General and Practical Property-based Testing for Android Apps

Yiheng Xiong
Shanghai Key Laboratory of
Trustworthy Computing, East China
Normal University
China
xyh@stu.ecnu.edu.cn

Ting Su
Shanghai Key Laboratory of
Trustworthy Computing, East China
Normal University
China
tsu@sei.ecnu.edu.cn

Jue Wang
State Key Lab for Novel Software
Tech. and Dept. of Computer Sci. and
Tech., Nanjing University
China
juewang591@gmail.com

Jingling Sun
University of Electronic Science and
Technology of China
China
jingling.sun910@gmail.com

Geguang Pu
Shanghai Key Laboratory of
Trustworthy Computing, East China
Normal University. China
ggpu@sei.ecnu.edu.cn

Zhendong Su
ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Finding non-crashing functional bugs for Android apps is challenging for both manual testing and automated GUI testing techniques. This paper introduces and designs a *general* and *practical* testing technique based on the idea of property-based testing for finding such bugs. Specifically, our technique incorporates (1) a property description language (PDL) to allow specifying desired app properties, and (2) two exploration strategies as the input generators for effectively validating the properties. We implemented our technique as a tool named Kea and evaluated it on 124 historical bugs from eight real-world, popular Android apps. Our evaluation shows that our PDL can specify *all* the app properties violated by these historical bugs, demonstrating its generability for finding functional bugs. Kea successfully found 66 (68.0%) and 92 (94.8%) of the 97 historical bugs in scope under the two exploration strategies, demonstrating its practicability. Moreover, Kea found 25 new functional bugs on the latest versions of these eight apps, given the specified properties. To date, all these bugs have been confirmed, and 21 have been fixed. In comparison, prior state-of-the-art techniques found only 13 (13.4%) historical bugs and 1 new bug. We have made all the artifacts publicly available at *https://github.com/ecnusse/Kea*.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Property-based testing, Android app testing, Non-crashing functional bugs

ACM Reference Format:
Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3694986

1 INTRODUCTION

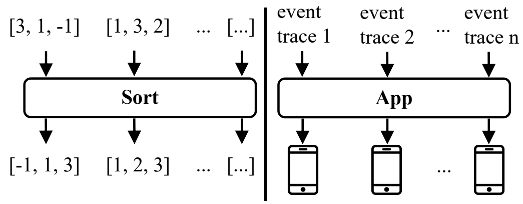
Mobile apps are ubiquitous and playing an important role in serving people's daily life [41]. However, it is reported that 88% of the users would abandon an app if they encounter bugs or glitches [5]. The app quality and reliability are therefore important for the competitive edge. Specifically, a recent comprehensive study [45] reveals that non-crashing functional bugs (*functional bugs* for short) account for the majority (about 65.4%) of the bugs in the apps. Indeed, some functional bugs may even lead to severe consequences in real life [26, 29, 33]. Thus, effectively validating the functional correctness of the apps is crucial for their success.

Challenges. Manual testing (*e.g.*, *manually* writing GUI tests or interacting with the apps) is the most widely-used practice to validate the functional correctness [16, 20]. In this process, the testers compare expected app behaviors (*e.g.*, encoded in the assertions of GUI tests) with the actual app behaviors to find inconsistencies (*i.e.*, functional bugs). However, manual testing is usually expensive, small-scale, and inadequate — exercising only the *happy paths* of app functionalities, thus likely missing non-trivial functional bugs. Despite automated GUI testing techniques [8, 24, 34, 35, 44] can automatically explore the apps and thus reduce manual testing cost, they cannot find functional bugs due to the lack of test oracles [3].

To *automatically* find functional bugs, some novel techniques like Genie [36] and Odin [43] propose automated oracles. For example, Genie proposes the *independent view property*, one likely-hold metamorphic relation [6, 7, 32] in the apps, as the automated oracle. Odin uses differential analysis to automatically mine the abnormal app behaviors from a large number of GUI traces based on the classical oracle of "*bugs as deviant behaviors*" [10]. However, these techniques are limited in generability and practicality. First, the oracles of Genie and Odin can only capture limited portions of functional bugs. For example, only 29.5% of the functional bugs fall into the scope of Genie's oracle (*cf.* Section 1 in [36]). Second, both Genie and Odin suffer from the high false positive rates of 59% and 68%, respectively (*cf.* Section 5.4 in [36] and Section 5.5

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694986



(a) Property-based testing for function *sort*

(b) Property-based testing for Android apps

Figure 1: Conceptual comparison between property-based testing for traditional program and Android apps.

in [43]) because the proposed oracles are *heuristic*. It incurs a lot of manual overhead of filtering false positives.

The preceding situation underlines the *challenges* of validating the functional correctness of the apps. *On the one hand*, manual testing comes with the human knowledge of expected app behaviors (*i.e.*, the oracles) but is limited by the high cost and inadequacy of manually creating GUI tests. *On the other hand*, automated GUI testing techniques excel at automatically exploring the apps but are struggling with the availability and effectiveness of the oracles.

High-level idea. To tackle the preceding challenges, this paper introduces and designs a general and practical testing technique based on the idea of *property-based testing* (PBT) [9, 11]. Our *key* insight behind this technique is to synergistically combine (1) the strengths of the human knowledge of expected app behaviors and (2) the abilities of automated GUI testing to explore the apps.

Specifically, classic property-based testing validates whether a piece of program satisfies some desired properties by automatically generating a large number of random inputs. Note that the properties for PBT are typically *manually* specified [12]. For example, in Figure 1(a), for a function `sort` which takes as input an integer array `arr` and returns the sorted `arr` with its elements in the ascending order, one of its desired property is $\texttt{arr}[i] \leq \texttt{arr}[j]$ ($\forall i$, j, $0 \leq i \leq j \leq L-1$, L is the array length of `arr`). PBT would generate a number of arrays with different sizes and elements (*e.g.*, [3, 1, -1], [1, 3, 2]) to validate whether the property holds.

At the high-level, the idea of our testing technique is similar (shown in Figure 1(b)): we aim to validate whether an app satisfies the desired property by automatically generating a large number of random event traces. These event traces drive the app to output different app states (like GUI pages). Based on these states, we can check whether the desired property holds. However, instantiating the preceding idea in the settings of GUI applications as Android apps is not straightforward. We face two technical challenges: (1) how to specify the desired app properties covering general functional bugs, *and* (2) how to effectively explore the app (*i.e.*, generating GUI tests) to validate the properties. Despite some work like PBFDroid [37] explores property-based testing in this setting, they are limited to specific bug types (discussed in Sections 5.5 and 7).

Our approach. To facilitate specifying app properties, we design a property description language (PDL) in a flexible and general manner. Specifically, in this PDL, a property is represented in the form of precondition, interaction scenario, and postcondition, which can cover general app functionalities (detailed in Section 3.2). To effectively explore the app for validating the property, we design two UI exploration strategies (detailed in Section 3.3): (1) *random exploration* and (2) *main path guided exploration* strategies. Specifically, the random exploration strategy randomly explores the app in a wide exploration space. On the other hand, the main path guided exploration strategy is inspired by the typical process of manual testing — a tester usually follows a *main path* (typically the happy path) from the app entry to reach the target app functionality for functional testing. Our key insight is that such a main path can be easily obtained as a by-product when a user specifies an app property, and thus can effectively guide the exploration of alternative paths along the main path to validate functional correctness. Moreover, when multiple properties of an app are available, the two exploration strategies can validate multiple properties together. It improves the testing efficiency. Meanwhile, the interaction scenarios of these properties provide a partial model of the app, thus enabling these two strategies to explore more diverse and deeper app states (detailed in Section 3.3.3).

