ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations

Jue Wang
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
juewang591@gmail.com

Yanyan Jiang
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
changxu@nju.edu.cn

Chun Cao
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
caochun@nju.edu.cn

Xiaoxing Ma
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
xxm@nju.edu.cn

Jian Lu
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
lj@nju.edu.cn

ABSTRACT

Android apps demand high-quality test inputs, whose generation remains an open challenge. Existing techniques fall short on exploring complex app functionalities reachable only by a long, meaningful, and effective test input. Observing that such test inputs can usually be decomposed into relatively independent short use cases, this paper presents ComboDroid, a fundamentally different Android app testing framework. ComboDroid obtains use cases for manifesting a specific app functionality (either manually provided or automatically extracted), and systematically enumerates the combinations of use cases, yielding high-quality test inputs.

The evaluation results of ComboDroid on real-world apps are encouraging. Our fully automatic variant outperformed the best existing technique APE by covering 4.6% more code (APE only outperformed Monkey by 2.1%), and revealed four previously unknown bugs in extensively tested subjects. Our semi-automatic variant boosts the manual use cases obtained with little manual labor, achieving a comparable coverage (only 3.2% less) with a white-box human testing expert.

KEYWORDS

Software testing, mobile apps

ACM Reference Format:
Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380382

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380382

1 INTRODUCTION

Android apps are oftentimes inadequately tested due to the lack of *high-quality test inputs*[1] to thoroughly exercise an app's functionalities and manifest potential bugs [11]. Existing automatic testing techniques fall short on exploring complex app functionalities that are only reachable by long and "meaningful" event sequences [33, 59]. Random or heuristic test input generation techniques [5, 6, 10, 28, 41–43, 56] can quickly cover superficial app functionalities, but have difficulty in reaching deeper app states to cover complex ones. Systematic input space exploration techniques [7, 47, 48, 61, 65] have severe scalability issues. Manual testing is effective and thorough, but also tedious, labor-intensive, and time-consuming, and usually hinders the rapid release of an app.

To generate high-quality test inputs to thoroughly explore an app's functionalities, we observe that a long and meaningful test input can usually be *decomposed* into relatively independent *use cases*. A use case is a short event sequence for manifesting a designated app's functionality, e.g., ① toggling a setting, ② switching to an activity, or ③ downloading a Web content. ① → ② → ③ is a long (and meaningful) test input, and is particularly useful in manifesting diverse app behaviors when the app's behavior in ③ varies on different settings in ①.

Conversely, we can solve the problem of generating long and meaningful test inputs by a fundamentally different two-phase approach, which we call it the *ComboDroid* framework:

1. *Collect high-quality use cases* that cover as many basic app functionalities as possible.
2. *Concatenate a number of use cases* to form a test input for covering complex functionalities.

Use cases can be either manually provided (e.g., by an app's developer) or automatically extracted from execution traces. Since developers clearly know how the requirements are implemented, they can easily provide high-quality use cases with little manual labor. To extract use cases automatically from execution traces,

[1] In the context of testing Android apps, a test input is a sequence of the Android system's atomic *input events* (touching, swiping, etc.).

we leverage the insight that use cases, by their definitions, almost begin and end at *quiescent* app states, usually with a stable GUI. We accordingly designed an algorithm to automatically identify such GUI states and extract use cases from long event sequences.

To efficiently generate high-quality use case combinations (or *combos* for short) as test inputs, we devise an algorithm to triage combos for a maximized testing diversity. Particularly, we define the *aligns-with* relation, which determines whether two use cases connected at the same quiescent state, to prune likely invalid combos. We also define the *depends-on* relation, which determines whether a use case can affect the behavior of another. We generate only aligned combos with sufficient data-flow diversities for an effective test input generation.

We implemented these ideas as the ComboDroid tool, including the fully automatic ComboDroid$^\alpha$ and semi-automatic ComboDroid$^\beta$. The evaluation results are encouraging that ComboDroid is effective in both testing scenarios:

1. The fully automatic ComboDroid$^\alpha$ covered 4.6% and 6.7% more code on average compared with the most effective existing technique APE [28] and most widely used technique Monkey [26], respectively. ComboDroid$^\alpha$ also revealed four previously unknown bugs in extensively tested subjects [51–54].
2. The semi-automatic ComboDroid$^\beta$ boosted the coverage of manually provided use cases by 13.2%, achieving a competitive code coverage (the gap is only 3.2%) compared with a human testing expert, but with much less manual labor.

The rest of this paper is organized as follows. Section 2 presents an overview of our approach with an illustrative example. Details of our approach are discussed in Section 3. Section 4 introduces the ComboDroid implementation and our extensive evaluation is conducted in Section 5. Section 6 surveys related work, and Section 7 concludes this paper.

2 OVERVIEW

Figure 1 displays the ComboDroid workflow. ComboDroid takes an app under test P and repeats the two-phase testing procedure consisting of obtaining use cases (the left box) and enumerating use case combos (the right box). We explain the workflow of ComboDroid using a motivating example, a previously unknown bug[2] found by ComboDroid$^\alpha$ in Aard2 (a popular dictionary app). This bug requires a long (and meaningful) test input ① → ③ → ② → ④ → ⑤ to trigger.

Obtaining use cases. We first observe that a meaningful use case (event sequence) usually begins and ends at *quiescent* app states, in which the app is idle (completes handling of all received events) on a stable GUI. Quiescent states naturally indicate that a human can perform the next step of an action in the computer-human interaction. In Aard2, useful use cases include adding/deleting a dictionary, searching for a word, view a word's detail explanations, etc.

Use cases can be provided by a human developer (noted CD$^\beta$). ComboDroid contains an auxiliary tool to help developers collect uses cases by recording event sequences (both UI and system events [46]) at a specified time interval. ComboDroid automatically identifies quiescent states, and collects execution traces and GUI snapshots along with the use cases. In Aard2, the app's developer would have no difficulty in providing meaningful use cases like ①, ②, . . . , ⑤, and ✪.

Use cases can also be extracted by an automatic analysis of an app's existing execution traces (noted CD$^\alpha$). ComboDroid mines an *extended labeled transition system* (ELTS) [31] at runtime based on the GUI transitions using an existing algorithm [10]. Similar stable GUIs are clustered as a single state in the ELTS. Each input event between a pair of stable GUIs in the execution traces is added as a transition (labeled with that event) in the ELTS.

To bootstrap CD$^\alpha$ (as there is no trace at first), we implemented a baseline DFS-alike state space exploration tool [5] to generate initial testing traces. Unique acyclic transitional paths on the ELTS are extracted as likely use cases. In Aard2, automatically generated use cases are not as readable as manual ones, but share similar features (e.g., starting from and ending at quiescent app states). Nevertheless, ComboDroid$^\alpha$ successfully identified different pages (e.g., the dictionary, search, and detail page) as distinct states in the ELTS, and the generated use cases cover all functionalities in ①, ②, . . . , ⑤, and ✪.

Enumerating use case combos. Either way, ComboDroid enumerates the combinations (combos) of use cases to obtain high quality test inputs. A combo is a sequence of use cases

$$\textcircled{u_1} \rightarrow \textcircled{u_2} \rightarrow \ldots \rightarrow \textcircled{u_n}$$

where u_1 starts from the app's initial state. To make combos effective in testing, a combo should additionally satisfy:

1. *Deliverability*: for all $1 \le i < n$, u_i aligns with u_{i+1}. For u to be aligned with v, the last GUI layout in u should be similar to the first one of v (such that it is sane to deliver v to the app immediately after u). Similarity is characterized by an editing-distance based measurement.
2. *Dataflow diversity*: there exists at least k distinct pairs of (u_i, u_j) where u_i depends on u_j and $i > j$. For u to be dependent on v, there should be some shared program states used in u and modified in v. Thereby we filter out loosely connected use case combos.

