KEA2: Practical Property-based Testing for Mobile Apps

Xixian Liang[1], Cheng Peng[1], Bo Ma[1], Xiangchen Shen[1], Yiheng Xiong[2], and Ting Su[1*]
[1]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[2]Singapore Management University, Singapore
{xixian,pengcheng,boma,XiangchenShen}@stu.ecnu.edu.cn, yihengx98@gmail.com, tsu@sei.ecnu.edu.cn

Abstract

Validating functional correctness of mobile apps is challenging for manual testing and automated UI testing (limited to *simple* crashing bugs). Property-based testing is promising to tackle this challenge given functional properties. To this end, we introduce KEA2, a practical property-based testing tool for apps: (1) specifying properties in Python with enough flexibility and expressiveness; and (2) reusing existing GUI fuzzing techniques to support effective property checking. Indeed, KEA2 can find functional (logic) bugs in real-world apps. KEA2 has been open-sourced at https://github.com/ecnusse/Kea2 (a demo video: https://youtu.be/HS4rTCcaSPE), and received positive feedback for its usability and features.

ACM Reference Format:
Xixian Liang, Cheng Peng, Bo Ma, Xiangchen Shen, Yiheng Xiong, and Ting Su. 2026. KEA2: Practical Property-based Testing for Mobile Apps. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3806416

1 Introduction

Property-based testing (PBT) [2] is an effective testing approach. It validates a piece of software against user-defined properties by automatically generating a large number of diverse inputs. PBT can effectively uncover latent bugs which are difficult to be found by classic example-based testing [1, 8–10, 15, 19].

Recently, the idea of PBT has been adapted to validate the functional correctness of mobile apps [11, 21, 22, 32]. In these work, users specify the intended app behaviors as properties, and then PBT generates a large number of GUI-based tests to validate these properties. Indeed, these work has found many interesting, previously-unknown functional (logic) bugs.

However, we find that the PBT tools proposed in these work are difficult to adopt in practice. This is mainly because (1) they only target limited app functionalities, and (2) they require developers to specify and maintain properties in ad-hoc languages, which limits flexibility and expressiveness. For example, PBFDroid [21] and PDTDroid [22] specify properties in JSON, making it difficult to express complex GUI interactions and property-checking logic; ChimpCheck [11] uses a self-defined DSL which however introduces additional learning overhead for users. To our knowledge, none of these tools has seen real-world industrial adoption.

To bridge this gap, we present KEA2, a practical property-based testing tool for mobile apps. KEA2 is the successor of our prior work KEA [32], but it is redesigned and implemented from scratch to ensure practical usability and scalability. It supports more flexible and expressive property specification, and leverage three widely used open-sourced project for praticality in an industrial setting: `unittest` [5] for property management, FastBot [12] for input generation, and `uiautomator2` [29] for device interaction. In addition, KEA2 has been continuously refined based on feedback from frontline testers, with substantial improvements to its command-line interface, test reporting, performance, and compatibility, making it suitable for industrial use. Since the first release in May 2025, KEA2 has received 200+ GitHub stars and 28k+ downloads from PyPI. To date, it has already been adopted by some industrial app vendors (*e.g.*, Tencent, Haier, and Huawei) and received positive feedback for its usability and versatile features.

2 High-level Idea

Property-based Testing. Property-based testing *automatically* generates a *large* number of inputs to validate whether a program satisfies some properties [2]. For example, consider testing a list reversing function `reverse`. Instead of manually enumerating concrete input-output pairs like example-based testing [3] (*e.g.*, `reverse([1,2])==[2,1]`), one can specify a property — reversing a list x twice should return x itself, *i.e.*, `reverse(reverse(`x`))` $== x$, for validation. Specifically, PBT would generate many lists with different sizes and integers to check whether this property holds. If the property is violated, a counterexample is reported.

Property-based Testing for Mobile Apps. Applying property-based testing to mobile apps is not straightforward. We need

*Ting Su is the corresponding author.

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/26/07
https://doi.org/10.1145/3803437.3806416

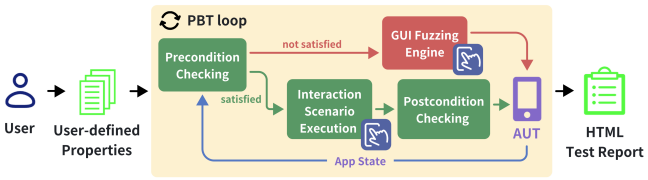

Figure 1. Workflow of KEA2

to explicitly model app states, interaction scenarios, and expected behaviors. We formalize the key concepts as follows.