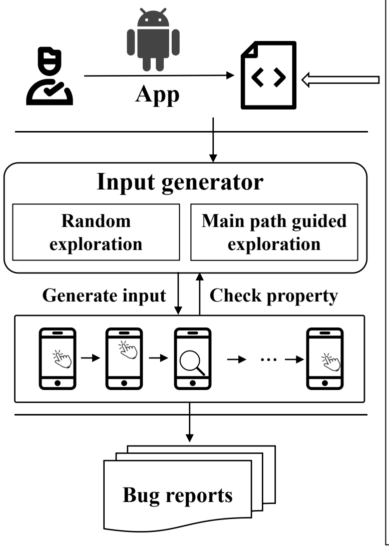
Evaluation and results. We implemented our testing technique as a tool named Kea. To evaluate Kea, we collected all the historical functional bugs from eight popular Android apps without any cherry picking. We obtained 124 historical functional bugs which are reproducible at the time of our study. Our evaluation shows that *all* the 124 desired app properties violated by these historical bugs can be successfully specified by our PDL. It indicates that Kea is *general* for finding functional bugs. On the other hand, on the 97 historical bugs in scope[1], Kea successfully found 66 bugs (68.0%) and 92 bugs (94.8%) under the random and main path guided exploration strategies, respectively. Moreover, the main path guided strategy is more efficient than the random one — the former is nearly 4X faster than the latter in terms of the average time of finding functional bugs.

Further, we applied Kea to validate all these properties on the latest versions of these apps on which those historical functional bugs have already been fixed. Despite these fixes, Kea still found 25 new functional bugs, all of which have been confirmed, and 21 have already been fixed by app developers. The result shows that Kea can validate functional correctness more thoroughly. In comparison, prior state-of-the-art techniques can find only 13 historical bugs and 1 new bug. These results show that the practicability of Kea.

In summary, this paper has the following contributions:

- At the conceptual level, we introduce a general and practical property-based testing technique for validating the functional correctness of Android apps.
- At the design level, we design (1) a property description language to allow specifying app properties and (2) the two exploration strategies to effectively validate app properties.
- At the technical level, we have instantiated our design and idea as a property-based testing tool Kea for Android apps.
- At the empirical level, we have demonstrated the generability and practicality of Kea based on a dataset of 124 historical functional bugs. Kea further successfully found 25 new functional bugs. We have made all the artifacts publicly available at *https://github.com/ecnusse/Kea* for replication and facilitating further research.

[1] We excluded 27 historical bugs because the bug-triggering conditions are too specific and are not the focus of our work. We give more details in RQ2's setup in Section 5.1.



(a) Overview of Kea

```
1   @precondition(
2       findWidget(id="tag_button").exists() and
3       findWidget(id="content",textContains="#").exists()
4   )
5   @property
6   def tag_removement():
7       # get the text from the note's content
8       original_content = findWidget(id="content").get("text")
9       # click to open the tag list
10      findWidget(id="tag_button").click()
11      # get the tags in current note
12      checked_tag = []
13      for tag in findWidget(id="tag_name"):
14          # only checked tag is in current note
15          if tag.get("checked") == true:
16              checked_tag.append(tag)
17      # random select a tag
18      selected_tag = random.choice(checked_tag)
19      # get the name of the tag
20      selected_tag_name = '#' + selected_tag.get("text")
21      # click to uncheck the selected tag
22      selected_tag.click()
23      findWidget(text="OK").click()
24      # get the updated content after removing the tag
25      updated_content = findWidget(id="content").get("text")
26      # get the expected content after removing the tag
27      expected_content = origin_content.replace(selected_tag_name, "")
28      # the tag should be removed in the content and the updated content should be the
29      # same as expected content
30      assert not findWidget(id="content", textContains=selected_tag_name).exists() and
31                 updated_content == expected_content
```

(b) Example property of the functionality *removing note tags* in OmniNotes

Figure 2: Overview of Kea

2 OVERVIEW AND EXAMPLE

Overview. Figure 2(a) shows the overview of our property-based testing technique and the implemented tool Kea. Given an app and one property of interest (specified by a human tester), Kea automatically explores the app to validate the property. If the property is violated, Kea will output a bug report, which contains some GUI tests illustrating the violation. Specifically, to support the application of property-based testing, Kea provides (1) a Python-based property description language to help users specify the desired app properties, and (2) two exploration strategies to generate a large number of GUI tests for validating the properties. Note that an app property characterizes the expected behaviors of specific app functionality. In the following, we use an example to illustrate this testing technique.

Example. Figure 3 shows *OmniNotes*, a popular note-taking app with 2.7k stars on GitHub. Figure 3(a) shows one major app feature: *a user can create a note, add a note tag, and remove the tag*. Specifically, a user can create a note by clicking the floating action button on page (1) to create an (empty) note (page (2)), and add some texts with a note tag (“read a book #Tag1” in this case, page (3)). Note that *OmniNotes* explicitly stores the tag in the note content (“#Tag1” in this case). To remove the note tag, a user can click the *note-tag* button on the top-right of page (3) to open the tag list (page (4)), uncheck the tag “Tag1” in the tag list, and click "OK" on page (5). On page (6), we can see that the functionality of *removing the note tag* works: after unchecking the note tag “Tag1” in the tag list, the text “#Tag1” is correctly removed from the main text of the note.

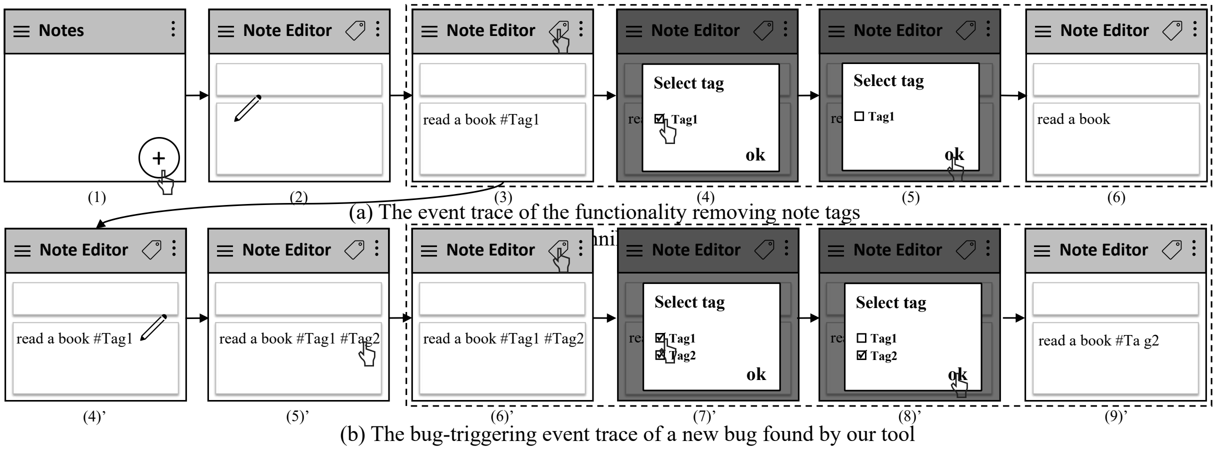
Since *removing note tags* is a basic functionality of *OmniNotes*, we are interested in validating its correctness by specifying the desired property in Figure 2(b). The property defines the precondition (*when we could remove the tag*, lines 1-4), the interaction scenario (*how we could remove the tag*, lines 8-27), and the postcondition (*what are the expected results after removing the tag*, lines 30-31). Specifically, the postcondition is defined by an `assert` statement which checks whether the tag is removed from the note content and the note content (excluding the removed tags) remains unchanged.

When we applied Kea to validate this property, a new functional bug was quickly found. Figure 3(b) shows one of bug-triggering event traces. If an app user creates a note (pages (1)~(2)), adds some texts (*e.g.*, “read a book”) with one tag (“Tag1”) to the note content (pages (2)~(3)), returns back to the note list and reopens the current note (pages (3)~(4)’), adds the second tag (“Tag2”) to the note content (pages (4)’~(5)’), clicks the second tag ("Tag2") on page (5)’, and removes the tag “Tag1” (pages (6)’~(8)’). We can see that the tag "Tag1" is successfully removed, but the tag “Tag2” is erroneously changed to “Ta g2” (page (9)’), which violates the postcondition. Even worse, if we open the tag list on page (9)’, the original tag “Tag2” in the tag list becomes “Ta”. Note that the prior tools like Genie and Odin cannot find this bug because of their limited automated oracles. PBFDroid can check the property of data manipulation functionalities (like the functionality of *removing note tags* in this case), but it still cannot find this bug. Because its property only checks whether the note’s tag (“Tag1”) is removed.

