NATE: A Network-Aware Testing Enhancer for Network-Related Fault Detection in Android Apps

Yuanhong Lan, Shaoheng Cao, Yifei Lu*, Minxue Pan*, Xuandong Li
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Software Institute, Nanjing University, Nanjing, China
{yhlan, shaohengcao}@smail.nju.edu.cn, {lyf, mxp, lxd}@nju.edu.cn

***Abstract*—As Android apps become increasingly dependent on network services, Network-Related Faults (NRFs) are gradually more prevalent and severely degrade user experience. These faults are typically scattered across apps and require complex, often non-trivial network patterns to trigger, which makes their detection challenging. To date, we still lack a general and in-depth understanding of NRFs in real-world Android apps. To fill this gap, we conduct the *first* empirical study on 154 real-world network-related bugs collected from 42 diverse, representative Android apps, investigating their characteristics, influences, triggering patterns, and origins. Our study reveals several notable findings and practical implications to guide future research on detecting and mitigating NRFs. Motivated by the empirical results and the limitations of existing Android testing approaches—namely, the lack of targeted network events and efficient injection mechanisms—we propose NATE, a novel Network-Aware Testing Enhancer that augments existing general Android testing approaches for NRF detection. NATE leverages curiosity-driven reinforcement learning to provide network-aware guidance and to inject effective network events, enabling testing approaches to explore network-related extra app functionalities and detect NRFs. When integrated with two state-of-the-art general Android testing approaches, experiments conducted on 12 large, active apps demonstrate the effectiveness and efficiency of NATE, with 1.7-5.7× as many faults detected, as well as 8.8% and 12.5% more code covered. Among the network-related faults detected by NATE, 21 have been explicitly confirmed as real-world bugs by the developers (six of which have already been fixed), where 16 of them were first reported by NATE. Notably, none of the 21 bugs were detected by the original general testing approaches, demonstrating the unique contributions of NATE.**

***Index Terms*—Android app, mobile testing, network-related fault, reinforcement learning**

I. Introduction

Mobile applications have become integral to everyday life. In recent years, with the increasing abundance of app functions, modern mobile apps rely much more heavily on network services. Recent statistics [1] report that there are more than four billion mobile users in 2025 worldwide, with about four hours on average spent online with mobile apps daily.

However, the unpredictable and ever-changing network environment has brought great challenges to these apps, leading to increasingly emerging Network-Related Faults (NRFs) that appear at any time and severely harm user experience [2]–[11]. For the popular browser *Firefox* alone, 22 NRFs have been reported and confirmed by the developers as real-world bugs during the past three years, with 14 of which were found just in the past year. Specifically, network-related faults refer to *unexpected app behaviors under abnormal network conditions* (only faults explicitly confirmed by the developers are called bugs in this paper). A simple yet highly damaging case is a direct app crash when launched without a network [2], [3]. Another typical recent example is from Issue #17551 [4] of the popular flashcard app *AnkiDroid*, where auto synchronizations under slow or crappy networks lead to annoying uncancellable dialog or even severe Application Not Responding (ANR). This bug was thought to be a big deal since such network-related bugs typically lead to substantial degradation of user experience. Developers have tagged it with the highest priority, though this bug has not been completely fixed so far due to its inner complexity. To maximize user retention, both developers and researchers have placed significant emphasis on quality assurance in Android apps, since merely Android apps account for over 70% of the mobile market [12]. Meanwhile, given the severe influence of such bugs, it is particularly critical to design comprehensive testing frameworks that specifically target network-related functionalities to address faults stemming from improper network handling, thereby improving app reliability and strengthening market competitiveness.