App State and Property. An app state s can be denoted as $s = \langle l, \sigma \rangle$, where l denotes the visual GUI layout, and σ denotes the internal data of the app. We define an app property ϕ (in the form of Hoare logic [31]) as a triple $\phi = \langle P, I, Q \rangle$, where (1) P is precondition that defines when or where the property is checkable, (2) I is a piece of program that simulates the interaction scenario with the app, and (3) Q is postcondition that defines the expected results after I.

Input Generator and Property Checking. PBT could repeatedly drive the app into different states by adopting some GUI-based input generators [12, 20]. At each reached state, app properties are checked. Given a state s and a property $\phi = \langle P, I, Q \rangle$, ϕ is considered satisfied if $(s \models P \wedge s \overset{I}{\rightsquigarrow} s') \Rightarrow s' \models Q$. That is, whenever the precondition P holds on the state s, executing the interaction scenario I from s must transition the system to a subsequent state s' that satisfies the postcondition Q. If $s \models P$ but executing I leads to a state s'' such that $s'' \not\models Q$, then a property violation is found.

3 KEA2

3.1 Overview

We realized the preceding idea of property-based testing as a tool named KEA2 targeting Android apps. KEA2 is built on top of (1) Python's `unittest` framework for property management and execution, (2) `uiautomator2` as the driver for interacting with Android devices, and (3) `Fastbot` as the input generator. Figure 1 shows KEA2's workflow. Users first specify app properties. During testing, KEA2 repeatedly performs *Precondition Checking* for all properties on the current app state. If one or more preconditions are satisfied, KEA2 selects a property and executes its *Interaction scenario*, followed by *Postcondition Checking* on the app under test (AUT). If no precondition is satisfied, KEA2 invokes the *GUI Fuzzing Engine* as the input generator to generate GUI events and drive the AUT to new states. Finally, KEA2 generates an *HTML Test Report* to summarize the testing results.

3.2 Property Definition

In KEA2, app properties are defined as Python test methods in a `unittest` test class. Each property is a method decorated with `@precondition`:

- The `@precondition` decorator is used to specify the precondition P of a property. It takes a Python function that returns `True` or `False` to specify P. In the decorated method

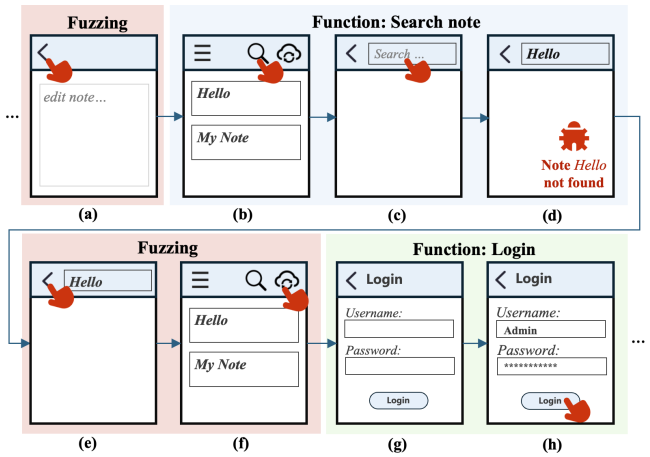

Figure 2. An example of KEA2's execution flow

Listing 1. An illustrative example of app properties

```
from kea2 import state, precondition, prob, max_tries

state["notes"] = list()
class PropertyBasedTestCase(unittest.TestCase):
    @precondition(
        lambda self: self.d(resourceId="create").exists
    )
    def create_note(self):
        self.d(resourceId="create").click()
        content = generate_random_text()
        self.d(resourceId="content").set_text(content)
        state["notes"].append(content)

    @prob(0.5)
    @precondition(
      lambda self: self.d(resourceId="search").exists
      and len(state["notes"]) > 0
    )
    def search_note(self):
      self.d(resourceId="search").click()
      note = random.choice(state["notes"])
      # search the note
      self.d(resourceId="search_box").input(note)
      self.d.press("ENTER")
      # assert the searched note is displayed
      assert self.d(text=note).exists

    @max_tries(1)
    @precondition(
      lambda: self.d(description="login").exists
    )
    def login():
      self.d(resourceId="username").input(USERNAME)
      self.d(resourceId="password").input(PASSWORD)
      self.d(description="login").click()
```

body, it defines the interaction scenario I and postcondition Q (by using `asserts`). Within this method, users can write arbitrary Python code and use `uiautomator2` APIs to interact with the apps. This design makes property definitions flexible and expressive.