3 DESIGN OF KEA

3.1 Preliminary

Android apps are GUI-centered and event-driven. When an Android app $\mathcal{A}$ runs on the device, its state s can be abstractly represented by its runtime GUI layout ℓ (s is thus named as a GUI state). A GUI layout ℓ is a tree, in which each node is a GUI widget w (e.g., a `button` or a `textview`). A user usually interacts with the app by sending *events* to a GUI widget. An event $e = \langle t, w, d \rangle$, where $e.t$ denotes the event’s type (*e.g.*, "click", "long click"), $e.w$ denotes the receiver widget, and $e.d$ denotes the associate data (*e.g.*, the texts needed in the `EditText` widget). In addition, a user can also send non-GUI events to $\mathcal{A}$, such as rotating the screen.



(a) The event trace of the functionality removing note tags

(b) The bug-triggering event trace of a new bug found by our tool

Figure 3: *OmniNotes*: (a) shows the functionality of removing note tags, (b) shows a new bug found by Kea.

```
ϕ          ::=  @precondition(Pred)
                @property
                def prop():
                   Stmts
─────────────────────────────────────────────────────────────────────────
Stmts      ::=  Stmt Stmts | ϵ
Stmt       ::=  widget.event
             |  assert  Pred /* A postcondition */
             |  var = widget | var = attr
             |  if Pred: Stmts else: Stmts | ... /* All conditional constructs */
             |  For var in var: Stmts | ... /* All loop constructs */
             |  strval.replace(var, strval) | ... /* All function callings */
             |  var = var + var | ... /* All arithmetic statements */
             |  ... /* All other statements supported by Python*/
Pred       ::=  Clause | not Pred | Pred and Pred | Pred or Pred
Clause     ::=  widget.exists()
             |  attr relop numval | attr eqop numval
             |  attr eqop strval | attr eqop boolval
             |  var relop numval | var eqop numval
             |  var eqop strval | var eqop boolval
─────────────────────────────────────────────────────────────────────────
widget     ::=  findWidget(crit)
                /* An API returning a widget of the current GUI layout
                that matches certain criteria */
crit       ::=  attr_id=value, crit | ϵ
attr       ::=  widget.get("attr_id")
attr_id    ::=  id | className | description | text | ...
                /* An attribute of a widget */
event      ::=  click() | long_click() | set_text(strval) | scroll().to(crit)
             |  rotate_screen() | ...
                /* An API sending a specified event to the app */
var        ::=  original_content | checked_tag | ... /*A Python variable*/
relop      ::=  > | < | ≤ | ≥
eqop       ::=  = | ≠
value      ::=  strval | numval | boolval
strval     ::=  "OK" | "tag_name" | ... /*A concrete string value*/
numval     ::=  0 | 1 | 0.1 | ... *A concrete numeric value*/
boolval    ::=  True | False
```

Figure 4: Core syntax of our PDL

Based on the definitions of state abstraction and event, we can define an execution trace $\tau = \mathtt{Execute}_{\mathcal{A}}(E)$, given a sequence of events $E = [e_1, e_2, \ldots, e_n]$. Executing $\mathcal{A}$ with a sequence of events $E = [e_1, e_2, \ldots, e_n]$ yields an execution trace τ. τ can be denoted as $\tau = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} s_n$ or $\tau = s_0 \overset{E}{\rightsquigarrow} s_n$. Without ambiguity, we also denote τ as a sequence of GUI states $[s_0, s_1, \ldots, s_n]$, where s_0 is the state of $\mathcal{A}$ before sending e_1, and $\tau = [s_0]$ if $E = []$.

3.2 Specifying Properties

3.2.1 High-level property definition. To achieve property-based testing, we need to specify the desired property of an app functionality. We observe that in manual testing, to exercise an app functionality, a user needs to (1) navigate to a starting GUI state s where the functionality is ready for execution, (2) interact with the app to perform the functionality by executing a sequence of events E, and (3) observe whether the ending GUI state s' is expected.

To this end, we design the property ϕ in the form of $\phi = \langle P, I, Q \rangle$, where (1) P is a *precondition* which defines when or where we could perform the app functionality, (2) I is an *interaction scenario* which defines how to perform the functionality, and (3) Q is a *postcondition* which defines what are the expected results after the functionality.

3.2.2 Property Description Language. To facilitate specifying general properties in the form of $\phi = \langle P, I, Q \rangle$, we design a *property description language* (PDL). It is a domain specific language based on Python. Figure 4 shows the core syntax of our PDL, which is a superset of the syntax of Python. In our PDL, the interaction scenario I and the postcondition Q are specified in a Python function annotated with `@property`. The precondition P is specified in the function's annotation `@precondtion`. We give relevant definitions below and illustrate our PDL with the property in Figure 2(b).

Precondition and Postcondition. The precondition P and the postcondition Q are defined as the *predicates* over the starting and ending GUI states s and s', respectively. A predicate p over GUI states is a function $p : S \longrightarrow \{\top, \bot\}$. We denote $s \models p$ if an app's GUI state s satisfies the predicate p, i.e., $p(s) = \top$.

In our language, the precondition is specified in the annotation `@precondtion`, while the postcondition is specified in an `assert` statement. In Figure 4, the rule of *Pred* shows that a predicate could be a first-order clause *Clause* or a number of clauses connected by logical operators (`and`, `or`, and `not`). A clause *Clause* can check whether a specific widget exists on the current GUI layout *or* the value of a widget's specific attribute (*e.g.*, `id`, `text`). To support more general predicates, our language provides an API named as *widget* to obtain a widget (from the current GUI layout) which matches some criterion *crit*. The *crit* specifies the expected values of some attribute *attr_id* (*e.g.*, `id` or `text`) of the obtained widget. For instance, in Figure 2(b), the precondition checks whether a

Algorithm 1: Random Exploration

```
1  Function main($\phi = \langle P, I, Q \rangle$):
2      while not timeout do
3          cleanApp ();
4          restartApp ();
5          for $i \leftarrow 0$ to MAX_EVENT_NUMBER do
6              $s \leftarrow$ getCurrentState();
7              if $s \models P \wedge$ random() $< 0.5$ then
8                  checkProperty($I, Q$) ;
9              else
10                 $e \leftarrow$ generateRandomEvent($s$);
11                 sendEventToApp ($e$);
```

widget whose `id` is `tag_button` exists (Line 2), and whether a tag exists in the note's content whose `id` is `content` (Line 3); the postcondition (Lines 30-31) checks whether the removed tag still exists in the note content and the updated content is expected.

Interaction Scenario. The interaction scenario I defines how to interact with the app to perform the target functionality. We denote the execution trace of I as $s_0 \overset{\text{I}}{\rightsquigarrow} s_n$, where s_0 and s_n are the starting and ending GUI states, respectively.

Following the syntax of Python, our PDL allows the users to define a sequence of statements as the interaction scenario. In Figure 4, the rule of *event* shows that our PDL supports generating and sending various events such as `click()`, `long_click()`, and `rotate_screen()`. Since our PDL is a superset of Python, the user can utilize all features in Python, *e.g.*, conditional statements, loops, and function calls, to facilitate specifying the interaction scenarios with complicated logic. For example, in Figure 2(b), the interaction scenario includes such an event sequence E: (1) open the tag list (Line 10), (2) uncheck a selected tag (Line 22) and (3) click "OK" to remove the tag (Line 23). Specifically, our PDL allows to find which tags are checked in the current note via a loop (Lines 13-16).

