Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning

Zhengwei Lv
lvzhengwei.m@bytedance.com
Bytedance
Beijing, China

Chao Peng*
pengchao.x@bytedance.com
Bytedance
Beijing, China

Zhao Zhang
zhangzhao.a@bytedance.com
Bytedance
Beijing, China

Ting Su*
tsu@sei.ecnu.edu.cn
East China Normal University
Shanghai, China

Kai Liu†
liukai.0914@bytedance.com
East China Normal University /
ByteDance, Beijing, China

Ping Yang
yangping.cser@bytedance.com
Bytedance
Beijing, China

ABSTRACT

We introduce a reusable automated model-based GUI testing technique for Android apps to accelerate the testing cycle. Our *key* insight is that the knowledge of *event-activity transitions* from the previous testing runs, *i.e.*, executing which events can reach which activities, is valuable for guiding the follow-up testing runs to quickly cover major app functionalities. To this end, we propose (1) a probabilistic model to memorize and leverage this knowledge during testing, and (2) design a model-based guided testing strategy (enhanced by a reinforcement learning algorithm). We implemented our technique as an automated testing tool named Fastbot2. The evaluation on two popular industrial apps (with billions of user installations), Douyin and Toutiao, shows that Fastbot2 outperforms the state-of-the-art testing tools (Monkey, Ape and Stoat) in both activity coverage and fault detection in the context of continuous testing. To date, Fastbot2 has been deployed in the CI pipeline at ByteDance for nearly two years, and 50.8% of the developer-fixed crash bugs were reported by Fastbot2, which significantly improves app quality. Fastbot2 has been made publicly available to benefit the community at: *https://github.com/bytedance/Fastbot_Android*.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Reinforcement learning*.

---

*Chao Peng and Ting Su are the corresponding authors.
†Kai Liu was a research intern at ByteDance when this work was conducted.

---

1 INTRODUCTION

Mobile apps have drastically increased in number over the recent years [1]. Ensuring app quality is crucial to keeping user loyalty and maintaining business success. To this end, automated GUI testing has become an attractive and cost-effective solution [2, 10, 11].

In practice, an industrial app undergoes frequent updates to catch up with the changing user demands. At ByteDance, we release new updates of major apps on a weekly basis. As a result, continuous testing becomes crucial for quick feedback on app quality (*e.g.*, doing smoke testing) whenever a new internal version is built. However, simply adopting existing testing tools [10], although feasible, is *inefficient* and *ineffective*, as they simply rerun each version from scratch and do not leverage the knowledge from previous testing runs to accelerate GUI testing in the current run.

To fill this important gap, we introduce a reusable automated GUI testing technique. Our key idea is to leverage model-based testing (MBT). Among existing testing solutions, MBT is recognized for its unique model construction phase, which is ideal for storing and leveraging the prior knowledge. However, we face two major technical challenges in putting our idea into practice.

The first challenge is how to effectively store the knowledge from previous testing runs. Our *key* insight is that the knowledge of *event-activity transitions*, *i.e.*, executing which events reaches which activities, is valuable for guiding the follow-up testing to quickly cover core activities. Thus, we propose a *probabilistic model* as the basis of MBT to memorize such knowledge from each testing run. This model stores a set of event-activity transitions, each of which records the historical probability of an event to reach an activity. Moreover, to tackle the complexity of industrial apps, we introduce a conception of *hyper-event* to represent events in this model, which is useful to balance the model scalability and accuracy.