KEA2 offers another two decorators `@prob` and `@max_tries` to customize property definition:

- The `@prob` decorator controls how often a property is executed and checked when its precondition is satisfied. This

Algorithm 1 Property-Based Testing in KEA2

```
1:  function MAIN(Φ, T, σ_0)        ▷ Φ = {φ_1, φ_2, ..., φ_n}, φ_i = ⟨P_i, I_i, Q_i⟩
2:      σ ← σ_0, t ← 0                          ▷ Initialization: state σ, time t
3:      while t < T do                          ▷ Loop until time budget exhausted
4:          l ← DUMPLAYOUT()
5:          s ← ⟨l, σ⟩                                  ▷ Construct app state
6:          Φ_checkable ← GETCHECKABLEPROPERTIES(Φ, s)
7:          if Φ_checkable ≠ ∅ then
8:              φ_j ← RANDOMSELECT(Φ_checkable)
9:              PROPERTYCHECKINGENGINE(φ_j, s)
10:         else
11:             GUIFUZZINGENGINE(s)
12:         end if
13:         t ← t + Δt
14:     end while
15:     GENERATETESTREPORT()
16: end function
17: function GETCHECKABLEPROPERTIES(Φ, s)
18:     Φ_checkable ← ∅
19:     u ← RANDOM(0, 1)
20:     for each φ_i ∈ Φ do
21:         if s ⊨ P_i and u ≤ φ_i.p and φ_i.c < φ_i.c_max then
22:             Φ_checkable ← Φ_checkable ∪ {φ_i}
23:         end if
24:     end for
25:     return Φ_checkable
26: end function
27: function PROPERTYCHECKINGENGINE(φ, s)
28:     φ.c = φ.c + 1          ▷ Increment execution count of the property
29:     σ' ← I(σ)              ▷ Execute interaction scenario function I
30:     l' ← DUMPLAYOUT()
31:     s' ← ⟨l', σ'⟩
32:     UPDATERESULT(s' ⊨ Q)                        ▷ Check postcondition Q
33: end function
```

probabilistic design allows users to trade-off between exploration (invoking the GUI fuzzing engine) and exploitation (invoking property checking).

- The `@max_tries` decorator specifies an upper bound on how many times a property can be executed and checked during a testing session. It is useful for modeling some one-shot interactions (*e.g.*, initial setup, account login).

Example. Figure 2 shows an example execution flow of a note-taking app. Figure 2(b–d) illustrates note searching: clicking the search button (b), entering a previously created note name (c), and pressing Enter to perform the search (d). One interesting property is that *a created note should be found via note searching*. Listing 1 shows its specification in KEA2. The properties are organized in a `unittest` test class (Line 4). The `search_note` property is defined as a Python method (Line 19): (1) `@precondition` specifies the search button should exist and some created notes should exist (Lines 15–18); (2) the interaction scenario randomly selects an existing note and searches for this note (Lines 20–24); and (3) `assert` checks whether the expected note appears on the screen (Line 26).

3.3 Core algorithm

Algorithm 1 formalizes the core algorithm of KEA2. The algorithm starts from the MAIN function (Lines 1–16), taking the user-defined properties $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ as input. It repeats the testing loop until the time budget T is exhausted. For each iteration, KEA2 first gets the current state s, and computes the set of checkable properties $\Phi_{\text{checkable}}$ via GETCHECKABLEPROPERTIES (Lines 17–26). A property $\phi_i = \langle P_i, I_i, Q_i \rangle$ is considered checkable only if: (1) the current state s satisfies its precondition P_i (*i.e.*, $s \models P_i$); (2) its execution probability $\phi_i.p$ (defined via `@prob`) exceeds a random threshold $u \sim U(0, 1)$ (*i.e.*, $\phi_i.p \geq u$); and (3) its execution count $\phi_i.c$ has not reached the maximum limit $\phi_i.c_{\max}$ (defined via `@max_tries`) (Line 21). Based on the result, the algorithm proceeds as follows. If the set of checkable properties is non-empty ($\Phi_{\text{checkable}} \neq \emptyset$), KEA2 randomly selects a property ϕ_j from the set and delegates control to the PROPERTYCHECKINGENGINE. Otherwise ($\Phi_{\text{checkable}} = \emptyset$), it invokes the GUIFUZZINGENGINE for further exploration.