Additional features of our language. As the rule of *Stmts* and *Stmt* in Figure 4 shows, our PDL allows users to specify multiple `assert` statements in the function body. Assume a GUI state s_0 satisfies the precondition of a property $\phi = \langle P, I, Q \rangle$ and the interaction scenario I yields a sequence of GUI states $[s_0, s_1, s_2, s_3]$. Our PDL allows users to place the `assert` statements following any GUI state (say s_1) to define the postcondition Q. In other words, Q is not limited to be placed after the ending state s_3. Moreover, we can use conditional statements, loops, and function calls to specify more complicated postconditions. The users can also specify multiple properties in the function body.

Property-based Testing for Android Apps A property $\phi = \langle P, I, Q \rangle$ is a tuple where the precondition P and postcondition Q are both predicates over GUI states and I is an interaction scenario. An app's GUI state s satisfies ϕ, denoted by $s \models \phi$, iff

$$(s \models P \wedge s \overset{\text{I}}{\rightsquigarrow} s') \Rightarrow s' \models Q.$$

An app $\mathcal{A}$ satisfies the property, denoted by $\mathcal{A} \models \phi$, iff for every GUI state $s \in S$ of $\mathcal{A}$, $s \models \phi$. If we find some GUI state $s \not\models \phi$, we find a functional bug.

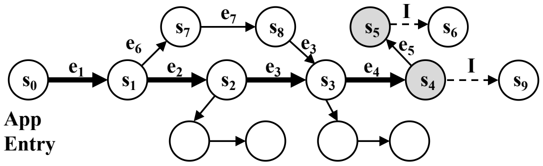


Figure 5: Example for main path guided exploration strategy

3.3 Validating Properties

Given a specified property $\phi = \langle P, I, Q \rangle$, our tool automatically explores the app's GUI state space for property validation. Specifically, We design two UI exploration strategies: (1) *random exploration*, and (2) *main path guided exploration*. The random exploration strategy aims to generate random events to explore the GUI state space in a wide range and check the property if the precondition is satisfied. On the other hand, we observe that when a user specifies an app property, the user would follow a main path (typically the happy path) from the app entry to reach the target app functionality. Such a main path can be easily obtained as a by-product and leveraged to guide the exploration. Accordingly, we design a guided exploration strategy that utilizes such main paths.

3.3.1 *Random Exploration.*

Algorithm 1 presents the random exploration strategy. The algorithm takes the property $\phi = \langle P, I, Q \rangle$ as input and iterates for multiple rounds trying to reach GUI states where the property can be checked until the time budget runs out (Lines 2–11). At each round, it restarts the app to reach an initial GUI state (Lines 3–4), and generates random events to explore the GUI state space (Lines 10–11). At each GUI state, it checks whether the precondition P is satisfied (Line 7). If so, it checks the property at the current GUI state s with 50% probability. If s needs to be checked, the interaction scenario I is used to execute the app, and the postcondtion Q is checked for property violations (Line 8). We check the property with 50% probability to balance the property checking and the UI exploration, which allows us to reach deep GUI states. For example, the bug in Figure 3(b) is found because we explore more GUI states (*e.g.*, pages (4)', (5)') even if the precondition is satisfied on page (3). If we always check the property when the precondition is satisfied, the property will always be checked once the app reaches page (3). It hinders the exploration to reach the deep GUI state where the bug can be found.

3.3.2 *Main Path Guided Exploration.*

Algorithm 2 presents the main path guided exploration strategy. This strategy takes input a property $\phi = \langle P, I, Q \rangle$ and a main path in the form of an event sequence $E = [e_1, e_2, \ldots, e_n]$. Our insight is that the main path can be obtained by-product when a user specifies a property. This main path provides the guidance of how to reach a GUI state where the app property is ready for checking. In other words, when the main path is executed from the app entry, we can obtain a sequence of GUI states $[s_0, s_1, \ldots, s_n]$, where $s_n \models P$. Moreover, exploring the states close to the main path could give a higher chance of reaching GUI states satisfying the precondition. In Figure 3(a), the main path includes the two events driving the app from page (1) to page (3).

Given the property $\phi = \langle P, I, Q \rangle$ and the event sequence of the main path $E = [e_1, e_2, \ldots, e_n]$, The guided exploration traverses backward along the main path, and explores GUI states close to the main path (Lines 3–12). Specifically, it iterates backward from e_n to e_1 (Line 8). For each event e_i where $0 < i \leq n$, it sends the prefix

Algorithm 2: Main Path Guided Exploration

```
1  Function main($\phi = \langle P, I, Q \rangle, E = [e_1, e_2, \ldots, e_n]$)
2      $i \leftarrow n$;
3      while not timeout do
4          if $i > 0$ then
5              for $e \leftarrow e_1$ to $e_i$ do
6                  sendEventToApp ($e$);
7          explore ($\phi, E$);
8          $i \leftarrow i - 1$;
9          if $i = -1$ then
10             cleanApp ();
11             $i \leftarrow n$ ;
12         restartApp ();
13 Function explore ($\phi = \langle P, I, Q \rangle, E$):
14     for $t \leftarrow 1$ to MAX_STEP do
15         $s \leftarrow$ getCurrentState();
16         if $s \models P \wedge$ random() $< 0.5$ then
17             checkProperty($I, Q$) ;
18         else
19             $e \leftarrow$ generateRandomEvent($s$);
20             sendEventToApp ($e$);
21     $e_j \leftarrow$ canGoToSatisfyPrecondition($E$) ;
22     if $e_j \neq null$ then
23         goToSatisfyPrecondition ($e_j, E$);
24     $s \leftarrow$ getCurrentState();
25     if $s \models P$ then
26         checkProperty($I, Q$) ;
27 Function canGoToSatisfyPrecondition($E = [e_1, e_2, \ldots, e_n]$):
28     $s \leftarrow$ getCurrentState();
29     for $e_j \leftarrow e_n$ to $e_1$ do
30         if $e_j.widget \in s$ then
31             return $e_j$;
32     return null;
33 Function goToSatisfyPrecondition($e_j, E = [e_1, e_2, \ldots, e_n]$):
34     for $e \leftarrow e_j$ to $e_n$ do
35         $s \leftarrow$ getCurrentState();
36         if $e.widget \in s$ then
37             sendEventToApp ($e$);
```

$[e_1, e_2, \ldots, e_i]$ to the app to reach a GUI state s_i of the main path (Lines 5–6). Next, it explores GUI states close to s_i, trying to find states satisfying P of the property (Line 7). Note that after s_1, we also explore GUI states close to s_0 by sending no event from E (Line 4). Multiple traverses along the main path can be conducted if the time budget allows it, we clean the app data after exploring every state on the main path (Lines 9–11).

The exploration starting from a GUI state of the main path resembles the random exploration strategy 1 (Lines 14–20). Specifically, it checks whether $s \models P$ at each visited GUI state s (Lines 15–16). If so, it checks the property at s by coin flipping (Line 16). Otherwise, a random event is generated and sent to reach another state (Lines 19–20). Such a process iterates for MAX_STEP times (Line 14).

After the exploration, we try to get to the state that satisfies the precondition (Lines 21–23). The insight is that the random exploration starting from a GUI state of the main path may change the internal state of the app, and getting to the state that satisfies precondition may further exhibit different behaviors of the app [36]. To do so, we search for the latest event e_j in E that can be sent at the current GUI state (Lines 28–32). If e_j exists, we try to send the suffix $[e_j, e_{j+1}, \ldots, e_n]$ of E (Lines 34–37). Finally, we try to check the property again (Lines 24–26).