The second challenge is how to effectively leverage the prior knowledge to guide GUI testing. Classic MBT methods traverse the model to generate GUI events (*i.e.*, GUI tests). However, one prominent problem is that such GUI tests are likely to be broken due to the unawareness of the connectivity between different GUI events. To overcome this issue, our *key* insight is to employ the probabilistic model to achieve on-the-fly, guided model-based testing. Specifically, the probabilistic model (which stores the knowledge of event-activity transitions) provides one-step guidance about which events on the current GUI page could be selected to quickly reach
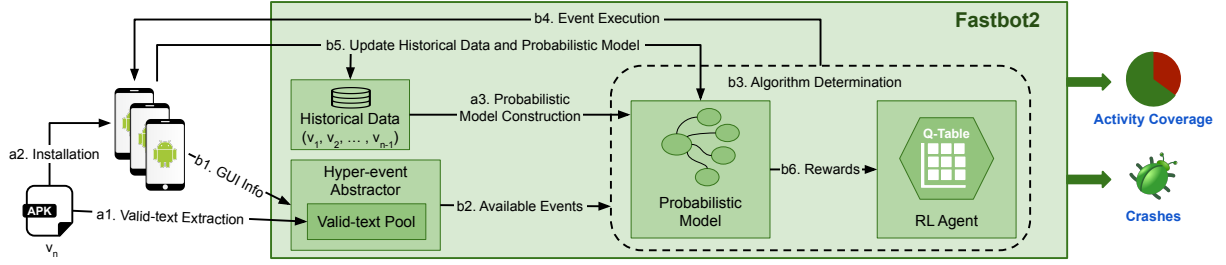
Figure 1: Fastbot2's workflow in each testing run. The arrows denote different steps, which are annotated by "a1", "b1", etc.

those not-yet-covered activities in the current testing run. Moreover, to further improve performance, we develop a reinforcement learning algorithm to provide multi-step guidance (also informed by the probabilistic model), which aims to reach those deep activities requiring executing multiple sequential events.

We implemented our technique as an automated testing tool named Fastbot2. Our evaluation on our two popular industrial apps, Douyin and Toutiao, shows that Fastbot2 outperforms the two state-of-the-art MBT tools, Stoat [9] and Ape [5], and the random testing tool Monkey [4] in both activity coverage and bug finding in the context of continuous testing. *To sum up, our work makes two major contributions*: (1) We propose a reusable automated model-based GUI testing technique enhanced by reinforcement learning to satisfy the practical needs of continuous testing, which has *not* been considered by prior work. (2) Our implementation, Fastbot2, outperforms the state-of-the-art. It has also been successfully deployed in the CI pipeline at ByteDance and received positive feedback on its ability of improving app quality.

2 FASTBOT2

Figure 1 shows Fastbot2's workflow. Fastbot2 takes as input an APK file of the app, outputs coverage and crash reports. Fastbot2 includes two major phases. The first phase does setup before testing: decompiling the APK to gather static widget text labels of (Step "a1"), installing the app on a pool of mobile devices (Step "a2"), and loading the historical data if available to populate the probabilistic model (Step "a3", *cf.* Section 2.2.1). The second phase does guided GUI exploration (*cf.* Section 2.3). Fastbot2 dumps the current GUI page from the app (Step "b1"), identifies and abstracts available hyper-events from the current page (Step "b2", *cf.* Section 2.2.2), selects the event which is likely to increase activity coverage (Step "b3"), executes this UI event (Step "b4") and updates the historical testing data, the probabilistic model (Step "b5") and the reinforcement learning agent (Step "b6"). These steps ("b1"~"b6") will be iteratively conducted until the time budget is used up.