The systematic enumeration in ComboDroid first searches for data-dependent pairs for a maximized data flow diversity, and then adds random transitional use cases to satisfy the deliverability. In Aard2, ① → ② and ④ → ⑤ are data-dependent[3]. Then, ComboDroid generates ① → ② → ④ → ⑤ as a skeleton, which is filled with transitional use cases (③ and ✪s) to yield the bug-triggering combo in Figure 1 (a combo of $n = 8, k = 2$).

The feedback loop. Generated combos are delivered to the app with execution traces being collected. After the delivery, ComboDroid terminates if there is no newly explored quiescent app state other than those identified during the use case generation. Otherwise, ComboDroid restarts the first phase to either ask a human for additional effective use cases concerning these states (e.g., visiting them during the execution), or extract more potentially profitable

[2] Aard2 has been extensively evaluated in the existing studies [43, 56, 57]. However, ComboDroid$^\alpha$ is the first to uncover this bug.

[3] Use case *delete a dictionary* (②) overwrites the dictionary object referred in *add a dictionary* (①), and thus ② depends on ①. For a similar reason on the shared `WebView` object, ⑤ depends on ④.

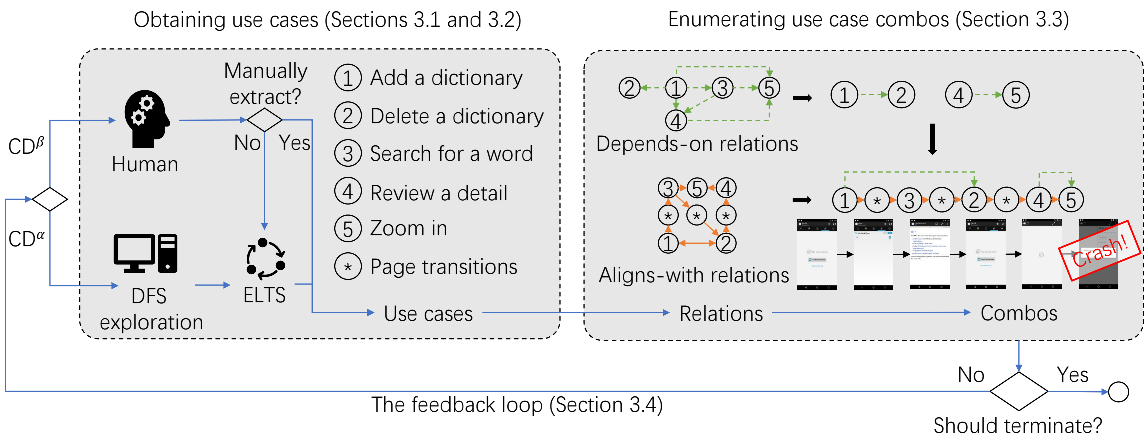


Figure 1: ComboDroid overview and a motivating bug example

use cases from the ELTS refined by the newly collected execution traces.

Manifestation of the bug. The combo in Figure 1 crashes the app. After deleting a dictionary, all of its detail word explanations are removed. However, the "detail" page of a previously searched word is still cached in the app. Returning to such a detail page displays a null (blank) `WebView`. A subsequent zoom-in triggers the crash by a `NullPointerException`. All eight use cases (12 events) are necessary to trigger the bug, and such a long event sequence is not likely to be generated by existing techniques, which indeed failed to do so in our evaluation.

3 APPROACH

3.1 Notations and Definitions

Given an Android app P, our goal is to generate high-quality test inputs via use case combinations. Android apps are GUI-centered and event-driven. The runtime GUI layout (snapshot) ℓ is a tree in which each node $w \in \ell$ is a GUI widget (e.g., a button or a text field object). We use $w.type$ to refer to w's widget type (e.g., a button or a text field). When P is inactive (closed or paused to background), there is no GUI layout and $\ell = \bot$.

An event $e = \langle t, r, z \rangle$ is a record in which $e.t$, $e.r$, and $e.z$ denote e's event type, receiver widget, and associated data, respectively. An event can be either a UI event or a system event, and examples of t are "ui-click", "ui-swipe", or "sys-pause". For a UI event, the receiver $r(\ell) = w$ denotes that e can be delivered to $w \in \ell$ at runtime. $r(\ell) = \bot$ indicates that this event cannot be delivered. A system event's receiver is always the "system" widget. Other event-specific information is stored in z, e.g., texts entered in a text field or the content of an added file.

Executing P with a sequence of events $E = [e_1, e_2, \ldots, e_n]$ yields an *execution trace* $\tau = \texttt{Execute}_P(E) = \langle L, M, T \rangle$. As defined in Algorithm 1, L[4], M, and T denote the dumped GUI layouts, method invocation trace, and each event's corresponding method invocations, respectively.

Algorithm 1: Execution of a sequence of events

```
Function Execute_P(E)
    $\ell \leftarrow$ GetGUI(); $L \leftarrow [\ell]$; $M \leftarrow \varnothing$; $T \leftarrow \varnothing$;
    for each $e \in E$ do
        if $r(\ell) \neq \bot$ then // $e$ can be sent on $\ell$
            $M' \leftarrow$ SendEventToApp($e.t$, $e.r(\ell)$, $e.z$); // send event $e$ to $P$, wait for a quiescent state, and return the corresponding method invocation sequence
            $M \leftarrow M :: M'$; $T \leftarrow T \cup \{\langle e, M' \rangle\}$;
            $\ell \leftarrow$ GetGUI(); $L \leftarrow L :: [\ell]$;
        else
            return $\bot$;
    return $\langle L, M, T \rangle$;
```

A use case $u = [e_1, e_2, \ldots, e_{|u|}]$ is also an event sequence. It is straightforward for a human developer to manually provide use cases in either way: (1) annotating use cases as substrings in an execution trace τ, or (2) feeding τ to the following automatic extraction algorithm.

3.2 Use Case Extraction

Use cases are extracted upon a mined extended labeled transition system [31] (ELTS). Furthermore, in the fully automatic settings in which no trace is provided, we use a standard depth-first exploration to obtain a bootstrapping trace.

Mining an Automaton. Given an execution trace $\tau = \langle L, M, T \rangle$ from executing event sequence E, its corresponding ELTS is a three-tuple $G = \langle S, E, \delta \rangle$, in which S is a set of abstract states ($\{s \mid s \in S\}$ is a partition of the GUI layouts L) and $\delta : S \times E \longrightarrow S$ contains the state transitions.

We adopt the existing algorithm in SwiftHand [10] for mining a minimal ELTS that groups similar GUI layouts together, i.e., $\texttt{equivalent}(\ell_1, \ell_2)$[5] holds for all GUI layouts ℓ_1, ℓ_2 in the same

[4] For $L = [\ell_1, \ell_2, \ldots, \ell_{n+1}]$, ℓ_i is the GUI layout dump (at a quiescent state) after the first $i - 1$ events in E are sent to the app.

[5] We use the Lv.4 GUI Comparison Criteria (GUICC) of AMOLA [6] to measure the similarity between GUIs, i.e., GUI layouts ℓ_1 and ℓ_2 are equivalent if and only if $\forall e \in E. e.r(\ell_1) \neq \bot \leftrightarrow e.r(\ell_2) \neq \bot$.