Example. Figure 5 illustrates the guided exploration strategy. Let s_0 be the starting GUI state of the app entry, the main path be $E = [e_1, e_2, e_3, e_4]$ and the property be $\phi = \langle P, I, Q \rangle$. In 1st iteration, this strategy would send all the events of E and reaches s_4 (by definition $s_4 \models P$). The states satisfying P are marked in grey. It then starts the random exploration from s_4. Assume it generates e_5 on s_4 and reaches s_5. Suppose $s_5 \models P$, the strategy may decide to execute I for checking Q on the ending state s_6. Assume $s_6 \models Q$, no property violation is found. At this time, suppose the number of executed events of e_5 and I exceeds MAX_STEP, the strategy would stop the random exploration and try to navigate to follow the main path for satisfying P. Suppose no event in E can be sent on s_6, the strategy would give up this navigation, and start the 2nd iteration. In the 2nd and 3rd iterations, it would start from s_3 and s_2, respectively, and do a similar process like the 1st iteration. In the 4th iteration, it starts from s_1. Assume it explores $s_1 \xrightarrow{e_6} s_7 \xrightarrow{e_7} s_8$ by generating two random events e_6 and e_7, but neither $_7$ or s_8 satisfies P. It then tries to navigate back to follow the main path for satisfying P. Suppose it finds that e_3 can be sent on s_8. It would send e_3 and e_4 sequentially to try to follow the main path. Suppose it reaches s_4 satisfying P, it would execute I and check Q on the ending state s_9. If the property is violated, we find a bug.

3.3.3 Validating Multiple Properties. The random and main path guided exploration strategies by default validate one property of an app at one run. When multiple properties of an app are available, these two strategies can validate any subset of these properties together. *One benefit* is that we can improve the efficiency of validating properties. *Another benefit* is that the interaction scenarios of multiple properties provide a partial model of the app. This partial model enables us more likely to reach deeper app states during testing. For example, the property in Figure 2(b) can only be validated when its precondition (*i.e.*, a tag exists in the note's content) is satisfied. In this case, this precondition is more likely to be satisfied if another property's interaction scenario is adding a tag for a note. Otherwise, it might be difficult for the exploration strategies alone to achieve the effect of adding a tag for a note.

Specifically, to validate multiple properties together, the random strategy (Algorithm 1) would check whether multiple properties' preconditions are satisfied at Line 7, and *randomly* select one property for checking at Line 8. The main path guided exploration strategy (Algorithm 2) would *randomly* select one property as the target, and perform guided exploration along its main path. When every state on this main path has been explored, this strategy would *randomly* select another property as a new target. In addition, this strategy would *randomly* select a property for check when multiple properties' preconditions are satisfied at Lines 16-17 and 25-26.

4 IMPLEMENTATION

Kea is built on top of DroidBot [19], a popular open-source automated GUI testing tool. Specifically, we implemented the random and main path guided exploration strategies in the input generator module of DroidBot. Uiautomator2 [42] is used to support specifying app properties and parsing GUI layout information. Kea currently supports the following UI and system events: `click`, `long click`, `set text`, `swipe`, `scroll`, `rotate screen`, and `naviagtion` (*e.g.*, `back`, `home`). We use the random text generator in Hypothesis [25] to support generating random input texts. For the precondition and postcondition, our property description language currently supports checking the widget attributes (*e.g.*, `id`, `className`, `text`) which are parseable by Uiautomator2.

5 EVALUATION

Kea is a property-based testing technique which requires manually specified app properties for validation. To this end, we decided to evaluate it based on a dataset of historical functional bugs of real-world Android apps. This setup has two important benefits. First, the historical bugs could indicate the affected app functionalities and the expected app behaviors, from which we can identify the desired app properties in an objective and unbiased manner[2]. Second, the historical bugs enable us to quantitatively analyze the generability and practicality of Kea (*e.g.*, how many properties could be specified and how many historical bugs could be found). To this end, we evaluate Kea by investigating the following research questions:

- **RQ1**: How general is Kea in specifying the app properties violated by the historical functional bugs? How complex are these specified properties?
- **RQ2**: How many of these historical functional bugs can be found by Kea, given the specified properties?
- **RQ3**: Can Kea find new functional bugs on the latest versions of these apps, given the specified properties?
- **RQ4**: How many historical or new functional bugs can be found by prior functional testing techniques, compared to Kea?

RQ1 aims to evaluate the generability of Kea (*i.e.*, whether Kea can be applied to different functional bugs) and the complexity of specifying the properties. **RQ2** and **RQ3** aim to evaluate the practicality of Kea in finding known and new functional bugs. **RQ4** aims to compare Kea with prior relevant testing techniques in finding functional bugs.

5.1 Evaluation Setup and Method

App subjects. We selected eight representative, open-source apps from prior work in functional testing of Android apps [36, 37, 43, 45]. We excluded the other apps because (1) they have similar app features with the selected ones, or (2) many of their old versions cannot be run anymore (which prevents us from evaluating the historical bugs). Table 1 gives the details of these eight apps, where *App Feature* denotes the major app feature, and *#Installations* and *#Stars* give the numbers of installations on Google Play and stars on GitHub, respectively. Most of these apps are popular.

Collecting historical functional bugs. We crawled *all* the issues reported in the issue repositories of the selected apps. Specifically, we filtered the issues which were explicitly labeled as *bugs* and have already been *closed*. We focused on the closed issues because such issues have already been fixed by developers and are more likely reproducible. In this process, we note that *AnkiDroid* and *AntennaPod* respectively have more than 500 closed issues. Since it is not feasible for us to examine all these issues, we constrained our efforts on *all* the closed issues that were reported within the recent three years. For these closed issues, we manually examined each of them and excluded the invalid ones like duplicated, mislabeled, feature requests, crashing bugs, cosmetics bugs (*e.g.*, the issues related to the colors), and non-functional bugs (*e.g.*, performance or energy issues). Then, we manually tried to reproduce each of the remaining issues and excluded the issues which were not reproducible anymore. Specifically, we followed the reproduction steps and other information (*e.g.*, buggy app versions, Android OS versions) to reproduce the issue. Finally, we get 124 reproducible historical functional bugs from these eight apps. In Table 1, *#Historical bugs* gives the numbers of historical bugs of these apps.

Table 1: Apps used in our experiment (K=1,000, M=1,000,000)

App Name	App Feature	#Installations	#Stars	#Historical Bugs
OmniNotes	Note Manager	10~50M	2.7K	20
Markor	Text Editor	10~50M	3.3K	16
SimpleTask	Task Manager	10~50K	544	12
AmazeFileManager	File Manager	1~5M	5.1K	16
ActivityDiary	Activity Recorder	1~5K	72	6
AntennaPod	Podcast Manager	1~5M	5.8K	14
AnkiDroid	Flashcards Manager	10~50M	7.9K	28
Transistor	Radio Listener	10~50K	431	12

Evaluation method of RQ1. RQ1 aims to investigate whether all the app properties violated by the 124 historical bugs can be specified by our PDL in Kea. Since these historical bugs are not cherry-picked, if all the app properties can be specified, it could demonstrate the generability of Kea.

To this end, given a historical bug, two co-authors of this paper independently (1) reviewed the corresponding bug report to identify the affected app functionality, (2) understand the expected app behaviors to identify the desired app property, and (3) specify the property in our PDL. Specifically, when specifying the property, they follow one *important* guideline — *the property should be as general as possible to mimic the human knowledge on app features without knowing the bug*. In other words, we need to abstract away the bug-specific information (*e.g.*, specific events or text inputs) from the known bug-triggering event trace but ensure that the property should still be able to reveal the bug when the trace is given. For example, one historical bug of *OmniNotes* (Issue #786) is: when a user removes a tag from a note, some characters (*e.g.*, “\n”, “;”, “-”) in the note content will be unexpectedly removed. From this historical bug, we first identify the affected app functionality, *i.e.*, removing note tags in Figure 3(a). Then, we understood the expected app behaviors to identify the desired property of this functionality — *when a user removes a tag in a note, the tag should be removed and the note’s content should not be altered* (Figure 2(b) specifies this property). Note that we abstracted away the bug-specific information (*i.e.*,“\n”, “;”, “-”) from the property.