2.1 An Illustrative Example

Figure 2(a) gives an illustrative example taken from Toutiao, a popular daily news app. We use this example to ease the exposition of our approach in Sections 2.2 and 2.3. In Toutiao, users can view news on Activity 1, click the news title (*e.g.*, "Title 1") to reach Activity 2 to view the content, click the news author ("Author") to view all news from this author on Activity 3 and watch the live video on Activity 4 if available by clicking the profile photo.
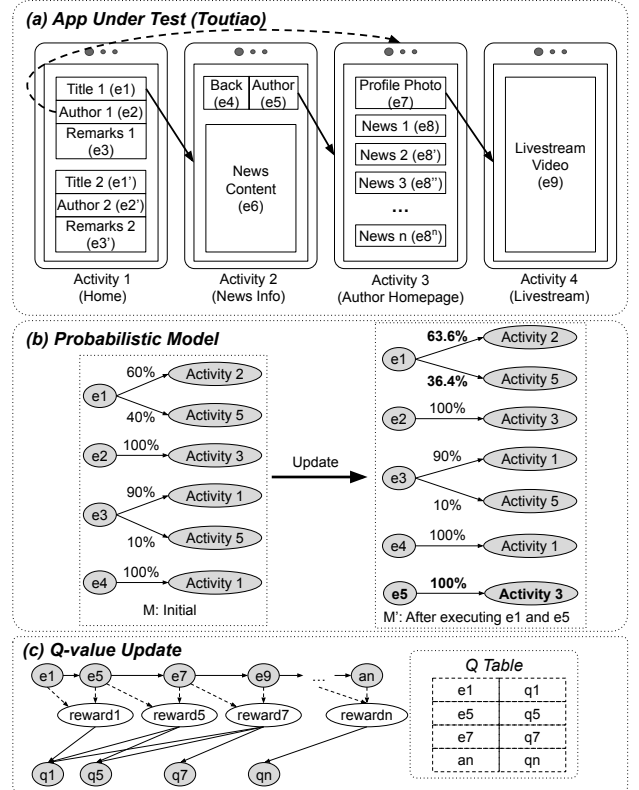

Figure 2: Activity Transition Example from Toutiao App

2.2 Probabilistic Model and Hyper-events

2.2.1 Probabilistic Model. We propose a probabilistic model as the basis of MBT to memorize the knowledge of *event-activity transitions* from previous testing runs. Specifically, this model M is formally defined as a 3-tuple $M = (\mathcal{E}, \mathcal{A}, \delta)$, where

- $\mathcal{E}$ is the set of hyper-events created from UI widgets.
- $\mathcal{A}$ is the set of activities of the app under test.
- δ is the transition function, *i.e.*, $\mathcal{E} \rightarrow \mathcal{P}(\mathcal{A} \times [0, 1])$. $\mathcal{P}$ is a powerset function and each transition is of the form $e \rightarrow (A, p)$, meaning that the probability of a hyper-event e reaching an app activity A is p, where $e \in \mathcal{E}$ and $A \in \mathcal{A}$.

The probabilistic model M is constructed from historical exploration data. The probability of reaching activity A_i by executing the hyper-event e (denoted by $P(e, A_i)$) is calculated by :

$$P(e, A_i) = \frac{N(e, A_i)}{N(e)} \tag{1}$$

where $N(e, A_i)$ denotes the number of times of e reaching A_i, and $N(e)$ denotes the total execution times of e_i in all the previous testing runs. If the hyper-event e can reach k activities (*e.g.*, $A_1, \ldots, A_i, \ldots, A_k$), $\sum_{i=1}^{k} P(e, A_i) = 1$ holds.

Example. Figure 2(b) gives an example of the initial probabilistic model (see the left part) loaded from previous testing runs before starting the current testing run. For example, Activity 2 can be reached by executing the hyper-event $e1$ on Activity 1, $e1$ can reach Activity 2 and 5 with probability values 60% and 40%, respectively.

2.2.2 Hyper-events. We propose the concept of *hyper-event* to represent the events in the probabilistic model. A hyper-event is created from each UI widget according to its properties. Specifically, we only consider the following *four* properties of a widget: the `activity` which the widget belongs to, the widget's `text`[1], `resource-id`, and the supported action types (*e.g.*, `click`, `long click`). In other words, if some widgets have the same four properties, we assume they have the similar functionality and only one hyper-event will be created. We ignore all the other minor widget properties (*e.g.*, a widget's type) when creating hyper-events with the aim of balancing between model scalability and accuracy.

Example. In Figure 2(a), on Activity 1, FASTBOT2 will create *only one* hyper-event "e1" for the widgets named "Title 1" and "Title 2" because these two widgets have the identical widget properties: the same activity, the same empty text ("Title 1" and "Title 2" are the texts dynamically loaded from the app server without static text labels), the same resource-id, and the same `click` action type. In this way, we will create three hyper-events, *i.e.*, "e1" (representing "Title 1" and "Title 2"), "e2" (representing "Author 1" and "Author 2"), "e3" (representing "Remarks 1" and "Remarks 2"), on Activity 1.