The PROPERTYCHECKINGENGINE function (Lines 27–33) handles property execution. Given a property $\phi = \langle P, I, Q \rangle$ and state s, the engine invokes the interaction scenario function I, which performs a sequence of UI events to drive the app from the current state s to a new state s'. Finally, the engine checks whether the resulting state s' satisfies the postcondition Q (*i.e.*, $s' \models Q$) and records the result.

Example. Figure 2 illustrates how KEA2 operates with the `search_note` property in Listing 1. KEA2 first randomly explore the app by invoking the input generator, which navigates to the home page (Figure 2(a)).

On the home page, KEA2 captures the current app state and checks whether the property's precondition P holds. Since P is satisfied, KEA2 executes the `search_note` property by searching for an existing note (Figure 2(b–d)). It then checks the postcondition Q and finds that the expected note does not exist (Figure 2(e)). Thus, a functional bug is found.

3.4 Implementation

KEA2 comprises over 14,000 lines of code in Python, Java, and HTML/JavaScript. KEA2 is designed as a client/server architecture. It uses Fastbot as the input generator which is deployed on mobile devices (*i.e.*, the clients) for fast GUI fuzzing. Fastbot uses model-based and reinforcement learning based testing algorithms. KEA2 runs on a local machine (*i.e.*, the server) and coordinates between GUI fuzzing (performed by Fastbot) and property checking.

In addition, we introduced two major optimizations: (1) *widget occlusion detection* to exclude occluded widgets from dumped UI layouts, thereby eliminating false positives during property checking; (2) *precondition validation caching* to enable checking the preconditions of all properties in one single UI dump, thereby improving the scalability of property checking.

Kea2 is released on PyPI and could be deployed by one command line. A typical usage is:

```
kea2 run -p package_names [...] --running-minutes time
         -s device_serial_number [--take-screenshots]
         propertytest discover -p pattern
```

Users can specify the target app(s), time budget, device serial, and optionally record screenshots; Kea2 uses `unittest`'s test discovery, which allows specifying the rules of matching property files (*e.g.*, `-p property*.py` matches any property file whose name starts with the string `property`).

4 Kea2's Features

Given user-specified properties, Kea2 can help find functional bugs beyond crashes. In addition, Kea2 offers additional features.

Stateful Testing. Property checking may depend on historical app states. To support this, Kea2 enables *stateful testing*, allowing users to explicitly maintain and query states during testing. For example, in Listing 1, the data structure `state` (Line 3) stores historical states. Specifically, the `create_note` property (Lines 5–12) can record a newly created note into `state` (Line 12); the `search_note` property can query them (Line 21) to check whether expected notes exist.

Guided Exploration. When a property's Q is set as always `True`, Kea2 enables guided exploration of specific interaction scenarios (*e.g.*, testing specific app functionalities, entering specific UI pages, reaching specific app states), that are hard for input generators to cover. For example, the `login` property (Lines 29–35) in Listing 1 enables user login.

Fusing scripted testing and GUI fuzzing. Kea2 is designed to coordinate between GUI fuzzing and property checking. Thus, Kea2 can fuse scripted testing and GUI fuzzing. For example, Kea2 implemented such a hybrid testing mode: Kea2 first runs UI test scripts to reach some interesting app state and then launches GUI fuzzing from that state to perform exploratory testing.

Comprehensive Test Report. Kea2 generates comprehensive HTML test reports, including statistics on property coverage, violations, and bug-triggering tests. It also supports merging reports to consolidate results from multiple testing runs. A sample report is available online [25].

5 Evaluation

We evaluate Kea2 on representative open-source and industrial apps to demonstrate its bug-finding ability (Table 1). We selected three highly-starred open-source apps from prior work [21, 32]: *Amaze* [23], *Markor* [26], and *AnkiDroid* [24]. We wrote 40 properties for these apps. Among them, 35 validate core functionalities (*i.e.*, these properties have `assertions`), 15 involve stateful testing (*e.g.*, testing CURD operations [21]), and 5 use guided exploration. Each app was tested by Kea2 for 3 hours. Kea2 successfully found 8 previously unknown functional bugs, all confirmed by developers, with 3 already fixed. One interesting bug (Issue #4558) found in *Amaze* reveals that the file list cannot be automatically refreshed under the "Audio" tab after deleting a file, although the developer has already fixed a similar bug under the main page in PR#4493. Kea2 found a different input (*i.e.*, deleting a file under "Audio" tab) that violates the property.