Algorithm 2: ELTS Mining

```
Function
  MineELTS($\langle L = [\ell_1, \ell_2, \ldots, \ell_{n+1}], M, T\rangle$, $E = [e_1, e_2, \ldots, e_n]$)
    $S \leftarrow \{\{\ell\} \mid \ell \in L\}$; // initially, no state is merged
    $\delta \leftarrow \{\langle \ell_i, e_i, \ell_{i+1}\rangle \mid 1 \le i \le n\}$;
    for each $(s_i, s_j) \in S \times S$ and $s_i \neq s_j$ do // in the BlueFringe
     ordering [35]
        $\langle S', \delta'\rangle \leftarrow$ merge-recursive($s_i$, $s_j$, $S$, $\delta$);
        if $\langle S', \delta'\rangle \neq \perp$ then
            $\langle S, \delta\rangle \leftarrow \langle S', \delta'\rangle$; // update merged states
    return $\langle S, E, \delta\rangle$;
Function merge-recursive($s$, $t$, $S$, $\delta$)
    if $\forall \ell_1 \in s, \ell_2 \in t$.equivalent($\ell_1$, $\ell_2$) then
        $S' \leftarrow S \setminus \{s, t\} \cup \{s \cup t\}$; $\delta' \leftarrow \delta[s/t]$;
        for each $\langle s, e, t_1\rangle, \langle s, e, t_2\rangle \in \delta$ where $t_1 \neq t_2$ do
            $\langle S', \delta'\rangle \leftarrow$ merge-recursive($t_1$, $t_2$, $S'$, $\delta'$);
            if $\langle S', \delta'\rangle = \perp$ then
                break;
        return $\langle S', \delta'\rangle$;
    return $\perp$; // merging failed
```

state s. Such an algorithm (Algorithm 2) is originally used in the dynamic model extraction of Android apps.

Extracting use cases. A valid path $p = [s_0, s_1, \ldots, s_m]$ on $G(S, E, \delta)$ where $\delta(s_{i-1}, e_i) = s_i$ for all $1 \le i \le m$ naturally corresponds to the sequence of events

$$u = [e_1, e_2, \ldots, e_m]$$

as a likely use case. Therefore, the automatic use case extraction algorithm enumerates all acyclic paths in G and produces a use case for each of them.

Note that our automatic algorithm extracts *likely* use cases from the ELTS. In such a manner, we can maximize the chance of exhausting all possible use cases. Moreover, most likely use cases can be real use cases, while others share similar features with them (e.g., starting from and ending at quiescent app states) and can also be effective exploring the app's behavior.

Bootstrapping the use case generation. In the fully automatic setting of ComboDroid, the use case extraction is bootstrapped by a standard DFS-alike state space exploration strategy similar to the A^{3}E algorithm [5].

Starting from the initial state, we take the GUI layout snapshot ℓ, analyze all widgets $w \in \ell$ for all possible actions on w. For each action (e.g., clicking a button, or entering a random text from a predefined dictionary to a text field [43]), we create an event e[6] and add it to E_{ui}. We then sequentially execute (send the event to the app and wait for a quiescent state) all events in $E_{ui} \cup E_{sys}$, where E_{sys} is a set of predefined system events. If executing an event reaches an unexplored GUI ℓ', the exploration is recursively conducted on ℓ'; if all events are exercised or reaching an explored GUI, backtracking is performed (thus this is a depth-first exploration). The depth-first exploration yields a sequence of events E_{dfs}.

3.3 Enumerating Use Case Combos

Suppose that use cases $U = \{u_1, u_2, \ldots, u_n\}$ are extracted from execution trace $\tau = \langle L, M, T\rangle$ by executing event sequence E. A *use case combination* (or *combo*) is a sequence of use cases denoted by $[u_{i_1} \rightarrow u_{i_2} \rightarrow \ldots \rightarrow u_{i_k}]$. Sequentially concatenating the events in the use cases of a combo yields a runnable test input.

Unfortunately, randomly generated combos usually stop early in an execution because there will likely exist an event e that has no receiver on the deliver-time GUI ℓ, i.e., $e.r(\ell) = \perp$. Consider the combo ② → ⑤ in the motivating example (Figure 1). The "zooming in" event has no receiver after deleting a dictionary because the current GUI does not contain a `ListView` menu containing the `ZoomIn` button.

To generate high-quality use case combos, we leverage the following two use case relations:

Aligns-with. For two use cases $u = [e_1, e_2, \ldots, e_n]$ and $v = [e'_1, e'_2, \ldots, e'_m]$, we say that u aligns with v, or $u \rightsquigarrow v$, if we have witnessed once that e'_1 can be successfully delivered after e_n. In other words, $u \rightsquigarrow v$ if $e'_1.r(\ell_n) \neq \perp$ where ℓ_n is the GUI layout after the execution of $e_n \in E$ in the trace τ.

Another issue in the use case alignment (and replaying an event sequence) is to determine how to deliver a UI event e to a particular GUI layout. For $\ell = \{w_1, w_2, \ldots, w_{|\ell|}\}$ being the GUI layouts right before e was sent in τ, and an arbitrary $\ell' = \{w'_1, w'_2, \ldots, w'_{|\ell'|}\}$, we know that there exists $1 \le i \le |\ell|$ such that $e.r(\ell) = w_i \in \ell$ because w_i is e's receiver widget in τ. Therefore, the widget $w'_j \in \ell'$ that is "most similar" to w_i should be the receiver of e on ℓ', i.e., $e.r(\ell') = w'_j$.

To measure the similarity between GUI layouts, we compute the *editing distance* between ℓ and ℓ' using the algorithm in RepDroid [68]. We find the shortest editing operation sequence (each editing operation is either inserting or removing a widget) that transforms ℓ to ℓ'. If w_i is not removed during the transformation, it must have a unique correspondence $w'_j \in \ell'$. We thus let $e.r(\ell') = w'_j$; otherwise w_i is removed and $e.r(\ell') = \perp$.

Depends-on. For use cases u and v, we say that v depends on u, or $u \dashrightarrow v$, if the two use cases are potentially data-dependent. Data dependency is measured at a method level. Considering the method invocation trace in τ, if there exists a method $m \in T(e)$ for $e \in u$ and $m' \in T(e')$ for $e' \in v$ such that m' data-depends on m, we say that $u \dashrightarrow v$. Data dependencies between methods are determined by a lightweight static analysis. m' data-depends on m if there is an (abstract) object or resource write-accessed in m and read-accessed in m'.

Combo generation. *Aligns-with* and *depends-on* relations guide our use case combination (combo) generation. To maximize the diversity of generated combos, we enforce each combo $c = [u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_{|c|}]$ to satisfy:

(1) *Each combo is an independent test case*: $e_1.r(\ell_0) \neq \perp$ for $\ell_0 \in L$ being the app's initial GUI layout and e_1 being the first event in u_1;
(2) *Consecutive use cases in the combo are aligned*: $u_i \rightsquigarrow u_{i+1}$ for all $1 \le i < |c|$; and

[6] For $e = \langle t, r, z\rangle$, $e.t$ and $e.z$ are straightforward to determine. The receiver $e.r$ is determined by an editing-distance based algorithm described later in Section 3.3.

Algorithm 3: Combo Generation

```
1  Function RandomCombo($U$, $\ell_0$, $k$)
2      $G(V, E) \leftarrow$ randomDAG($2k$); // random DAG of $|E| = 2k$
3      $F \leftarrow \{(v,$ randomChoice$(U)) \mid v \in V\}$; // randomly assign each
         $v \in V$ a use case in $U$
4      if $|\{e \mid e = \langle v_1, v_2 \rangle \in E \wedge F(v_1) \dashrightarrow F(v_2)\}| \geq k$ then
5          for each linear extension $[v_1, v_2, \ldots, v_{|V|}]$ of $G$ do
6              for $u_0 = [e_1, e_2, \ldots, e_m] \in U \wedge e_1.r(\ell_0) \neq \bot$ do
7                  $c \leftarrow$ connect($u_0$, $F(v_1)$, $U$, 0) ::
                     connect($F(v_1)$, $F(v_2)$, $U$, 0) :: ... ::
                     connect($F(v_{n-1})$, $F(v_{|V|})$, $U$, 0) :: $[F(v_{|V|})]$;
                     // add paddings such that consecutive use cases are
                     aligned
8                  if $\bot \notin c$ then
9                      return $c$;
10     return $\bot$;
11 Function connect($u$, $dst$, $U$, $depth$)
12     if $u \rightsquigarrow dst$ then
13         return $[u]$;
14     if $depth >$ MAX_DEPTH then
15         return $\bot$;
16     for $u' \in U \wedge u \rightsquigarrow u'$ do
17         $seq \leftarrow$ conncet($u'$, $dst$, $U$, $depth + 1$);
18         if $seq \neq \bot$ then
19             return $[u]$ :: $seq$;
20     return $\bot$;
```

(3) *Use cases in a combo exhibit k-data-flow diversity*, i.e., there exists k distinct pairs of (u_i, u_j) $(1 \leq i < j \leq |c|)$ such that $u_i \dashrightarrow u_j$.

