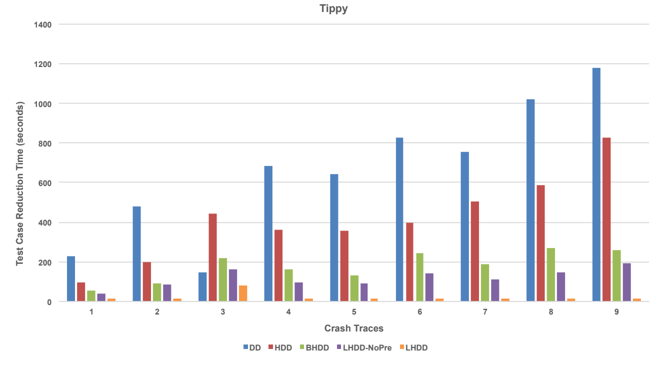


Fig. 9 Test Case Reduction Time for Tippy

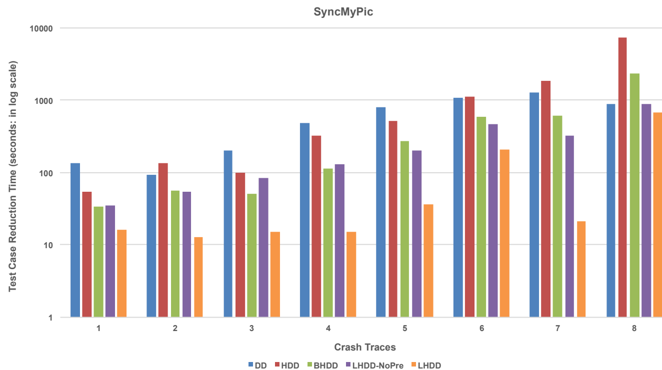


Fig. 10 Test Case Reduction Time for SyncMyPic

Fig. 11 Test Case Reduction Time for WhoHasMyStuff

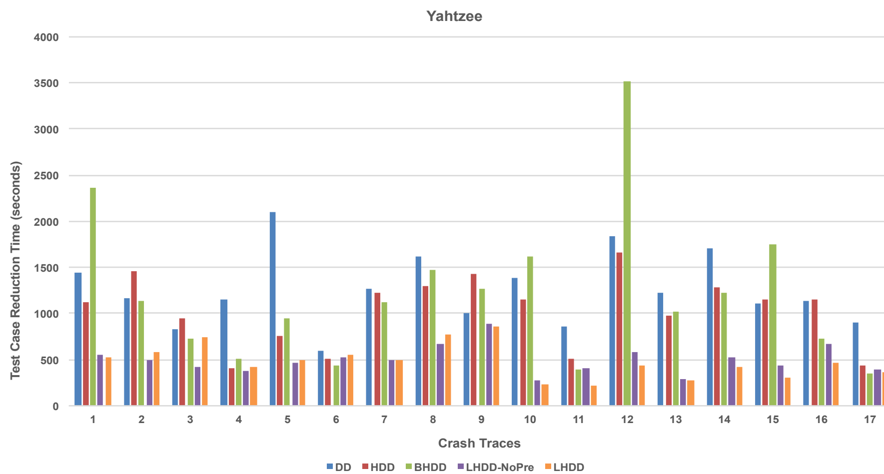


Fig. 12 Test Case Reduction Time for Yahtzee

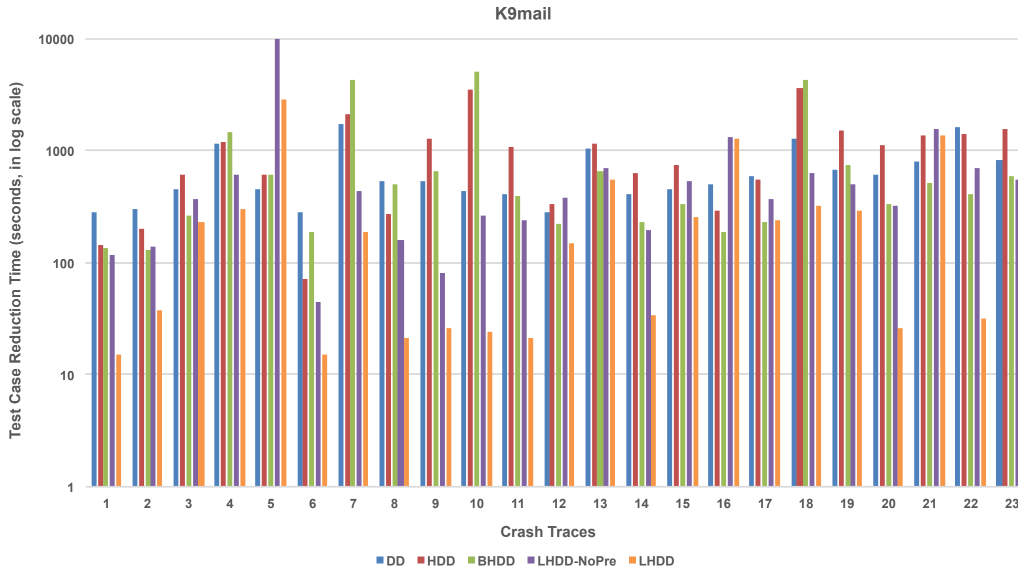


Fig. 13 Test Case Reduction Time for K9mail

There are a few exception cases where LHDD is slower to perform reduction than DD. We have examined those cases to understand the reasons behind. For example, the crash trace No. 1 and No. 6 of Ringdroid optionally popped up an AlertDialog, which hided the underlying Activity. This situation made our Monkey tool unable to log the correct GUI states. Similarly, the crash trace No. 5 and No. 16 of K9mail popped ups floating windows during execution, which made our Monkey tool unable to log the same GUI states each time. This in turn disrupted the reduction process of LHDD.

Despite those exceptional cases, we can see from our results that this family of HDD techniques increasingly improves the efficiency of DD (where LHDD > BHDD > HDD > DD) without significant loss of reduction effectiveness.

*3) Answering RQ3:* For our LHDD algorithm, there are two optimizations to improve its performance. In this section, we would like to find out how much each optimization has contributed to the overall optimization. To perform the comparison, in the experiment, we turned off the pre-selection optimization in LHDD so that we can see the impact of each factor with comparison.

In Table 5, the rows of *LHDD-NoPre* show the reduction results of LHDD without pre-selection. We can see that in general, the reduction effectiveness of LHDD-NoPre is close to LHDD. A further ANOVA analysis also confirms that there is no significant difference between LHDD-NoPre and LHDD in terms of the size of reduced test case.

However, as shown in Fig. **6** to Fig. **13**, the test case reduction time of LHDD-NoPre is larger than LHDD on most crash traces. We further count the number of crash traces where LHDD performs better than LHDD-NoPre. We found that there are 83 out of the 92 (i.e., 90%) crash traces on which LHDD performs better. Furthermore, if we compare LHDD-NoPre with its base technique BHDD (note their ***partition()*** routine is the same), there are 60 out of the 92 (i.e., 65%) crash traces on which LHDD-NoPre outperforms. This result shows the performance of LHDD-NoPre lies in between BHDD and LHDD. Therefore, we can answer RQ3 that both optimizations (local reduction and pre-selection) contribute to the performance improvement of LHDD.