To ensure all the properties are correct and general, the two co-authors cross-checked their specified properties and updated the properties if there were inconsistencies. Afterwards, they explained each historical bug and presented the corresponding property to the other two co-authors for reaching consensus. Additionally, We conducted a sanity check on these properties to ensure their correctness. For each property, we followed the original bug-triggering trace from the app entry to the app state satisfying the property’s precondition and executed the interaction scenario to confirm that the bug could be triggered. Finally, we got 124 carefully-validated properties, which are made publicly available at *https://github.com/ecnusse/Kea*.

We compute the complexity of each property in the following way: (1) for the preconditions and postconditions: the complexity is represented by the sum of the number of clauses (defined in Figure 4) and the number of logical operators (*e.g.*, *and*, *or*, and *not*), *and* (2) for the interaction scenario: the complexity is represented by two metrics: the number of events and the number of code lines.

[2] We tried to identify the app properties from the open-source apps’ public documentations, which however are too simple and incomplete to be useful.

For example, in Figure 2(b), the complexities of the precondition and postcondition are 3 (two clauses and one logical operator at Lines 2-3) and 4 (two clauses and one logical operator at Lines 30-31), respectively. The complexity of the interaction scenario is represented by 3 events (Lines 10, 22, and 23), and 12 lines of code.

Evaluation method of RQ2. RQ2 aims to evaluate how many historical bugs can be found by Kea. Specifically, to achieve a fair evaluation, we examined all the 124 historical bugs, and excluded 27 bugs from this evaluation. Because the bug-triggering conditions of these 27 bugs are too specific and are not the focus of this paper: (1) domain-specific text inputs (20 bugs), *e.g.*, adding a specific URL of the radio station in *Transistor*'s Issue #9, (2) specific system settings (5 bugs), *e.g.*, setting Turkish as the system language in *Markor*'s Issue #1443, (3) human knowledge (1 bug), *e.g.*, giving a password to lock the notes in *OmniNotes*'s Issue #598, and (4) specific timings of events (1 bug), *i.e.*, waiting until the timing reminder is triggered in *OmniNotes*'s Issue #381. These specific bug-triggering conditions were also identified as the common challenges of input generation in testing apps [4]. It is an orthogonal problem and could be mitigated by those LLM-based techniques [21, 22].

Thus, we focus on finding the remaining 97 historical bugs under the random and guided exploration strategies. We allocated 6 hours for finding each bug per strategy. To mitigate the randomness, we repeated the experiment *three* times for each bug and counted the average time if the bug was found. For the random exploration strategy, we evaluated it with five different configurations of `MAX_EVENT_NUMBER` (the maximum number of events allowed in each test, *i.e.*, 20, 40, 60, 80, 100) in Algorithm 1. For the guided exploration strategy, the maximum number of random events (`MAX_STEP` in Algorithm 2, Line 14) is set as 20, and the main path is set as the shortest event trace starting from the app entry and reaching the app state satisfying the precondition of the corresponding property. We examined all the bugs reported by Kea to confirm whether a historical bug was found.

Evaluation method of RQ3. RQ3 aims to evaluate whether Kea can find new functional bugs on the latest app versions given the specified properties. To this end, we manually examined whether the functionality *w.r.t.* each property from RQ1 still exists on the latest app versions at the time of our study. If so, we checked and updated the property to fit the latest app version if needed. In this process, we excluded 8 properties which are not supported anymore. Thus, RQ3 was evaluated on the 116 properties. Specifically, we allocated 24 hours for validating each property under the random and guided strategies, respectively. It took 24×116 machine hours per strategy. Additionally, we also validated all properties of one app together under the random and guided strategies, respectively (*cf.* validating multiple properties in Section 3.3.3). We did not validate all the properties of one app together in RQ2 because the historical bugs occur on different app versions. We allocated the same testing time (24×116 machine hours) per strategy to ensure fairness. We inspected all the bugs found by Kea and reported unique ones to app developers for confirmation.

Evaluation method of RQ4. RQ4 aims to compare Kea with prior functional testing techniques. To this end, we compared with Genie [36] and Odin [43] which propose automated oracles to find functional bugs, and PBFDroid [37] which is a property-based testing technique for finding data manipulation errors. We evaluated these three prior tools on the 97 historical bugs from RQ2 and the new bugs from RQ3. Specifically, we conducted a two-step analysis to evaluate these tools' ability in finding bugs. First, we carefully read the corresponding papers and ran these tools to understand their techniques. To avoid misunderstanding, we contacted and discussed with the authors of Genie, Odin, and PBFDroid, to validate our understanding of their techniques. Then, we manually analyzed which bugs in RQ2 and RQ3 are within these tools' capability scopes. Second, we ran each tool on the bugs within their scopes to validate whether they could find the bugs in practice. We followed the default setup of Genie and Odin in their papers. For PBFDroid, it requires users to specify the properties of data manipulation functionalities (DMFs). Thus, we manually specified the properties of needed DMFs for finding the corresponding bug. We also conducted a sanity check to ensure the specified properties can indeed capture the corresponding bug like what we did in RQ1 Then, we allocated the same time in RQ2 (6 hours) and RQ3 (24 hours) for Genie, Odin, and PBFDroid to find each bug.

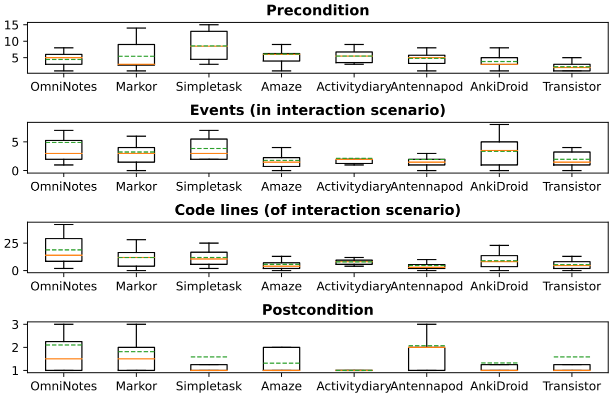


Figure 6: Complexity of the properties in different apps

All the experiments in RQ2, RQ3, and RQ4 were conducted on a 64-bit Ubuntu 22.04 machine (128 cores, AMD EPYC 7742 CPU, 256G RAM) and Android emulators (Android 10, Pixel). Note that all the 124 historical bugs are reproducible on Android 10.

5.2 Results of RQ1

Generability of Kea. We find that all the properties violated by the 124 historical bugs can be specified in our PDL in Kea, demonstrating the generability. Moreover, all the historical bugs can be revealed when following the original bug-triggering event traces from the app entry to the app state satisfying the property's precondition and executing the interaction scenario. Thus, all the properties are correctly specified.

Complexities of the properties. Figure 6 presents the complexity of the 124 properties across eight apps. The horizontal axis gives the app names, and the vertical axis gives the value of the corresponding complexity. Specifically, the average values of complexities for the precondition and postcondition are 4.9 and 1.6, respectively. The average values of the number of events and code lines for interaction scenario are 3.1, and 9.9, respectively. The precondition is more complicated than the postcondition because more clauses are typically needed in the precondition to ensure precise checking of the property. In contrast, the postcondition usually needs only

Table 2: #Historical bugs found by different strategies.

Random_100	Random_80	Random_60	Random_40	Random_20	Guided exploration
66 (68.0%)	63 (64.9%)	59 (60.8%)	63 (64.9%)	54 (55.7%)	92 (94.8%)

Table 3: Bug finding time of the 65 historical bugs found by both Random_100 and the guided exploration strategies.

Bug Finding Time (s)	Average	Min	Q1	Median	Q3	Max
Random_100	3,171	19	210	931	4,943	20,939
Guided exploration	684	22	42	88	187	7,956

Table 4: Number of new bugs found by the four modes.

Mode	#Found Bugs
Random_Single_Property	17
Guided_Single_Property	23
Random_All_Properties	22
Guided_All_Properties	22

one or two clauses to check the property. The results show that the properties can be specified with an acceptable complexity.