The algorithm for generating a combo is presented in Algorithm 3. Given a set of use cases U, the app's initial GUI layout ℓ_0, and a data-flow diversity metric k, a random *skeleton* is first sampled. A skeleton is a directed acyclic graph $G(V, E)$ where $|E| = 2k$. If the data-flow diversity of G is less than k (Line 4), the generation should be restarted. Otherwise, each vertex $v \in V$ is assigned with a random use case $F(v)$ in U (Lines 2–3);

A liner extension of the skeleton G corresponds to a sequence of use cases: $[F(v_1), F(v_2), \ldots, F(v_{|V|})]$. We try to *pad* use cases $F(v_i)$ and $F(v_{i+1})$ $(1 \leq i < |V|)$ with more use cases to obtain a combo c such that consecutive use cases in c are aligned (Line 7). The padding use cases are depth-first searched with a maximum length limit MAX_DEPTH (Lines 11–20).

We also add paddings before the first use case in c (Line 6) such that the resulting combo can be delivered to the initial app state (and thus c can be used as an independent test case). If all aforementioned paddings exist[7], we successfully obtained a use case combo satisfying our requirements (Lines 8–9). Such a combo is sent to the app for testing.

3.4 Feedback Loop of ComboDroid

As Figure 1 shows, there can be multiple iterations of use case generation and combo enumeration. When enumerated combos are sent to the app and discovered previously unknown states, a new iteration should be initiated. Before the next iteration starts, a developer can manually inspect the testing report and provide/annotate more use cases.

Suppose that we concatenate the execution traces in all previous iterations of use case generation and combo enumeration. Conceptually, this can be regarded as adding an extra "restart" event after sending all events in a combo[8]. Such a merged trace is used for the ELTS mining and use case extraction in the next round of iteration.

4 IMPLEMENTATION

The ComboDroid framework is implemented using Kotlin and Java. ComboDroid consists of a fully automatic variant ComboDroid$^\alpha$ and a semi-automatic variant ComboDroid$^\beta$. We extensively used open-source tools in the implementation, and ComboDroid is also open-source available[9]: GUI events are recorded by `Getevent` [25]; GUI and system events are delivered using Android Debug Bridge (ADB) [22]; GUI layouts are dumped by Android `UI Automator` [24]; method traces are collected by program instrumentation with Soot [13]. The implementation follows the descriptions in Section 3. We follow the common practice of existing state-of-the-art techniques [6, 28, 43, 56] and identify quiescent app states by stabilized GUIs. Specifically, we take the same implementation as APE [28] by dumping GUI layouts every 200ms until it is stable (using the Lv.4 GUICC) with a 1000ms upper-bound.

For ComboDroid$^\alpha$, we implement the DFS-alike exploration tool, and follow the same implementation as APE [28] by replaying previous execution traces for backtracking. For ComboDroid$^\beta$, we analyze the GUI layouts, where the human tester sends each event, together with the corresponding event to determine each event's receiver.

In the lightweight static analysis to determine the depends-on relation, we also model the Android 6.0 APIs (API level 23) [23] to determine read/write accesses to resources, e.g., we determine whether an SQL command in `SQLiteDatabase.execSQL` is a read or write to the database. Moreover, for an (abstract) object, if any method whose name matches the regular expression

```
(get|is|read)(.+)|on(.+)changed
```

is called, we consider it a read; Similarly, calling any method whose name matches

```
(set|write|change|modify)(.+)
```

is considered a write.

The depth-first exploration of ComboDroid$^\alpha$ was set with a time limit of 30 minutes in each iteration. In the combo generation (both ComboDroid$^\alpha$ and ComboDroid$^\beta$), we set data-flow diversity $k = 2$ and MAX_DEPTH $= 5$. We generate d^2 random combos (by `RandomCombo`) if there are d *depends-on* edges.

[7] A transition between each pair of GUIs naturally exist for a well-designed app. Therefore, it is highly like that all aforementioned paddings exist.

[8] A restart event is also added after ELTS mining.

[9] https://github.com/skull591/ComboDroid-Artifact.

5 EVALUATION

This section presents our evaluation of ComboDroid. The experimental subjects and setup are described in Section 5.1, followed by evaluation results of ComboDroid$^{\alpha}$ (the fully automatic variant) and ComboDroid$^{\beta}$ (the human aided variant) in Sections 5.2 and 5.3, respectively. Discussions including threats to validity are presented in Section 5.4.

5.1 Experimental Subjects and Setup

The first column of Table 1 lists the 17 evaluation subjects. The apps are selected using the following rules: First, we selected the three largest (in LoC) apps evaluated in existing work [28, 43, 56]: WordPress, K-9 Mail, and MyExpense. Second, we randomly selected nine apps with at least 10K Downloads evaluated in the existing work [28, 43, 56]: Wikipedia, AnkiDroid, AmazeFileManager, AnyMemo, Hacker News Reader, CallMeter, Aard2, World Clock, and Alogcat. Additionally, we randomly selected five popular (at least 100 stars by 2018) open-source apps from Github: AntennaPod, PocketHub, SimpleTask, Simple Draw, and CoolClock.

If an app's major functionalities cannot be accessed without a proper initial setup (e.g., user login), we provide the app a script to complete the setup. All evaluated techniques receive exactly the same script (and the script runs automatically once the initial setup GUI is reached) to ensure a fair comparison. We did not mock any further functionality other than the initial setup script.

We use two metrics to measure the testing thoroughness. The first is bytecode instruction coverage collected by JaCoCo [32], as a higher code coverage strongly correlates to a better exercise of an app's functionalities. Second, we study whether the techniques can manifest (and reproduce) previously known or unknown bugs by examining the Android system's logs.

To evaluate ComboDroid$^{\alpha}$, we compare it with the state-of-the-art automated techniques: Monkey [26], Sapienz [43], and APE [28][10]. For each subject, we ran each automatic testing technique for 12 hours to simulate a nightly continuous integration build-and-test cycle. We ran ComboDroid$^{\alpha}$ for termination or 12 hours at most. For each subject, we ran each techniques three times and reported the average results. Test coverage and bug manifestation results are then studied.

To evaluate ComboDroid$^{\beta}$, we compare it with a human expert. For manual use case generation of ComboDroid$^{\beta}$, we gave a recruited tester one work day (~8 work hours) for each test subject and then ran each subject for 12 hours. Meanwhile, we asked another independently recruited Android testing expert (a post-graduate student who had published a few research papers on testing and analysis of Android apps) to cover as much code as possible given a time limit of three workdays (~24 work hours). The human expert had access to an app's source code and was told to use coverage feedback to maximize code coverage. Since manual labor is not scalable, we only evaluated the subjects of top 10 LoC, as shown in Table 3. To reduce distractions (as confirming and diagnosing bugs are time-consuming), we asked the human expert not to provide any bug report. Therefore, only test coverage is studied in the evaluation of ComboDroid$^{\beta}$.

All experiments were conducted on an octa-core Intel i7-4790U PC with 16 GiB RAM running Ubuntu 16.04 LTS and an Android 6.0 Emulator.

5.2 Evaluation Results: ComboDroid$^{\alpha}$

The 12-hour coverage results and manifested bugs are listed in Table 1 and Table 2, respectively. The detailed coverage trends are plotted in Figure 2. These results are consistent with a recent empirical study [59]: automated techniques by that time barely outperform the simplest Monkey. The best existing technique, APE, marginally outperforms Monkey by covering 2.1% more code.