2.3 Model-based Guided UI Exploration

The key idea of FASTBOT2 is to reuse the prior knowledge stored in the probabilistic model to effectively guide GUI testing. To achieve this, the key step is to decide which UI event on the current GUI page should be selected so as to quickly increase activity coverage. This step corresponds to Step "b3" in Figure 1. Specifically, given a GUI page, FASTBOT2 extracts the available hyper-events, and selects the event[2] to be executed based on the two synergistically combined strategies: (1) *model-based event selection* (*cf.* Section 2.3.1), and (2) *learning-based event selection* (*cf.* Section 2.3.2).

2.3.1 Model-based event selection. Model-based event selection contains two modes, *i.e.*, *model expansion* and *model exploitation*.

Model expansion. If some hyper-events from the current GUI page have not been included in the probabilistic model M, FASTBOT2 will activate this mode to randomly select one not-yet-executed hyper-event. This situation may occur because the previous testing runs may not cover all hyper-events *or* some new app features have

been added in the current tested app version. This mode can help expand the model and prioritize exploring potentially new features.

Model exploitation. If all the hyper-events from the current GUI page have been included in the probabilistic model M, FASTBOT2 will activate this mode to select an event with higher probability to cover those not-yet-covered activities in the current testing run (which were covered in the previous testing runs). Let $\mathcal{A}_t$ be the set of already covered activities in the current testing run and $\mathcal{E}_c$ be the set of hyper-events from the current GUI page, the expectation of improving activity coverage by executing e_i ($e_i \in \mathcal{E}_c$) can be computed as $\mathbb{E}(e_i) = \sum_{A \notin \mathcal{A}_t} P(e_i, A), 0 \le i \le |\mathcal{E}_c|$.

Here, $\mathbb{E}(e_i)$ represents the expectation value of probability that those not-yet-covered activities in the current testing run will be covered after the hyper-event e_i is executed. The higher $\mathbb{E}(e_i)$, the more likely it is to improve activity coverage. Thus, FASTBOT2 in this mode selects the hyper-event e_i by probability $P_M(e_i)$:

$$P_M(e_i) = exp(\frac{\mathbb{E}(e_i)}{\alpha}) \quad / \sum_{e_i \in \mathcal{E}_c} exp(\frac{\mathbb{E}(e_i)}{\alpha}) \tag{2}$$

where, α is a hyperparameter which adjusts the randomness of this mode. This equation is adapted from the softmax formula. We also require that e_i should be selected no more than K times to ensure fairness. In practice, we set α as 0.8 and K as 1. By using the probabilistic model as priori information, the model exploitation mode can quickly improve activity coverage in a short time.

Example. In Figure 2(a), three hyper-events are available on Activity 1. Since all these three events have been included in the probabilistic model M (see the left part of Figure 2(b)), FASTBOT2 activates the model exploitation mode to select events. According to M, event e1 and e2 are more likely to reach unexplored activities (*i.e.*, Activity 2, 3, 5), while event e3 has 90% probability to stay in Activity 1. Thus, FASTBOT2 is likely to select e1 or e2. Assume e1 is selected and then Activity 2 is covered. In Activity 2, event e4 (the back button) has 100% probability to return back to Activity 1, while event e5 and e6 have not been included in M. As this time, FASTBOT2 activates the model expansion mode and randomly selects e5 or e6. Assume e5 is selected and then Activity 3 is covered. Meanwhile, M is updated by adding e5→Activity 3 with probability value 100% (see the right part of Figure 2(b)).

2.3.2 Learning-based event selection. However, the probabilistic model can only express one-step guidance information. Fortunately, reinforcement learning technique is able to spread one-step into multiple-step guidance information.