5.3 Results of RQ2

Table 2 shows the average numbers of historical bugs found by the random and guided exploration strategies. Specifically, Random_100, Random_80, Random_60, Random_40 and Random_20 denote the five different configurations of maximum numbers of allowed events (*i.e.*, 100, 80, 60, 40, 20 events) in one GUI test for the random exploration strategy, respectively. We can see that Random_100, Random_80, Random_60, Random_40 and Random_20 found 66, 63, 59, 63, and 54 bugs, respectively, while the guided exploration strategy found 92 bugs. These results show that (1) Random_100 found the most number of bugs among the five configurations of the random exploration strategy; (2) the guided exploration strategy is more effective in finding historical bugs than the random one. Specifically, all but one bugs found by Random_100 were found by the guided exploration strategy.

Among the 97 historical bugs, four bugs cannot be found by any strategy. We find that triggering these bugs requires long and specific event traces which are difficult to generate by the random or the guided strategy alone. For example, in *OmniNotes*'s issue #812, a note will lose the attached photos after it is backed up and restored. The property is specified as follows: the precondition checks the existence of a note, the interaction scenario is backing up and restoring the note, and the postcondition checks whether the note is correctly restored. However, to reveal this bug, we need to automatically generate a long event trace which opens the camera, takes a photo, creates a note, and attaches the photo to this note.

Table 3 shows the time cost (in seconds) of 65 bugs that were found by both the Random_100 and the guided exploration strategies. The guided strategy is nearly 4X faster than Random_100 (684 vs. 3,171 seconds) in terms of the average time of finding bugs. Among these 65 bugs, 17 bugs cannot be triggered by executing the main path but require guided exploration along the main path. For these 17 bugs, the guided strategy only took 40% time cost of Random_100 in terms of the average time of finding bugs (2,358 vs. 5,858 seconds). Specifically, on 14 out of these 17 bugs, the guided strategy was faster than Random_100. Among the 27 bugs only found by the guided strategy, 13 bugs cannot be found by executing the main path but require guided exploration along the main path.

5.4 Results of RQ3

We found 25 new functional bugs in total on the latest app versions. All these bugs have been confirmed, and 21 have already been fixed. Table 5 lists these 25 functional bugs (the affected app name, bug ID, bug status, whether the bug occurs on the main path or alternative path, and bug description). Here, "*Main path*" denotes that the bug is found by executing the main path during property checking (*i.e.*, the bug is a regression), while "*Alternative path*" denotes that the bug requires additional exploration and occurs on an alternative path rather than the main path during property checking.

We applied four modes to validate properties under the random and the guided strategies: (1) validating one property at one run (denoted by Random_Single_Property and Guided_Single_Property, respectively), and (2) validating all properties of an app together (denoted by Random_All_Properties and Guided_All_Properties, respectively). Note that the random strategy is configured as Random_100 which performs best in RQ2. Table 4 shows that these four modes found 17, 23, 22, and 22 bugs, respectively.

We examined the bugs found by different modes and obtained some interesting observations. *First*, all the bugs found by Random_Single_Property were found by the other three modes. *Second*, Guided_Single_Property found more bugs than the other three modes. Indeed, it found two bugs that were not found by Random_All_Properties and Guided_All_Properties. The main reason is that the main path provides guidance for reaching a property, and the guided exploration can generate many alternative paths for extensive testing. For example, manifesting the bug (with bug ID 9) requires performing guided exploration along the main path to generate some new events (archiving a note, clicking the search button, clicking back). This bug was found by Guided_Single_Property alone. Although Guided_All_Properties also has exploration guidance, it missed this bug. Because validating multiple properties together may decrease the chance of validating one single property, thus missing some bugs.

Third, validating multiple properties together has its own benefits. For example, Random_All_Properties found 5 more bugs than Random_Single_Property; both Random_All_Properties and Guided_All_Properties found two bugs which were not found by Guided_Single_Property. The main reason is that the interaction scenarios of multiple properties provide a partial model of the app. This partial model can help reach deeper app states, thus increasing the chance of finding bugs. Validating multiple properties together also improves the testing efficiency. For example, Random_All_Properties and Guided_All_Properties found 6 bugs in *Markor* within 24 machine hours, while Random_Single_Property and Guided_Single_Property only found 5 bugs by taking 12*24 machine hours (24 hours for each of the 12 properties in *Markor*), respectively.

5.5 Results of RQ4

Table 6 investigates how many of the 97 historical bugs and the 25 new bugs found by Kea can be found by Genie, Odin and PBFDroid. "#Historical Bugs in Scope" and "#New Bugs in Scope" give the numbers of bugs within the capability scopes of these tools. "#Found Historical Bugs" and "#Found New Bugs" give the numbers of bugs found by these tools in practice. Among the 97 historical

Table 5: Statistics of the 25 new functional bugs found by Kea.

App Name	ID	Bug Status	Bug Occurs On	Bug Description
OmniNotes	1	Fixed	Alternative path	The note tag cannot be removed.
	2	Fixed	Alternative path	The uncategorized item in the navigation still appears when it does not contain notes.
	3	Fixed	Alternative path	Deleting one tag in the note changes another tag.
	4	Fixed	Alternative path	The note content is changed when clicking to share.
	5	Confirmed	Alternative path	There exists duplicated note categories.
	6	Fixed	Alternative path	Wrong search result when searching for the note with tags.
	7	Fixed	Alternative path	The uncategorized item in the navigation does not appear when it should.
	8	Fixed	Alternative path	The locked note's content can be searched.
	9	Fixed	Alternative path	The archived note erroneously appears in the note list when clicking the search bar.
	10	Fixed	Alternative path	The search filter does not apply to the note list.
AmazeFileManager	11	Fixed	Main path	Recent files directory fails to display recent files.
	12	Fixed	Main path	The search result's title disappears after screen rotation.
	13	Fixed	Alternative path	Floating action button does not display when it should.
Markor	14	Fixed	Main path	Recently viewed documents does not update after users view some documents.
	15	Fixed	Main path	The file modification time does not change after modifying the file.
	16	Fixed	Main path	Clicking file format incorrectly jumps to the "Seach documents" dialog.
	17	Fixed	Alternative path	File content is overridden when creating a new file with the same name.
	18	Fixed	Alternative path	When creating a new file, the file type is not consistent with the suffix.
	19	Fixed	Alternative path	Delayed appearance of newly created folder.
SimpleTask	20	Confirmed	Alternative path	When creating a new task, an existing task is opened rather than a new task.
Transistor	21	Fixed	Alternative path	Playback indicator is not consistent.
	22	Fixed	Alternative path	After deleting one station, the playback metadata text still exists in another station.
	23	Fixed	Alternative path	The newly added station erroneously displayed as being long-clicked.
AnkiDroid	24	Confirmed	Main path	Card Browser does not remember the scroll position after editing a card.
	25	Confirmed	Main path	After repositioning a card, "Undo Reschedule" is shown instead of "Undo Reposition".

Table 6: Results of prior functional testing tools for finding the historical and new functional bugs.

Tool	#Historical Bugs in Scope	#Found Historical Bugs	#New Bugs in Scope	#Found New Bugs
Genie	13	4	1	0
ODIN	4	0	0	0
PBFDroid	23	9	7	1
#Total	34	13	7	1

bugs, only 35% (≈34/97) bugs are within the capability scopes of these three tools, and only 13 out of 34 bugs were found. For the 25 new bugs, only 28% (7/25) bugs are within the capability scopes of these three tools, and only 1 bug was found. It indicates that prior tools are limited in finding functional bugs. We further analyzed why these tools are limited in finding these functional bugs and found two major reasons. First, these tools only target specific types of functional bugs. Genie and Odin can only find bugs that violate their automated oracles, and PBFDroid can only find data manipulation errors. Second, the generated inputs are low-quality, and are difficult to reach the target functionality. Genie, Odin, and PBFDroid do not provide guidance during exploration like Kea. They generate many redundant tests.