Encouragingly, ComboDroid$^{\alpha}$ consistently outperforms existing techniques in nearly all subjects[11]. For Alogcat, CoolClock, and CallMeter, ComboDroid$^{\alpha}$ terminated within 12 hours, while for other subjects it ran until the time limit exceeded. Compared with the best existing technique APE, ComboDroid$^{\alpha}$ covered 4.6% more code on average. This improvement is even 2× as much as the improvement of APE over Monkey. Considering that the APE implementation generates ~1.5× more events in 12 hours (~120K for ComboDroid$^{\alpha}$ v.s. ~300K for APE), ComboDroid$^{\alpha}$ is considerably more effective in exploiting each event's merits.

The progressive coverage in Figure 2 shows that ComboDroid$^{\alpha}$ usually begins to outperform existing techniques after six hours. Consider that the current ComboDroid$^{\alpha}$ implementation emits events at ~1/2 speed, the result is also promising. Furthermore, code coverage gain of existing techniques is usually marginal (or zero) in the last hours. In contrast, ComboDroid$^{\alpha}$ is consistently exploring useful use case combinations to cover more code.

The bug manifestation evaluation results (Table 2) are also encouraging. We manually examined the test logs of all techniques and found 12 reproducible bugs with an explicit root cause. Excluding the bug in Simple Draw (ComboDroid$^{\alpha}$ missed it due to an implementation limitations), ComboDroid$^{\alpha}$ manifested *all* 11 previously known or unknown bugs, where the best existing technique, APE, manifested 7 (64%). We also reported four *previously unknown* bugs (APE can discover only two of them) to the developers. All of them were confirmed and two of which have been fixed. Furthermore, the two previously unknown bugs uniquely discovered by ComboDroid$^{\alpha}$ are *deep* bugs which require a long (and meaningful) input sequence to trigger. The motivating example (Figure 1) is such a case.

Therefore, we hold strong evidence that ComboDroid$^{\alpha}$ is more effective in automatically generating high-quality test inputs for Android apps compared with existing techniques.

5.3 Evaluation Results: ComboDroid$^{\beta}$

The evaluation results of ComboDroid$^{\beta}$ are displayed in Table 3. For all subjects, ComboDroid$^{\beta}$ ran until the time limit exceeded. It is expected that the human testing expert significantly outperforms

[10] Since APE [28] significantly outperforms Stoat [56] and other related work, we did not show results of other techniques in this paper.

[11] APE and ComboDroid$^{\alpha}$ covered less code for Simple Draw compared with Monkey because the implementations do not identify canvas widgets and thus not send dragging events. We consider this an implementation limitation.

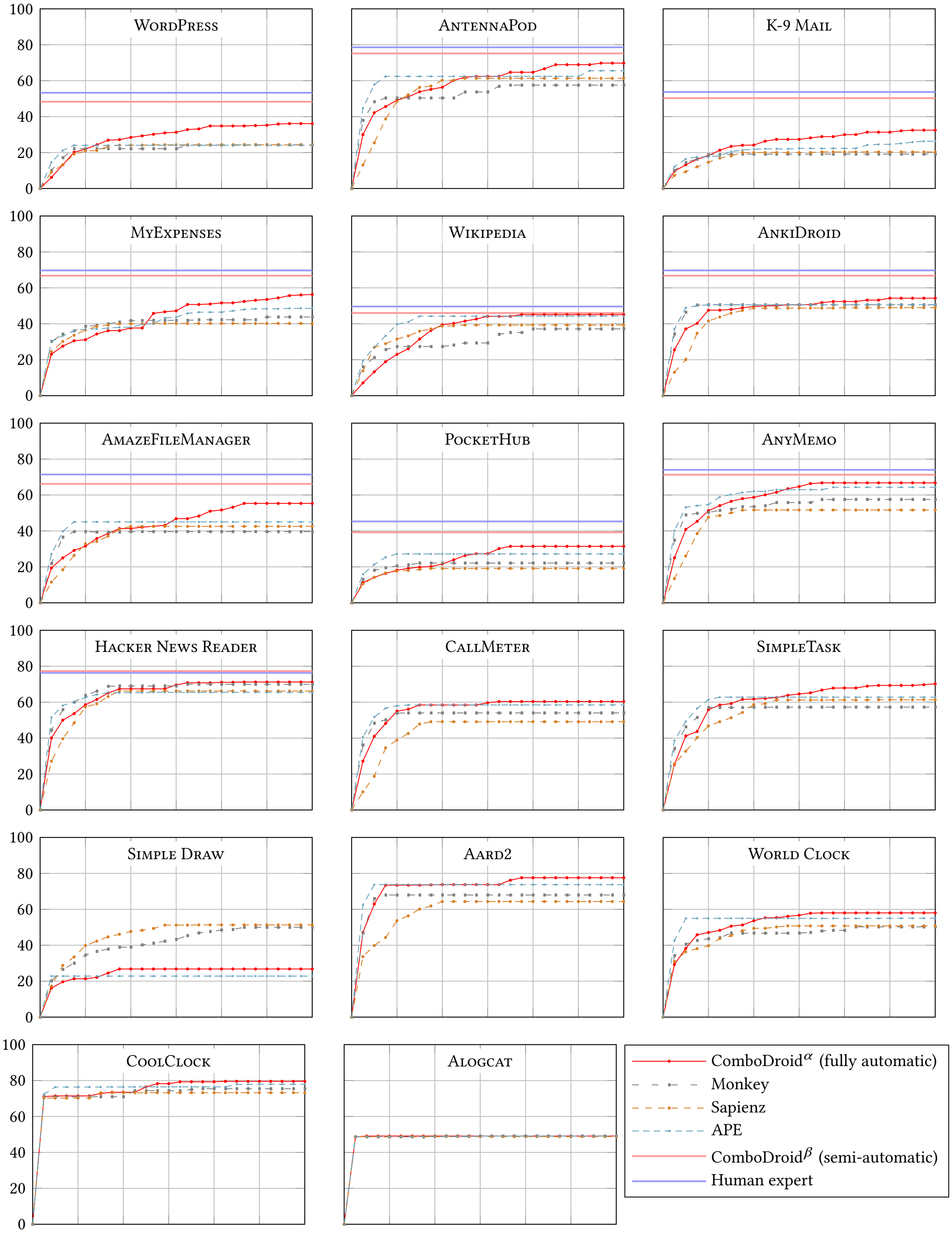
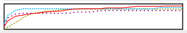
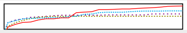
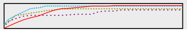
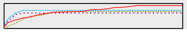
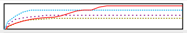
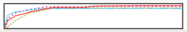
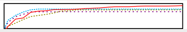
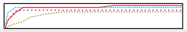
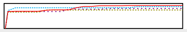
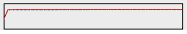


Figure 2: Progressive coverage report of evaluated techniques (averaged over three runs). The *x* axis is the time spent (0–12 hours). The *y* axis indicates the percentage of code covered thus far.