### F. *Threats to Validity*

We focused on improving the efficiency of test case reduction technique rather than dealing with the impact of non-determinism in execution. Therefore, we selected stable crash traces to perform test case reduction. We leave the handling of non-deterministic crash traces as a future work.

The definition of state equivalence also has an impact on the test case reduction. Currently, our trace representation uses the Activity ID to represent the GUI state, which is a lightweight solution. Using a finer level of GUI state equivalence to build the GUI hierarchy tree may lead to different test case reduction results. However, the logging of state will also become expensive. We will leave the exploration of this trade-off in future work.

We use Java to realize our SimplyDroid tool. We have carefully performed code review and testing on our implementation to ensure their correctness. We used 8 subjects and 92 crash traces to evaluate our SimplyDroid platform. Additional studies on more real-life crash traces will further strengthen the validity of our study.

In our experiment, we removed *Sanity* and its 7 crash traces because replaying it stably requires resetting its data before each execution. To enhance the applicability of SimplyDroid, we may perform reinstallation of the application or reload a system snapshot with the emulator before each execution.

## V. Related Work

In this section, we will briefly review related works, including debugging techniques, Android testing techniques, and test case reduction techniques.

### A. Debugging Techniques

Fault localization is a time consuming. Program slicing [11][14][23][28] can confine the search range to a program slice instead of the entire program. A program slice can be obtained statically [24] or dynamically [1]. Spectrum-based fault localization correlates failures against problem spectra [13][21][22], which provides correlation information on program entities for users to review.

There are other techniques for fault localization, such as statistical methods to locate bugs with instrumented predicates [16], state-based methods that use variables and values, and even machine learning-based techniques that train neural networks to construct models for fault localization [4][5].

### B. Android Application Testing Technique

There are also many techniques proposed for Android application testing.