6 DISCUSSION

6.1 Generability and Practicability

The results of RQ1 show that Kea is general as our PDL can specify the properties violated by many different functional bugs. Specifically, our PDL can specify the properties of the functional bugs targeted by PBFDroid (which we compared in RQ4). PBFDroid tests the data manipulation functionalities which perform the CRUD operations, *i.e.*, *create*, *read*, *update* and *delete*. The properties of these functionalities can be easily specified by our PDL (like the example property in Figure 2(b)). Moreover, we can use our PDL to record the app data at finer granularity to check more sophisticated properties (*e.g.*, *how many* data entries have been added or deleted), which however are not supported by PBFDroid.

Moreover, the testing strategies of other prior functional testing tools like Thor [1], ChimpCheck [17] and SetDroid [38] can also be supported by extending Kea. For example, Thor [1] and ChimpCheck [17] randomly inject neutral events (*e.g.*, rotating the device screen from portrait to landscape and rotating back) into human tests to check whether the assertions in the tests still hold. Kea can easily support this testing strategy by adding a new exploration strategy in the input generator module (*i.e.*, adding the neutral events during the random exploration and/or injecting neutral events at any random position of the interaction scenario) for validating properties. SetDroid [38] randomly changes system settings and restores them to find system setting related functional bugs. To support this strategy, we can add the events of changing and restoring system settings in the exploration strategies.

Kea does not aim to replace prior fully automated tools like Genie [36] and Odin [43]. The automated oracles of Genie and Odin could still complement Kea in finding bugs. Like any property-based testing technique, Kea requires manually specifying app properties. Our experience shows that it took about 3~5 minutes to specify a property in our PDL.

6.2 Threats to Validity

Our work may suffer from some threats to validity. First, the apps used in our experiment may not represent all the real-world apps. But we believe our property-based testing technique is general for different types of apps as our property description language is general. Also, selecting the apps from prior relevant work allows a more fair and direct comparison with prior testing tools. In the future, we would apply Kea to more apps. Second, the app properties used in our experiment are manually specified based on 124 historical bugs. The quality and diversity of these properties may affect the bug finding results of RQ2 and RQ3. To this end, we tried our best to ensure these properties are unbiased, correct, and general. We selected the historical bugs without any cherry picking to reduce potential biases. Moreover, the specified properties are rigorously cross-checked between two co-authors and later double-checked by the other two co-authors.

7 RELATED WORK

Finding functional bugs in Android apps faces the oracle problem [3]. Most of prior work [1, 13, 30, 36, 38, 39, 43, 45, 46] designs *automated oracles* to overcome the preceding problem. In these work, Genie [36] and Odin [43] are the representative ones because they are not limited to specific types of functional bugs. However, as we have already discussed in Section 1, they are limited in generability (covering limited portions of functional bugs) and practicality (leading to high false positives). Other work in this direction is limited to specific types of functional bugs like data losses [1, 13, 30, 46] and system setting related defects [38, 39]. As we have discussed in Section 6.1, Kea could also support finding such types of functional bugs.

Property-based testing [9, 11] is a powerful testing approach and has been applied to find functional bugs in many different software systems [2, 14, 15, 27, 31, 40]. To our knowledge, ChimpCheck [17] and PBFDroid [37] are the only work applying property-based testing for Android apps. However, ChimpCheck's main contribution is designing novel UI trace generators, which can fuse example-based tests with random testing. It directly reuses the assertions in the example-based tests as the oracles. However, such assertions are much less general and expressive than our property description language, and applying ChimpCheck still requires manually creating complete GUI tests. As a side note, ChimpCheck has not been demonstrated for finding functional bugs in its paper. On the other hand, PBFDroid is limited to specifying the properties for data manipulation functionalities. The experimental comparison in Section 5.5 and the detailed discussion in Section 6.1 have already shown that Kea is more general and practical than PBFDroid. Moreover, PBFDroid only uses random testing for generating inputs, while Kea designs a guided exploration strategy which is more effective in finding bugs. Some prior work uses code coverage [18, 28] or energy consumption [23] as the guidance for improving property-based testing.

8 CONCLUSION

This paper introduces a general and practical testing technique based on property-based testing for finding functional bugs in Android apps. We design a property description language and two exploration strategies. The evaluation results show that this technique is general and practical for functional testing. It found 92 (94.8%) of 124 historical functional bugs, and 25 new functional bugs on the latest version of the apps. All of 25 new bugs have been confirmed and 21 of them have been fixed by the developers.

ACKNOWLEDGMENTS

We thank the anonymous ASE reviewers for their valuable feedback. This work was supported in part by NSFC Project (No.62072178), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, China Scholarship Council, and Grant ZYGX2024K008 of National Key Laboratory on Blind Signal Processing. Ting Su, Jue Wang and Geguang Pu are the corresponding authors of this paper.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 83–93.

[2] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.

[3] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[4] Farnaz Behrang and Alessandro Orso. 2020. Seven reasons why: an in-depth study of the limitations of random test input generation for Android. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1066–1077.

[5] David Bolton. 2024. *88% Of People Will Abandon An App Because Of Bugs*. Retrieved 2024-5 from https://www.applause.com/blog/app-abandonment-bug-testing/

[6] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[7] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[8] 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 (ASE)*. IEEE, 429–440.

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[10] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.

[11] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.

[12] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[13] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 605–616. https://doi.org/10.1145/3533767.3534402

[14] John Hughes, Benjamin C Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of dropbox: property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 135–145.

[15] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 131–141.

[16] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[17] Edmund SL Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. ChimpCheck: property-based randomized test generation for interactive apps. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 58–77.

[18] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[19] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.

[20] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 613–622.

[21] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367.

[22] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Zhilin Tian, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[23] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 46–56.

[24] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[25] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[26] Motherboard. 2020. *Here's the Shadow Inc. App That Failed in Iowa Last Night*. Retrieved 2024-5 from https://www.vice.com/en_us/article/y3m33x/heres-the-shadow-inc-app-that-failed-in-iowa-last-night

[27] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1025–1038.

[28] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.

[29] Reuters. 2021. *Japan's COVID-19 app failed to pass on some contact warnings*. Retrieved 2024-5 from https://www.reuters.com/article/us-health-coronavirus-japan-app-idUSKBN2A31BA

[30] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152. https://doi.org/10.1145/3395363.3397379

[31] André Santos, Alcino Cunha, and Nuno Macedo. 2018. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 56–62.

[32] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[33] Sixth Tone. 2019. *E-Commerce App Loses 'Tens of Millions' From Coupon Glitches*. Retrieved 2024-5 from https://www.sixthtone.com/news/1003483

[34] 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 *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 245–256. https://doi.org/10.1145/3106237.3106298

[35] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 119–130.

[36] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–31. https://doi.org/10.1145/3485533

[37] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1088–1100. https://doi.org/10.1145/3611643.3616286

[38] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 204–215. https://doi.org/10.1145/3460319.3464806

[39] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and Finding System Setting-Related Defects in Android Apps. *IEEE Trans. Software Eng.* 49, 4 (2023), 2941–2963. https://doi.org/10.1109/TSE.2023.3236449

[40] Jingling Sun, Ting Su, Jun Sun, Jianwen Li, Mengfei Wang, and Geguang Pu. 2024. Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 440–451.

[41] Ash Turner. 2024. *How Many Android Users Are There? Global and US Statistics (2024) (Source: https://www.bankmycell.com/blog/how-many-android-users-are-there)*. Retrieved 2024-5 from https://www.bankmycell.com/blog/how-many-android-users-are-there

[42] uiautomator2 Team. 2021. *uiautomator2*. Retrieved 2024-5 from https://github.com/openatx/uiautomator2

[43] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 434–446. https://doi.org/10.1145/3540250.3549170

[44] 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.

[45] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23)*. 1319–1331.

[46] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 183–192. https://doi.org/10.1109/ICST.2014.31