Table 1: Evaluation results of ComboDroid$^\alpha$: test coverage

Subject (Category, Downloads; LoC)	Monkey	Sapienz	APE	ComboDroid$^\alpha$	Coverage trend
WordPress, WP (Social, 5M–10M; 327,845)	24.4%	24.3%	24.1%	36.1% (+11.7%)	
AntennaPod, AP (Video, 100K–500K; 262,460)	57.5%	61.3%	65.5%	69.8% (+4.3%)	
K-9 Mail, K9 (Communication, 5M–10M; 159,708)	19.1%	20.4%	26.3%	32.5% (+6.2%)	
MyExpenses, ME (Finance, 500K–1M; 104,306)	43.8%	40.2%	48.6%	56.3% (+7.7%)	
Wikipedia, Wiki (Books, 10M–50M; 93,404)	37.2%	39.3%	44.3%	45.1% (+0.8%)	
AnkiDroid, AD (Education, 1M–5M; 66,513)	50.6%	49.0%	50.6%	54.3% (+3.7%)	
AmazeFileManager, AFM (Tools, 100K–500K; 66,126)	39.6%	42.5%	45.0%	55.2% (+10.2%)	
PocketHub, PH (Tools, 100K–500K; 47,946)	22.1%	19.1%	27.2%	31.4% (+4.2%)	
AnyMemo, AM (Education, 100K–500K; 40,503)	57.5%	51.7%	64.3%	66.8% (+2.5%)	
Hacker News Reader, HNR (News, 50K–100K; 38,315)	69.9%	66.2%	65.5%	71.2% (+1.3%)	
CallMeter, CM (Tools, 1M–5M; 21,973)	54.0%	49.1%	58.5%	60.4% (+1.9%)	
SimpleTask, ST (Productivity, 10K–50K; 20,980)	57.2%	57.2%	62.8%	70.2% (+7.4%)	
Simple Draw, SD (Tools, 10K–50K; 18,685)	50.0%	51.3%	22.8%	26.8% (-24.5%)	
Aard2, Aard (Books, 10K–50K; 9,622)	68.0%	64.3%	73.8%	77.6% (+3.8%)	
World Clock, WC (Bussiness, 1M–5M; 7,181)	50.2%	50.8%	55.1%	58.0% (+2.9%)	
CoolClock, CC (Tools, 10K–50K; 2,762)	75.4%	73.2%	78.0%	79.6% (+1.6%)	
Alogcat, ALC (Tools, 100K–500K; 846)	49.1%	48.8%	49.1%	49.1% (0.0%)	
Average	48.6%	47.6%	50.7%	55.3% (+4.6%)	

[1] Column **Coverage trend** plots the coverage trend of each tool. The red solid lines denote ComboDroid$^\alpha$, and dashed lines are existing techniques. The detailed coverage trends are displayed in Figure 2. Number in a bracket is the coverage differences between ComboDroid$^\alpha$ and the best existing technique (Monkey, Sapienz, and APE).

Table 2: Evaluation results of ComboDroid$^\alpha$: bug manifestation

Bug ID	Cause	Discovered by
WP-10147	Infinite recursion	APE, CD$^\alpha$
AP-1234	Atomicity violation	CD$^\alpha$
AP-3195	Null pointer dereference	all
K9-3308	Mismatched mime type	Sapienz, CD$^\alpha$
AFM-1351	Null pointer dereference	all
AFM-1402	Lifecycle event mishandling	APE, CD$^\alpha$
AM-480★	Lifecycle event mishandling	CD$^\alpha$
AM-503★	Null pointer dereference	APE, CD$^\alpha$
CM-128★	Text input mishandling	APE, CD$^\alpha$
SD-49	Miss-used local variables	Monkey
Aard-90★	Null pointer dereference	CD$^\alpha$
Aard-7	Null pointer dereference	all
Monkey: 4 (33%); Sapienz: 4 (33%); APE: 7 (58%); CD$^\alpha$: 11 (92%)		

[1] **Bug ID** is the issue ID in the project's GitHub repository. A starred **Bug ID**★ denotes a previously unknown bug.

Table 3: Evaluation results of ComboDroid$^\beta$: test coverage

Subject	UC	CD$^\alpha$	ComboDroid$^\beta$	Expert
WP (328K)	40.1%	36.1%	48.3% (+8.2%/+12.2%)	53.3% (+5.0%)
AP (262K)	65.4%	69.8%	75.2% (+9.8%/+5.4%)	78.6% (+3.4%)
K9 (160K)	38.5%	32.5%	50.3% (+11.8%/+17.8%)	53.7% (+3.4%)
ME (104K)	53.1%	56.3%	66.8% (+13.7%/+10.5%)	69.7% (+2.9%)
Wiki (93K)	37.3%	45.1%	46.0% (+8.7%/+0.9%)	49.6% (+3.6%)
AD (67K)	50.3%	54.3%	66.8% (+16.5%/+12.5%)	71.4% (+4.6%)
AFM (66K)	43.3%	55.2%	66.2% (+22.9%/+11%)	67.3% (+1.1%)
PH (48K)	31.5%	31.4%	39.2% (+7.7%/+7.8%)	45.3% (+6.1%)
AM (41K)	62.1%	66.8%	71.3% (+9.2%/+4.5%)	74.0% (+2.7%)
HNR (38K)	53.4%	71.2%	76.5% (+23.1%/+5.3%)	76.3% (-0.2%)
Average	47.5%	51.9%	60.7% (+13.2%/+8.8%)	63.9% (+3.2%)

[1] The number in Column **Subject** is the app's LoC. Columns **UC**, **CD**$^\alpha$, **ComboDroid**$^\beta$, and **Expert** display the code coverage of manual use cases, ComboDroid$^\alpha$, ComboDroid$^\beta$, and the human expert, respectively. The numbers in the brackets of Column **ComboDroid**$^\beta$ indicate the coverage differences between ComboDroid$^\beta$ and manual use cases and ComboDroid$^\alpha$, respectively. The numbers in the brackets of Column **Expert** indicate the differences between the human expert and ComboDroid$^\beta$.

automated techniques. Even the the best automated technique so far, ComboDroid$^\alpha$, covered 12.0% less code.

However, this gap is reduced to 3.2% when human knowledge is integrated into our framework: use case combinations additionally covered 13.2% more code than manual use cases only. ComboDroid$^\beta$ greatly amplified the use cases (covering 47.5% code, which is even 4.4% less than ComboDroid$^\alpha$) to achieve a result nearly as good as the human expert. Surprisingly, the ComboDroid$^\beta$ even outperformed the human expert in Hacker News Reader. After analyzing the code and coverage data, we found that Hacker News Reader can enable data-preload of news articles in the settings. When it is enabled, opening an article in an application-internal format

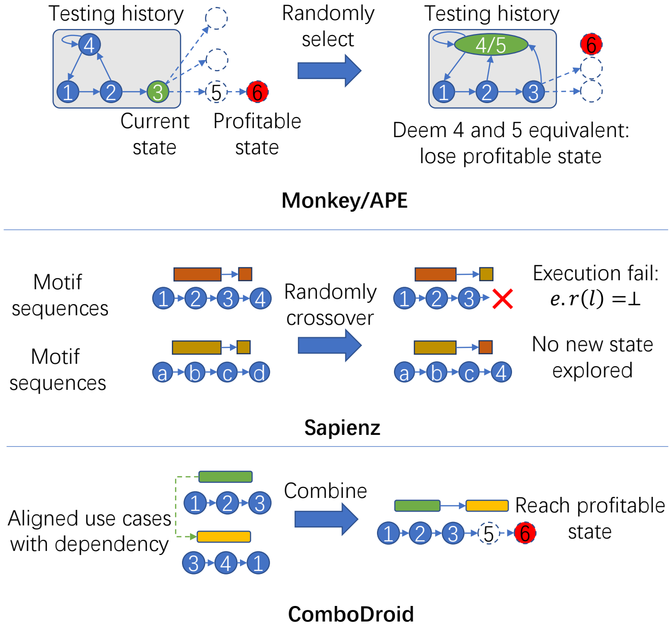


Figure 3: Qualitative illustration of the state space exploration strategies in evaluated techniques.

invokes additional code to process pre-loaded data. Such a subtle dependency is missed by the human expert; on the other hand, though also not covered by any single manual use case, ComboDroid$^\beta$ correctly pinpointed such a data dependency (in the depends-on relation) and accordingly generated the use case combination.

Though in a preliminary stage, ComboDroid$^\beta$ demonstrates the potential of automatically leveraging human insight in complementing and boosting automated techniques in testing Android apps.

5.4 Discussions

5.4.1 Towards Thorough Automatic Testing of Android Apps. Figure 3 illustrates the search strategies of the evaluated techniques, for giving a qualitatively explanation of why ComboDroid$^\alpha$ outperformed existing techniques.

Random-based techniques Monkey [26] and APE [28] at each time delivers exactly one event to the app, and therefore are completely unaware of the remaining state space. Their limitations are obvious: the search strategies are purely based on the noisy exploration history. Such strategies may easily lose a deep (and profitable) app state on random tries (e.g., pressing a button returns to the app's main menu).