Unfortunately, we still lack a general and in-depth understanding of such NRFs. To fill this gap, we conducted the *first* empirical study on real-world network-related bugs. With initial keyword screening, 1115 candidate issues were filtered from the 42 diverse benchmark apps [13]. After about three Man-Months of analysis and reproduction attempts on each candidate, 154 real-world network-related bugs were identified. The study revealed a number of interesting findings: ① Modern apps involving network services, including more than resource interactions, cloud services, and account links, suffer from NRFs. ② NRFs can lead to *app crashes*, *exceptional app logics*, *abnormal UIs*, and *deficient app logics*, seriously degrading user experiences. ③ NRFs are generally triggered by three patterns: Disconnect Before the Event, Disconnect During the Event, and Unstable Network. ④ Network is becoming more than a system state in modern apps, but has introduced many tailored network-related extra app functionalities to deal with abnormal networks, such as error-handling modules, offline messaging, temporary offline editors, etc. In conclusion, network-related faults mainly originate from the

∗ Corresponding authors.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00118

insufficient handling of abnormal networks or faulty network-related extra app functionalities. Such findings serve as the basis of our approach, as well as motivate future research.

While existing Android testing approaches, no matter general [14]–[18] or targeted [19]–[22], lack effective strategies to deal with such a new challenge. Although some [14], [23] include a few random attempts, there are several limitations. First, network events are triggered at a very low probability, inefficient for detecting NRFs. Second, only random network-connection-change events are considered, not enough to trigger real-world NRFs requiring complex triggering patterns. More importantly, without proper guidance, indiscriminate injection of network events seldom detects NRFs and instead undermines the effectiveness of the normal testing process.

To address the aforementioned challenges, we propose NATE, a Network-Aware Testing Enhancer for NRF detection. Considering leveraging the superiority of existing State-Of-The-Art (SOTA) general Android testing approaches, NATE is positioned as an enhancer and can be integrated with them in a loosely coupled manner. Three effective network events are proposed based on our empirical findings. Targeting the curiosity for exploring network-related extra app functionalities, we apply real-time network monitoring to identify network-related app events, and leverage Reinforcement Learning (RL) with curiosity-driven exploration to enable effective network-aware guidance for network event injection.

NATE has been implemented and integrated with two SOTA general Android testing approaches, deep-RL-based DQT [17] and model-based APE [18]. Experiments were conducted on 12 active apps across 10 categories with average Executable Lines Of Code (ELOC) over 100k. With the enhancement of NATE, DQT and APE detect 1.7-5.7 times faults, along with 8.8% and 12.5% more code coverage. NATE uniquely contributes 4.4 fatal errors, 13.6 all-type errors, and 4% instructions of each app on average. So far, 21 of the network-related faults detected by NATE have been explicitly confirmed as real-world bugs by the developers, six have been fixed, with 16 of them first reported by NATE, demonstrating NATE's effectiveness in real-world practice. Notably, none of the 21 bugs were detected by the original testing approaches, which further exhibits the unique ability of NATE.

In summary, we have the following main contributions:

- The *first* empirical study on network-related faults in Android apps to the best of our knowledge. 154 real-world network-related bugs are thoroughly analyzed, which serve as the basis of our approach and support future research.
- A testing enhancer guided by lightweight curiosity-driven exploration, named NATE, to empower existing general Android testing approaches to detect network-related faults.
- Empirical evaluations on 12 large, active apps have demonstrated the effectiveness and efficiency of NATE, as well as the real-world practical value. We have made all the artifacts publicly available [24] for replication and further research.

II. Empirical Study

While NRFs in Android apps have been occasionally reported, to the best of our knowledge, no existing research has specifically targeted these issues or developed dedicated detection techniques. Motivated by this gap, we conduct an empirical study to investigate the severity and characteristics of NRFs, as well as to surface the challenges inherent in their detection. Specifically, to build up a general and in-depth understanding of real-world NRFs in Android apps, we conducted the study with the following research questions:

- **Q1**: Do modern apps suffer from network-related faults?
- **Q2**: How do network-related faults interfere with the apps?
- **Q3**: How are network-related faults triggered in general?

A. Study Setup

Given the highly fragmented, diverse nature of Android apps, a systematic method was applied to select representative and referential real-world NRFs.

First, we adopted apps from the newly released AndroTest24 App Benchmark [13], which is based on the famous AndroTest App Benchmark [25] and incorporates the contributions of more than ten later open-source app benchmarks. There are 42 actively maintaining open-source apps across 19 different categories, with ELOC from 759 to 174k, which exhibits compelling category and scale diversities. We chose open-source apps for our analysis as their fault repositories provide better transparency and traceability with important and complete traceable information, such as issues and pull requests.