Table 1. New Functional Bugs Found by Kea2

App	Stars	Bugs found	Properties
Markor	5.1K	#2720	8(8*)
AnkiDroid	11K	#20094, #20095, #20102	7(4*, 2†)
Amaze	6K	#4558, #4559, #4560, #4561	25(3*,†)
WeChat	–	41 bugs	133

$\star$ denotes using stateful testing; † denotes using guided exploration.

We deployed Kea2 at Tencent to test *WeChat* [28], a popular messaging app with billions of active users worldwide. We replaced its default GUI fuzzing engine with `iExplorer`, WeChat's in-house engine based on multi-armed bandits [18] and random exploration. We derived 133 properties from the regression tests. We tested three released versions (v8.0.55-v8.0.57) and found 41 previously unknown functional bugs, which include ghost contacts, missing widgets, and hangs.

Community feedback. To date, Kea2 has been adopted by several industrial app vendors. For example, *OPay* [27]uses it as the default regression testing tool on POS machines. Its test manager commented, "Kea2 is a powerful tool for testing engineers, enabling them to easily customize exploration capabilities based on business requirements." A tester from Haier commented, "Kea2 is even more effective [than Fastbot] when combined with unittest (PBT). This project is very helpful for me and my team in testing our app." Beyond functional testing, Kea2 is also integrating into Huawei's HarmonyOS IDE, DevEco Studio [7], to automatically discover performance bugs.

6 Related work

Property-based testing (PBT) is effective in validating program correctness. The pioneering PBT tool QuickCheck [2] popularized this idea and has inspired a large body of subsequent work, including extensions to different languages (*e.g.*, Hypothesis [13] for Python, and JQF [16] for Java) and domains (*e.g.*, Robot operating system, RESTful APIs) [1, 9, 10, 15, 19, 32]. Recent work has shown the potential of PBT in testing mobile apps [11, 21, 22, 32]. These work applies PBT to specific app features, *e.g.*, data manipulation functionalities [21] and privacy-related functionalities [22]. However, the relevant tools are difficult to use in practice. To our knowledge, Kea2 is the only practical property-based testing tool for mobile apps. Moreover, Kea2 can help find functional (logic) bugs, while most of prior work on mobile app testing can only find crashing bugs [4, 6, 14, 17, 20, 30, 33].

Acknowledgments

We thank the anonymous FSE reviewers for their valuable feedback. This work was partially supported by NSFC Project No. 62572192 and 62072178.

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. 2–10.

[2] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP'00*. 268–279.

[3] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211.

[4] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 481–492.

[5] Python Software Foundation. 2026. *unittest — Unit testing framework*. Retrieved 2026-1 from https://docs.python.org/3/library/unittest.html

[6] 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 (ICSE)*. IEEE, 269–280.

[7] Huawei. 2026. *deveco-studio*. Retrieved 2026-1 from https://developer.Huawei.com/consumer/en/deveco-studio/

[8] John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 169–186.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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 (PLDI)*. 1025–1038.

[16] 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.

[17] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164.

[18] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. 2023. Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 894–905.

[19] 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.

[20] 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.

[21] 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 *ESEC/FSE'23*. 1088–1100. doi:10.1145/3611643.3616286

[22] 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.

[23] Amaze Team. 2026. *AmazeFileManager*. Retrieved 2026-1 from https://github.com/TeamAmaze/AmazeFileManager

[24] Ankidroid Team. 2026. *Ankidroid*. Retrieved 2026-1 from https://github.com/ankidroid/Anki-Android

[25] Kea2 Team. 2026. *Example Bug Report*. Retrieved 2026-1 from https://ecnusse.github.io/kea2_sample_test_report/

[26] Markor Team. 2026. *Markor*. Retrieved 2026-1 from https://github.com/gsantner/markor

[27] Opay Team. 2026. *Opay*. Retrieved 2026-1 from https://www.opayweb.com/

[28] Tecent. 2026. *WeChat*. Retrieved 2026-1 from https://www.wechat.com/en/

[29] uiautomator2 Team. 2021. *uiautomator2*. Retrieved 2026-1 from https://github.com/openatx/uiautomator2

[30] 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 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 469–480.

[31] Wikipedia. 2026. *Hoare logic*. Retrieved 2026-1 from https://en.wikipedia.org/wiki/Hoare_logic

[32] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 53–64.

[33] 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 *ISSTA'23*. 1319–1331.