Model based testing techniques first build a model for the application under test and then generate test cases based on the model [10]. Amalfitano et al. [2] proposed a crawler-based technique to build a GUI model and generate input events. PUMA [12] is a dynamic analysis framework and test case generation tool with generic design. Choi et al [7] proposed the SwiftHand tool to minimize the number of application restart during testing. Azim et al. [3] proposed the A3E tool with two complementary GUI traversal strategies: A3E-Depth-First and A3E-Targeted.

Monkey is a well-known Android application random testing tool developed by Google and has been widely adopted in stress testing and reliability testing [40]. Dynodroid is another automated testing tool with strategy similar to Monkey, yet with more sophisticated strategies [17]. Sapienz is a multi-objective and search-based Android application-testing tool whose optimization goal includes both code coverage rate and fault detection rate [18].

There are also several test development platforms, such as MonkeyRunner [36], Robotium [37], and UIAutoMator [29], for developers to write customized test scripts.

### C. Test Case Reduction

The goal of test case reduction or minimization is to find a minimal subset of inputs that can still produce the same failure.

Delta Debugging [25] is a method to automate the debugging of programs using the hypothesis-trial-result loop [32]. Simplification and isolation are two major algorithms of Delta Debugging [27]. DDMIN is direct realization of simplification algorithm [27], whose goal is to find a minimal subset of inputs that each element of it is necessary for reproducing the failure so that it cannot be removed.

For test case reduction of Android application, Clapp et al. [9] proposed the Non-Deterministic Delta Debugging Minimization (ND3MIN) algorithm. ND3MIN improved the DDMIN algorithm by handling the non-deterministic execution of Android input event sequence. It adopts a probability approach by executing each candidate reduction many times and only considers this candidate successful if its success rate exceeds a threshold. Our SimplyDroid tool differs from ND3MIN in two aspects. First, they have different reduction goals. SimplyDroid tries to reproduce a crash or failure while ND3MIN tries to find a minimal event trace that can reach a given target Activity. Second, SimplyDroid mainly addresses the problem of test case reduction efficiency, whereas ND3MIN mainly addresses the problem of non-deterministic execution.

### D. Simplification of Test Case with Structures

DDMIN is based on test cases of string input so that the minimal independent element of simplification is a single character. Based on DDMIN, the Berkeley Delta algorithm [31] uses a line of text as the basic unit in reduction. C-Reducer can simplify large C programs [30]. It shares the same basic idea with Delta Debugging, but uses a modularized strategy of deletion and simplification based on the grammar tree representation of the program source code [20].

Apart from C/C++ programs, there are also many test cases artifacts with explicit hierarchical structure characteristics such as HTML and XML files. Misherghi and Su proposed the original Hierarchical Delta Debugging algorithm to realize the simplification of C/C++ programs and HTML/XML files [19]. Utilizing the hierarchical structure information of the test input, their hierarchical delta debugging algorithm can be much more efficient than the standard DD algorithm.

## VI. Conclusion and Future Work

There are many automated test case generation techniques to ensure the quality of Android applications. The Monkey fuzz testing tool and its improved versions are simple, effective and widely adopted in the industry. The test cases generated by Monkey often contain a large number of input events, which are difficult to be used by developers in debugging. It is desirable to simplify the input event sequence as small as possible while triggering the same failure. However, the traditional delta debugging technique is slow to perform such simplification. We observe that the events within a failure-inducing trace have hierarchical relationships in the form of user interaction sessions that can be reduced together with high probability. In this work, we have proposed SimplyDroid, a crash trace simplification tool for Android applications. We have proposed a novel GUI state hierarchy tree as trace representation and a family of 3 hierarchical delta debugging algorithms to operate on this trace representation. Our experiments on 92 crash input traces on eight real-life Android applications show that techniques in this family are increasingly more efficient to perform test input reduction.

For future work, we will further study the non-deterministic execution problem in the context test case reduction. We will also perform empirical study to explore the impact of state equivalence on test case reduction.

## References


[1] H. Agrawal, J. R. Horgan. Dynamic Program Slicing. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp. 246-256, White Plains, New York, 1990.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon, MobiGUITAR – a tool for automated model-based testing of mobile apps. IEEE Software, 32(5):1-1, 2014.

[3] T. Azim, 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 (OOPSLA2013), New York, NY, USA: ACM, pp. 641–660, 2013.

[4] L.C. Briand, Y. Labiche, X. Liu. Using Machine Learning to Support Debugging with Tarantula. In Proceedings of the 18th IEEE International Symposium on Software Reliability, pp. 137-146, Trollhattan, Sweden, 2007.

[5] Y. Brun, M. D. Ernst. Finding Latent Code Errors via Machine Learning over Program Executions. In Proceedings of the 26th International Conference on Software Engineering, pp. 480- 490, Edinburgh, UK, 2004.