Sapienz [43], though exploits motif sequences in a guided search, fails to effectively assembling them. First, there is no rationale or quality guarantee of the motif sequences—they are more or less random event sequences. Second, mutation and crossover operations in the genetic search are inefficient in creating useful motif sequence combos: randomly concatenating two event sequences will mostly result in a useless combo. It is not surprise that Sapienz even covered less code than Monkey in the long run. This result is consistent with the existing studies [59].

In contrast, ComboDroid$^\alpha$ generated both high-quality use cases and their combos, and thus is highly effective in covering app functionalities even if it delivers 60% less events.

Compared with manual testing, automatic testing is still far less satisfactory: the human expert covered 12.0% more code on average than ComboDroid$^\alpha$. The evaluation results of ComboDroid$^\beta$ show that this gap is mainly due to the quality of use cases. Our use case extraction algorithm simply cannot "understand" the app's functions and semantics, however, meaningful use cases are quite natural even for an app user. Machine learning over large-scale app usage data set may be a promising direction to address this issue.

5.4.2 Leveraging Human Insights in Semi-Automatic Testing of Android Apps. ComboDroid$^\beta$ successfully "amplified" the manual use cases to achieve a competitive coverage compared with a human expert: adding a little more human aid boosts the testing thoroughness. This partially validated our intuition that humans are good at sketching the major functionalities of the app; once such insights are extracted (as use cases), tedious and repetitive work can be offloaded to machines.

Therefore, ComboDroid$^\beta$, as a concept-proving prototype, opens a new research direction towards the human-machine collaborative testing of Android apps. Automatically generating meaningful (and handy) suggestions (either by program analysis or machine learning) to help manual testers, developers, or even users to provide better use cases is a rewarding future direction.

5.4.3 Threats to Validity. **Bias in the selected subjects**. The representativeness of selected test subjects can affect the fidelity of our conclusions. To mitigate this threat, we selected evaluation subjects from various sources: popular benchmarks evaluated in existing work plus random ones from GitHub. These subjects are (1) large in size (around 76 KLoC on average), (2) well-maintained (containing thousands of revisions and hundreds of issues on average), (3) popular (all have 10K+ downloads), and (4) diverse in categories. Since ComboDroid consistently and significantly outperforms existing techniques in all these benchmarks (except for Simple Draw due to the implementation limitation), the conclusion that ComboDroid$^\alpha$ is more effective than existing techniques is evident.

Randomness and non-determinism. The evaluated techniques (including ComboDroid) involve randomness, and subjects may be non-deterministic. Therefore, for each subject and technique we report the average result of three independent runs under the same settings (the experiments cost over 2,400 CPU hours) to alleviate this issue.

Human factors. The performance of human testers vary form person to person. Therefore, the evaluation results of ComboDroid$^\beta$ only apply to that human testing expert. Since the post-graduate Android testing/analysis expert knew us in advance, we are certain that he/she tried the best to cover as much code as possible.

6 RELATED WORK

Many technologies have been proposed for input generation for Android app testing, including both fully automatic ones and semi-automatic ones. Moreover, some technologies generating test inputs for GUI/web testing also share similarities with ComboDroid.

Fully automatic test input generation for Android apps. A majority of existing technologies aim to fully automatically generate test inputs for Android apps. Many of them generate test inputs for general testing purposes.

Random testing is a lightweight and practical approach in which a large number of random events are quickly fed to an app, including Monkey [26], DynoDroid [41], DroidFuzzer [64], IntentFuzzer [63], etc.

Using a GUI model (either predefined or mined) may guide the exploration of an app's state space. Representative work includes MobiGUITAR [3], SwiftHand [60], AMOLA [6], and the state-of-the-art APE [28]. Such state space exploration is usually done by a depth(breadth)-first search, e.g., A^3E [5], GAT [61], and EHBDroid [55]. However, even if with a model, existing techniques fall short on generating long (and meaningful) test inputs.

Search-based software engineering techniques can also be applied, such as EvoDroid [42] and Sapienz [43], which employ genetic programming to evolve generated test inputs, or Stoat [56], which constructs a stochastic model and uses MCMC [8] to guide the generation. Moreover, some researchers propose to utilize machine learning to guide the input generation [12, 27, 34]. Furthermore, some pieces of work utilize symbolic or concolic execution to systematically generate test inputs for maximizing branch coverage, including SIG-Droid [47], the technology proposed by Jensen et al. [30], SynthesiSe [17], and DroidPF [7]. Existing search-based techniques barely scale to large apps.

Finally, ComboDroid is not the first to introduce the idea of combination in Android app testing. However, existing combinatorial-based strategies [1, 48] concern only combinations of single events and thus unable to generate long (and meaningful) test inputs.

In conclusion, all existing technologies fall short on generating long (and meaningful) test inputs for practical apps, which are essential in manifesting deep app states and revealing many nontrivial bugs. The limitation of existing techniques motivated the design of ComboDroid.

Semi-automatic test input generation for Android apps. Some technologies are proposed to utilize human intelligence to improve the quality of generated test inputs. For instance, Polariz [44] extracts common event sequences from crowd-based testing to enhance SAPIENZ. AppFlow [29] records short event sequences provided by human, and utilizes machine learning to synthesize long event sequences. Moreover, UGA [38] extends manual event sequences exploring the skeleton of the app's state space. Though capable of utilizing human intelligence, Polariz and UGA have no control over the quality of extracted manual event sequences. On the other hand, AppFlow lacks an effective mechanism for reusing the event sequences in testing.

Domain-specific test input generation for Android apps. Some technologies aim to generate test inputs for certain testing domains or for manifesting certain kind of bugs. For instance, EOEDroid [62] utilizes symbolic execution to generate inputs to testing `WebViews` of an app, while SnowDrop [69] aims to test background services of an app. APEChecker [16] and AATT+ [36, 57] generate test inputs to manifest potential concurrency bugs in Android apps. Moreover, some technologies are proposed to detect energy inefficiency in Android apps, Such as GreenDroid [40] and its extensions [37, 39, 58] These techniques are generally orthogonal to ComboDroid. They can be benefited by the high-quality test inputs generated by ComboDroid.

Test input generation for GUI/web testing. Some technologies utilize iterative GUI exploration or program analysis to generate test inputs for GUI/web testing. Some pieces of work [2, 18–21, 45, 66, 67] iteratively observes the execution of existing test inputs, extracts additional knowledge (e.g., a refined model), and derives new test inputs. For instance, Nguyen et al. [49] proposes the OEM* paradigm that automatically identifies new test inputs during the execution of existing ones, expands the current incomplete GUI event model, and generates additional test inputs based on current execution traces. Such iterative process resembles ComboDroid. However, the knowledge extracted by these technologies mostly comes from observations of GUI transitions, and other relations between test inputs such as data dependency are often neglected.

On the other hand, some technologies utilize static analysis on program code to find data dependencies between events, and thus generate effective test inputs [4, 9, 14, 15, 50]. However, these technologies cannot be directly applied for testing Android apps, since Android apps are component-based with broken control-/data-flow, and often invoke Android-specific APIs to access shared data, e.g. `SharedPreference.getBoolean`.

In contrast, ComboDroid extracts knowledge of the app under test from both GUI transitions and data dependencies, and utilize lightweight static analysis on execution traces with Android-specific API modeling to infer depends-on relations between inputs.

7 CONCLUSION AND FUTURE WORK

Leveraging the insight that long, meaningful, and effective test inputs are usually the concatenation of short event sequences for manifesting a specific app functionality, this paper presents the ComboDroid framework in which the Android app test input generation problem is decomposed into a feedback loop of use case generation and use case combination. The evaluation results are encouraging. The fully automatic ComboDroid$^\alpha$ covered on average 4.6% more code than the best existing technique and revealed four previously unknown bugs. With little human aid, the semi-automatic ComboDroid$^\beta$ achieved a comparable coverage (only 3.2% less on average) with a human testing expert.