Second, we filtered out historical NRFs from the issue repositories of all these apps. As there are hundreds of thousands of different kinds of issues, we employed the following filter for initial screening:

```
is:issue linked:pr (network OR internet
OR connection OR connectivity OR offline)
```

The filter includes three parts: ① The `is:issue` identifies issues. ② To reduce false positives, `linked:pr` is employed to retain only the ones that were finally accepted by the development team and become actual code contributions. ③ The last part contains five keywords. Based on the word frequency statistics of network-related official documents [26], the top three (i.e., `network`, `connectivity`, and `connection`) were included. Besides, as one of the important permissions of Android network access [27], word `internet` should be covered. In addition, during the process of screening with these four keywords, another important scenario `offline` was frequently mentioned by users. Therefore, we added it later and restarted our screening. While some apps have their specialized issue repositories (e.g., the *Bugzilla* for *Firefox* [28]), we utilize their advanced search function to achieve a similar screening. Eventually, 1115 candidates were achieved.

Third, to identify real NRFs, each of the 1115 issues was then analyzed manually by two independent authors who reviewed, classified, and attempted to reproduce the behavior. The two reviewers then cross-checked their labels through mutual proofreading, and any remaining discrepancies were resolved by the intervention of a third author for discussion until all three reached consensus. Given the complexity of these faults and the difficulty of the reproduction attempts, this step took approximately three Man-Months and eventually resulted in the identification of 154 network-related bugs.

B. *Study Result*

Table I lists the number of candidates (Cand.) and identified network-related bugs on apps with at least one candidate, along with the app category, GitHub stars (if available), and Google Play Downloads (GPD, if available). The full list of all these identified bugs is provided on the website [24].

TABLE I: The Source of the Identified Network-Related Bugs

App Name	Category	Star	GPD	Cand.	Bugs
Firefox	Browser	10.0k	100m+	346	28
WordPress	Productivity	3.1k	10m+	181	48
Wikipedia	Reference	2.7k	50m+	108	8
NewPipe	News	35.1k		105	20
AntennaPod	Audio	7.3k	1m+	103	10
AnkiDroid	Education	10.0k	10m+	101	11
K9Mail	Communication	12.5k	5m+	51	5
Signal	Communication	27.6k	100m+	49	13
AmazeFileManager	System Tools	5.8k	1m+	33	1
ConnectBot	System Tools	2.8k	5m+	10	1
SuntimesWidget	Daily	394		10	1
Materialistic	News	2.3k		9	6
RunnerUp	Fitness	842	50k+	5	1
DuckDuckGo	Browser	4.3k	50m+	1	1
Vanilla	Music	1.3k		1	0
AlarmClock	Daily	579	1m+	1	0
MoneyManagerEx	Finance	589	10k+	1	0
	TOTAL			1,115	154

As shown in Table I, the identified numbers vary considerably across apps, reflecting differences in app characteristics, history, user bases, and community activity. The famous browser *Firefox* yielded the largest number of candidates, reflecting a highly traceable issue base produced by hundreds of millions of users and an active open-source community. By contrast, the website-builder *WordPress*—which heavily depends on network services and provides a comprehensive offline mode—exhibited the greatest number of fixed network-related bugs. Surprisingly, several apps that we initially assumed to be primarily local (e.g., the file manager *AmazeFileManager* and the sports recorder *RunnerUp*) also manifested network-related bugs. Our follow-up analysis revealed that, with the rapidly increasing demand of contemporary users for cloud synchronization, cloud storage, social networking, etc., modern apps have gradually included corresponding online functionalities in their evolution. For example, *AmazeFileManager* has allowed users to build cloud connections and set up their own servers for synchronization, while *RunnerUp* has enabled users to connect to different kinds of accounts for daily sharing or conduct cloud storage.

Answer to Q1: Modern apps involving network services, including more than resource interactions, cloud services, and account links, suffer from network-related faults.

We statistically categorized the consequences of these network-related bugs based on how developers labeled and fixed each, where a four-tier classification scheme was applied:

- **App Crash**: Bugs that directly caused the app to crash.
- **Deficient App Logic**: Bugs without any associated traceable error stack trace or exception.
- **Abnormal UI**: Remaining bugs that were explicitly defined or tagged by the developer as UI problems.
- **Exceptional App Logic**: All the remaining other bugs.