[6] Y. Chen. Improving the Utility of Compiler Fuzzers, Utah, USA: The University of Utah, 2013.

[7] W. Choi, G. Necula, 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 (OOPSLA2013), New York, NY, USA: ACM, 2013, pp. 623–640, 2013.

[8] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated Test Input Generation for Android: Are We There Yet? (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15). IEEE Computer Society, Washington, DC, USA, 429-440, 2015. DOI=http://dx.doi.org/10.1109/ASE.2015.89

[9] L. Clapp, O. Bastani, S. Anand, A. Aiken. Minimizing GUI event traces. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). ACM, New York, NY, USA, 422-434. 2016.

[10] M. C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, M.I. Schwartzbach (Eds.), tapsoft '95: Theory and Practice of Software Development. Lecture Notes in Computer Science, number 915, Springer-Verlag, Heidelberg, pp. 82-96, 1995.

[11] T. Gyimothy, A. Beszedes, I. Forgacs. An Efficient Relevant Slicing Method for Debugging. In Proceedings of 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 303-321, Toulouse, France, September 1999.

[12] S. Hao, B. Liu, S. Nath, W. G. Halfond, R. Govindan, PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys2014), New York, NY, USA: ACM, pp. 204–217, 2014.

[13] J. A. Jones, M. J. Harrold. Empirical Evaluation of the Tarantula Automatic FaultLocalization Technique. In Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering, pp. 273-282, Long Beach, California, USA, 2005.

[14] B. Korel. PELAS – Program Error-Locating Assistant System. IEEE Transactions on Software Engineering, 14(9):1253-1260, 1988.

[15] B. Korel, S. Yalamanchili. Forward Computation of Dynamic Program Slices. In Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 66-79, Seattle, Washington, 1994.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan. Scalable Statistical Bug Isolation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 15-26, Chicago, Illinois, USA, 2005.

[17] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: An Input Generation System for Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), New York, NY, USA: ACM, pp. 224–234, 2013.

[18] K. Mao, M. Harman, Y. Jia. Sapienz: multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA2016), ACM, New York, NY, USA, pp. 94-105, 2016.

[19] G. Misherghi, Z. Su. HDD: Hierarchical Delta Debugging. In Proceedings of the 28th International Conference on Software Engineering. Shanghai, China: ACM, pp. 142-151, 2006.

[20] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. In Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation, Beijing, China: ACM, pp. 335-346, 2012.

[21] M. Renieris, S. P. Reiss. Fault Localization with Nearest Neighbor Queries. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp. 30-39, Montreal, Canada, October 2003.

[22] T. Reps, T. Ball, M. Das, J. Larus. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In Proceedings of the 6th European Software Engineering Conference, pp. 432-449, Zurich, Switzerland, September, 1997.

[23] M. Weiser. Programmers use Slices when Debugging. Communications of the ACM, 25(7):446-452, 1982.

[24] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, SE-10(4):352357, 1984.

[25] A. Zeller. Yesterday, my program worked. Today, it does not. Why?. In ACM SIGSOFT Software Engineering Notes, pp. 253-267, Volume 24 Issue 6, 1999.

[26] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 1-10, Charleston, South Carolina, USA, 2002.

[27] A. Zeller, R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering, 28(2): 183-200, 2002.

[28] X. Zhang, R. Gupta, Y. Zhang. Precise Dynamic Slicing Algorithms. In Proceedings of the 25th IEEE International Conference on Software Engineering, pp. 319-329, Portland, Oregon, USA, 2003.

[29] Android uiautomator. http://developer.android.com/tools/help/uiautomator/index.html.

[30] C-Reduce. http://embed.cs.utah.edu/creduce/, last access, 2017.

[31] Delta. http://delta.tigris.org/, last access, 2017.

[32] Delta Debugging, https://en.wikipedia.org/wiki/Delta_Debugging, Last access, 2017.

[33] Garnter. Worldwide smartphone sales to end users by operating system in 3Q16. http://www.gartner.com/newsrboom/id/3516317.

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

[35] Mobile Testing Center of Baidu. http://mtc.baidu.com, last access, 2017.

[36] Monkeyrunner. http://www.android-doc.com/tools/help/monkeyrunner_concepts.html.

[37] Robotium. https://github.com/RobotiumTech/robotium.

[38] Statista. Number of available applications in the Google Play Store from December 2009 to December 2016. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[39] Testin. www.Testin.cn, last access, 2017.

[40] The Monkey UI android testing tool. http://developer.android.com/tools/help/monkey.html.

[41] UTEST platform of Tencent. http://utest.qq.com, last access, 2017.