ComboDroid sheds light on a new research direction for obtaining high-quality test inputs, either fully automatic or with human aid. Based on this proof-of-concept prototype, a diverse range of technologies can be applied in the future enhancement of ComboDroid. Promising research includes exploiting machine learning in use case mining, crowd-sourced use cases acquisition, and model checking combos.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation (Grants #61690204, #61932021, and #61802165) of China. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn) and Chang Xu (changxu@nju.edu.cn) are the corresponding authors.

REFERENCES

[1] David Adamo, Dmitry Nurmuradov, Shraddha Piparia, and Renée Bryce. 2018. Combinatorial-based event sequence testing of Android applications. *Information and Software Technology* 99 (2018), 98–117.

[2] Pekka Aho, Matias Suarez, Teemu Kanstrén, and Atif M Memon. 2014. Murphy tools: Utilizing extracted gui models for industrial software testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 343–348.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.

[4] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, Martin Schäf, Ishan Banerjee, and Atif M Memon. 2012. Lightweight static analysis for GUI testing. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 301–310.

[5] Tanzirul Azim and Iulian Neamtiu. 2013. 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*. 641–660.

[6] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 238–249.

[7] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. 2017. Towards model checking android applications. *IEEE Transactions on Software Engineering* 44, 6 (2017), 595–612.

[8] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. 2011. *Handbook of markov chain monte carlo*. CRC press.

[9] Lin Cheng, Zijiang Yang, and Chao Wang. 2017. Systematic reduction of GUI test sequences. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 849–860.

[10] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.

[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 429–440.

[12] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 296–306.

[13] Soot Developers. 2019. *Soot*. Retrieved June 29, 2019 from https://github.com/Sable/soot

[14] Bernhard Dorninger, Josef Pichler, and Albin Kern. 2015. Using static analysis for knowledge extraction from industrial User Interfaces. In *2015 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 497–500.

[15] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. 2003. Improving web application testing with user session data. In *25th International Conference on Software Engineering, 2003. Proceedings*. IEEE, 49–59.

[16] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 486–497.

[17] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 419–429.

[18] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. 2015. Sitar: Gui test script repair. *Ieee transactions on software engineering* 42, 2 (2015), 170–186.

[19] Zebao Gao, Chunrong Fang, and Atif M Memon. 2015. Pushing the limits on automation in GUI regression testing. In *2015 IEEE 26th international symposium on software reliability engineering*. IEEE, 565–575.

[20] Ceren Şahin Gebizli, Abdulhadi Kırkıcı, and Hasan Sözer. 2018. Increasing test efficiency by risk-driven model-based testing. *Journal of Systems and Software* 144 (2018), 356–365.

[21] Ceren Sahin Gebizli, Hasan Sözer, and Ali Özer Ercan. 2016. Successive refinement of models for model-based testing to increase system test effectiveness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 263–268.

[22] Google. 2019. *Android Debug Bridge (adb)*. Retrieved June 29, 2019 from https://developer.android.com/studio/command-line/adb

[23] Google. 2019. *Android Documentation*. Retrieved June 29, 2019 from https://developer.android.com/docs/

[24] Google. 2019. *Android UI Automator*. Retrieved June 29, 2019 from https://developer.android.com/training/testing/#UIAutomator

[25] Google. 2019. *Getevent*. Retrieved June 29, 2019 from https://source.android.com/devices/input/getevent

[26] Google. 2019. *UI/Application Exerciser Monkey*. Retrieved June 29, 2019 from https://developer.android.com/studio/test/monkey

[27] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 103–114.

[28] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 269–280.

[29] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 269–282.

[30] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 67–77.

[31] Robert M Keller. 1976. Formal verification of parallel programs. *Commun. ACM* 19, 7 (1976), 371–384.

[32] Mountainminds GmbH & Co. KG and Contributors. 2019. *JaCoCo - Java Code Coverage Library*. Retrieved Augest 7, 2019 from https://www.jacoco.org/jacoco/trunk/index.html

[33] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[34] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*. IEEE, 105–115.

[35] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. 1998. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*. Springer, 1–12.

[36] Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. 2016. Effectively manifesting concurrency bugs in android apps. In *2016 23rd Asia-Pacific Software Engineering Conference*. IEEE, 209–216.

[37] Qiwei Li, Chang Xu, Yepang Liu, Chun Cao, Xiaoxing Ma, and Jian Lü. 2017. CyanDroid: stable and effective energy inefficiency diagnosis for Android apps. *Science China Information Sciences* 60, 1 (2017), 012104.

[38] Xiujiang Li, Yanyan Jiang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2014. User guided automation for testing mobile apps. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1. IEEE, 27–34.

[39] Yi Liu, Jue Wang, Chang Xu, Xiaoxing Ma, and Jian Lü. 2018. NavyDroid: an efficient tool of energy inefficiency problem diagnosis for Android applications. *Science China Information Sciences* 61, 5 (2018), 050103.

[40] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lü. 2014. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering* 40, 9 (2014), 911–940.

[41] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[42] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.

[43] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[44] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 16–26.

[45] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. Citeseer, 260–269.

[46] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. 2016. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. 365–376.

[47] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. Sig-droid: Automated system input generation for android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering*. IEEE, 461–471.

[48] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering*. IEEE, 559–570.

[49] Bao N Nguyen and Atif M Memon. 2014. An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering* 40, 3 (2014), 216–234.

[50] Jacinto Reis and Alexandre Mota. 2018. Aiding exploratory testing with pruned GUI models. *Inform. Process. Lett.* 133 (2018), 49–55.

[51] Issue report. 2019. *Issue 128 of CallMeter*. Retrieved June 29, 2019 from https://github.com/felixb/callmeter/issues/128

[52] Issue report. 2019. *Issue 480 of AnyMemo*. Retrieved June 29, 2019 from https://github.com/helloworld1/AnyMemo/issues/480

[53] Issue report. 2019. *Issue 503 of AnyMemo*. Retrieved June 29, 2019 from https://github.com/helloworld1/AnyMemo/issues/503

[54] Issue report. 2019. *Issue 90 of Aard2*. Retrieved June 29, 2019 from https://github.com/itkach/aard2-android/issues/90

[55] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 27–37.

[56] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[57] Jue Wang, Yanyan Jiang, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu. 2018. AATT+: Effectively manifesting concurrency bugs in Android apps. *Science of Computer Programming* 163 (2018), 1–18.

[58] Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. E-greenDroid: effective energy inefficiency analysis for android applications. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*. 71–80.

[59] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 738–748.

[60] Renato Werneck, Joao Setubal, and Arlindo da Conceicao. 2000. Finding minimum congestion spanning trees. *Journal of Experimental Algorithmics* 5 (2000), 11–es.

[61] Xiangyu Wu, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2016. Testing android apps via guided gesture event generation. In *2016 23rd Asia-Pacific Software Engineering Conference*. IEEE, 201–208.

[62] Guangliang Yang and Jeff Huang. 2018. Automated generation of event-oriented exploits in android hybrid apps. In *Proc. of the Network and Distributed System Security Symposium*.

[63] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 2014. IntentFuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 531–536.

[64] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. 68–74.

[65] Chao Chun Yeh, Han Lin Lu, Chun Yen Chen, Kee Kiat Khor, and Shih Kun Huang. 2014. Craxdroid: Automatic android system testing by selective symbolic execution. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*. IEEE, 140–148.

[66] Xun Yuan and Atif M Memon. 2008. Alternating GUI test generation and execution. In *Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008)*. IEEE, 23–32.

[67] Xun Yuan and Atif M Memon. 2009. Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering* 36, 1 (2009), 81–95.

[68] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2017. RepDroid: an automated tool for Android application repackaging detection. In *2017 IEEE/ACM 25th International Conference on Program Comprehension*. IEEE, 132–142.

[69] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. 2017. Systematically testing background services of mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 4–15.