Q-table expansion. The key component of the RL agent is the Q-table, which contains the Q-values (which indicate the possibility of executing each hyper-event to reach a new activity). During testing, no matter which event selection strategy is used, the Q-value of the selected hyper-event e_t on the current GUI page, *i.e.*, $Q(e_t)$, is updated to $Q(e_t) + \alpha(G_{t,t+n} - Q(e_t))$, where $G_{t,t+n}$ is the n-step cumulative reward calculated by an N-step Sarsa method [3]:

$$G_{t,t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n Q(e_{t+n}) \tag{3}$$

Here, γ is the discount factor. r_{t+1} is the immediate reward earned after the event e_t is executed, which is defined as

---

[1]Here, the text means the static text labels stored in the resource files of the APK file. If the text is dynamically loaded from the app server, we treat its text as empty.
[2]After we decide which hyper-event should be selected, if the selected hyper-event represents multiple UI widgets, we will randomly pick one UI widget to exercise.

$$r_{t+1} = \frac{\mathbb{E}(e_t)}{\sqrt{N(e_t)+1}} + \frac{V}{\sqrt{N(A_t)+1}} \qquad (4)$$

Here, $N(e_t)$ denotes the number of times e_t is executed, A_t denotes the activity e_t leads to, and $N(A_t)$ denotes the number of times A_t is visited so far in the current testing. V represents the value of A_t and is calculated using:

$$V = n_h + 0.5 * n_c + \sum_{e_i \in \mathcal{E}_c} \mathbb{E}(e_i) \qquad (5)$$

Here, n_h is the number of hyper-events in the reached GUI page but are not in the probabilistic model. Thus, executing these hyper-events will likely touch new features. n_c is the number of hyper-events in the next GUI page and contained in the probabilistic model but have not been executed in the current testing run. $\sum_{e_i \in \mathcal{E}_c} \mathbb{E}(e_i)$ is the sum of expectation values of executing e_i to improve activity coverage, which is the same formula defined in model exploitation.

Q-table exploitation. A hyper-event is selected by probability P_Q:

$$P_Q(e_i) = exp(\frac{Q(e_i)}{\beta}) \quad / \sum_{e_i \in \mathcal{E}_c} exp(\frac{Q(e_i)}{\beta}) \qquad (6)$$

where β is the hyperparameter that adjust the randomness of the strategy, which is set to 0.1 in our practice.

Example. In Figure 2(a), on Activity 2, e4 and e6 have not been executed yet and e6 is in the model M. In this case, e1 (on Activity 1) will be given a higher reward because e1 can lead to interesting actions (e4 and e6). Similarly, e8 is also a new event (not in M) on Activity 3 after executing e5, thus e5 is also given a higher reward. Assume e1, e2 and e3 are all executed for many times after a while, and Activities 2, 3 and 4 are all covered (all events have been included in the model M). In this case, the Q-table is used to make decision on event selection: e1 is likely to be selected as it has the highest rewards indicating that it can reach deeper activities.

2.4 Fastbot2's Implementation

Fastbot2 is implemented as a fully automated tool and consists of client and server modules. The client reuses the GUI tree dumping and action execution capabilities of Ape [5] to interact with the app. The server written in GoLang performs event selection and supports multi-device collaboration mode (which allows multiple clients to test the same app in parallel on multiple devices and share the same probabilistic model and RL agent).

3 EVALUATION

We evaluate the effectiveness of Fastbot2 by comparing it with two popular model-based tools, Stoat [9] and Ape [5]; and Monkey [4], a popular industrial testing tool. We tried to include Q-testing [8], a recent reinforcement learning-based testing tool. However, Q-testing always fails with exceptions after a few minutes of testing our industrial apps. Thus, we do not compare with Q-testing. We investigate the following research questions:

RQ1: Test Effectiveness: *Is Fastbot2 able to achieve higher activity coverage and reveal more unique crashes than existing tools when applied in the scenario of continuous app version updates?*

We use our two popular apps, Douyin (short video) and Toutiao (daily news), as our subjects and selected ten recent consecutive

versions of Douyin (v19.7~v20.6) and Toutiao (v8.7.0~v8.7.9) for continuous testing. We ran Fastbot2 to test one version on 10 devices in parallel for 1 hour, and test the next version by reusing historical data from all previous runs. For the other tools, we ran them on each version on 10 devices in parallel as this is the typical usage scenario of these tools. We compare the achieved activity coverage and the number of uncovered unique crashes by each tool.

RQ2: Ablation Study. *Do model-based and learning-based event selection strategies both contribute to Fastbot2's overall performance?*

We test Douyin (v.19.7) and Toutiao (v.8.7.0) on 10 devices for 1 hour with the model-based strategy enabled only, the learning-based strategy enabled only, and both strategies enabled to evaluate their respective impact on Fastbot2's overall performance.

In **RQ1** and **RQ2**, 10 different Huawei, OPPO and Google Pixel Android devices are used to mitigate the device fragmentation issue.

3.1 RQ1: Test Effectiveness

Figures 3(a) and 3(b) show activity coverage achieved: bars give numbers of activities covered by different tools on each app version, and curves give accumulated numbers of activities covered after testing ten consecutive versions. Figures 3(c) and 3(d) give the similar information on the number of unique crashes revealed by these tools (we deduplicate crashes according to stack traces [9]).

We can see that Fastbot2 achieved the highest activity coverage on each single version of both apps (except Toutiao's v8.7.3) and the highest accumulated activity coverage across ten continuous versions for both apps. *It indicates that reusing the knowledge from previous testing runs can effectively improve activity coverage within the same time budget.* On the other hand, We can see that Fastbot2 uncovered many more crashes on Douyin than all other tools (all the crashes were confirmed as real bugs). Fastbot2 uncovered 2 fewer crashes than Ape on Toutiao. We find that most crashes found by these two tools on Toutiao are similar native crashes (*i.e.*, crashes triggered by the app but reside in native C++ libraries). *The overall result indicates that reusing the knowledge from previous testing runs can also effectively improve Fastbot2's bug finding ability.*

Figures 4(a) and (b) use venn diagrams to compare the differences between Fastbot2 and other tools in terms of the accumulated activity coverage after testing ten app versions. The overlapped part denotes the number of activities covered by both tools, while other parts denote the number of activities covered by the two tools alone, respectively. We can see that *Fastbot2 is indeed effective as it can cover many more unique activities than other tools.*

We note that Fastbot2 performed much better on Douyin than Toutiao, compared to Ape. This is because Douyin is much more complicated than Toutiao. For example, Douyin has more complicated features, *e.g.*, online shopping in the livestreaming room, video recording and editing. In contrast, Toutiao has simpler features (*e.g.*, news reading). As a result, Toutiao is more likely to reach saturated coverage within 1-hour testing by Fastbot2 and Ape.

We also note that Stoat did not perform well as we expected. After inspection, we find Stoat only generated around 300 events during 1-hour testing. The main reason is that Stoat kept querying the GUI tree and only generated the next event until the current GUI page became stable. However, many features of Douyin and Toutiao, *e.g.*, advertisements, profiles and user comments, are dynamically changing, which makes Stoat waste a lot of time for waiting.
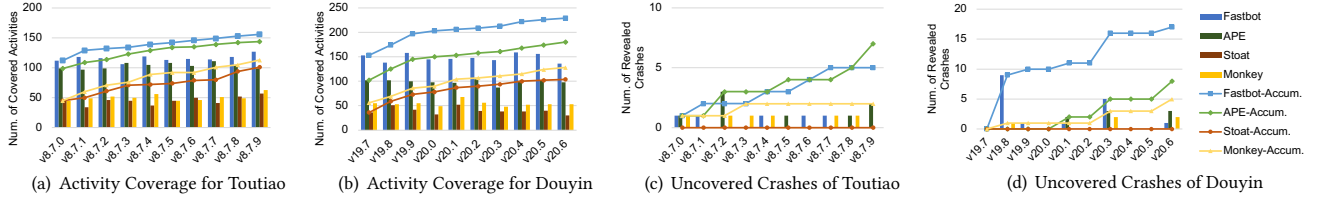
Figure 3: Testing results between Fastbot2 and other tools in terms of activity coverage and bug finding.

Figure 4: Differences of accumulative activity coverage.

3.2 RQ2: Ablation Study

Figure 5(a) and (b) shows the activity coverage of Toutiao and Douyin achieved by different testing strategies within 1-hour testing. Fastbot2 (denoted by "RL+PM") achieves 31.5% coverage for Toutiao, which is higher than both the model-based event selection strategy alone (28.5%, denoted by "PM Only") and the learning-based event selection strategy alone (29.0%, denoted by "RL Only"). Similarly, Fastbot2 achieves higher coverage for Douyin (20.8%) than model-based strategy alone (18.6%) and learning-based strategy alone (19.2%). The result indicates that both the model-based and learning-based event selection strategies contribute to Fastbot2's overall performance in improving activity coverage.
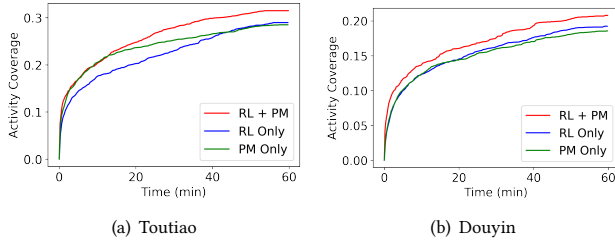

Figure 5: Comparing Fastbot2's internal strategies.

4 INDUSTRIAL DEPLOYMENT

To date, Fastbot2 have been deployed in the Continuous Integration pipeline at ByteDance for nearly two years. Fastbot2 is automatically triggered by nightly builds to obtain quick feedback on app quality when new code changes occur. We have received positive feedback from app development teams. For example, among all the developer-fixed bugs for Toutiao from September 1 to October 31, 2021, 50.8% of these bugs were uncovered by Fastbot2. Additionally, Fastbot2 can cover 80% of the hot-spot activities in Toutiao that are frequently visited by online users. These results corroborate Fastbot2's strong effectiveness.

5 RELATED WORK

We focus on discussing the industrial practice of automated Android GUI testing. Facebook Sapienz [6] adopts search-based testing to

improve code coverage and fault detection. They also extract information from crowd-based testing to enhance Sapienz [7]. WeChat WCTester [12, 13] adopts Monkey-based random testing. WCTester allows human testers to specify blacklisted widgets and define GUI event sequences to improve coverage. However, our work is significantly different from the prior work. First, Fastbot2 mainly adopts model-based testing (enhanced by a learning-based algorithm). Second, Fastbot2 reuses the knowledge from the historical exploration data to resolve the practical needs of continuous testing, which have not been considered by prior work.

6 CONCLUSION

This paper presents a reusable automated model-based GUI testing technique for Android enhanced by reinforcement learning to satisfy the practical needs of continuous testing. Our implementation Fastbot2 outperforms the three state-of-the-art testing tools in both activity coverage and bug finding in the scenario of continuous testing on two popular apps Douyin and Toutiao. Fastbot2 has been successfully deployed in the CI pipeline at ByteDance and received positive feedback on its ability of improving app quality.

REFERENCES

[1] AppBrain. 2022. . Retrieved June 3, 2022 from https://www.appbrain.com/stats/number-of-android-apps
[2] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?. In ASE. IEEE, 429–440.
[3] Kristopher De Asis, J Hernandez-Garcia, G Holland, and Richard Sutton. 2018. Multi-step reinforcement learning: A unifying algorithm. In AAAI, Vol. 32.
[4] Google. 2021. UI/Application Exerciser Monkey. Retrieved March 3, 2021 from https://developer.android.com/studio/test/monkey
[5] 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 ICSE. IEEE, 269–280.
[6] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In ISSTA. 94–105.
[7] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In ASE. IEEE, 16–26.
[8] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In ISSTA. 153–164.
[9] 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 ESEC/FSE. 245–256.
[10] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking Automated GUI Testing for Android against Real-World Bugs. In ESEC/FSE. to appear.
[11] 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 ASE. IEEE, 738–748.
[12] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for android: Are we really there yet in an industrial case?. In ESEC/FSE. 987–992.
[13] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated test input generation for android: Towards getting there in an industrial case. In ICSE-SEIP. IEEE, 253–262.