The classification is straightforward, as bugs of these four types exhibit distinct and identifiable characteristics. This is also the reason for their frequent adoption in numerous well-known Android app projects, such as *AnkiDroid* and *NewPipe*.

Among all the bugs, 17.5% (27) are fatal ***app crash***, 46.1% (71) are ***exceptional app logic***, 22.1% (34) are ***abnormal UI***, and 14.3% (22) are ***deficient app logic***.

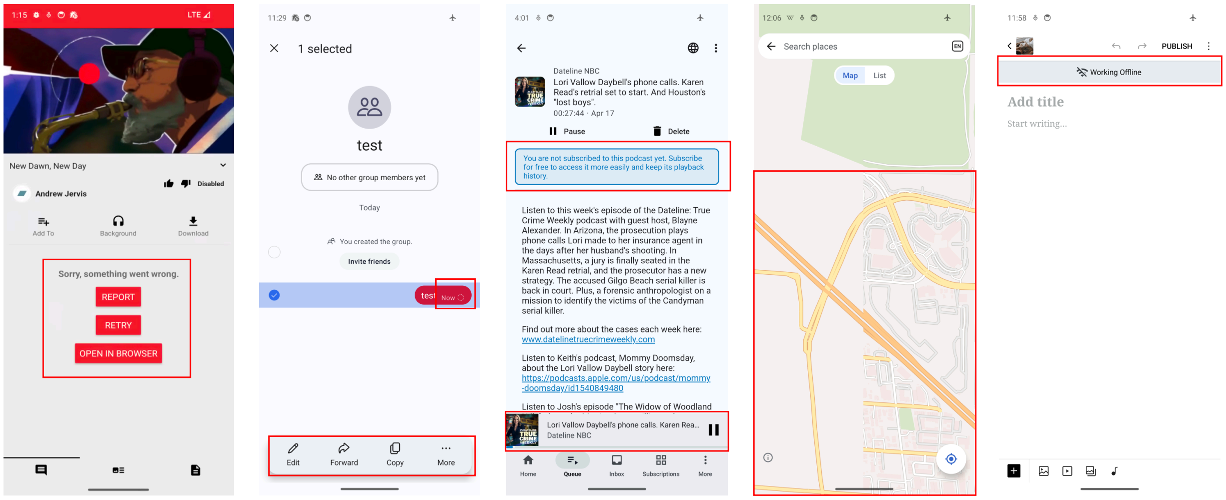
With very uncontrollable user networks introduced, various unexpected problems then emerge. For example, in Bug #11202 of *Signal* [5], the unstable network caused the user to encounter a conversation with an *Unknown* person, leading to an ***app crash*** each time trying to open it. The user expressed considerable anxiety and did not dare to operate much for fear of further data loss, as it was his daily primary social account. Another ***exceptional app logic*** bug comes from Bug #1891257 of the app *Firefox* [6], where the browser failed to load sites after the user switched between Wi-Fis and mobile data. The user complained that the situation had lasted months and had become intolerable. Fortunately, except for ***deficient app logic*** that is entirely dependent on subjective judgment by users and developers, the other three types that account for over 85% of all have traces in the stack with exceptions. For example, Bug #10000 of *AnkiDroid* [7], a non-crash ***abnormal UI*** bug, can be traced with *SSLException* in the stack. By contrast, ***deficient app logic*** bugs like #1942 of *AntennaPod* are typically decided by heated discussions between users and developers [8].

Answer to Q2: Network-related faults can lead to *app crashes*, *exceptional app logics*, *abnormal UIs*, and *deficient app logics*, seriously degrading user experiences.

Through the analysis of the provided triggering steps and our reproduction attempts, the trigger modes of these network-related bugs can be divided into the following three patterns.

P1: Disconnect Before the Event (43.5%, 67) The most common pattern is the offline execution of a specific app function, specifically manifested as disconnecting the network before executing the entry event of the app function. For example, a fatal crash from *AnkiDroid* (#12711) was reproduced by turning off the network before trying to sync data [9].

P2: Disconnect During the Event (19.5%, 30) This pattern is also under offline, but with network interruption after the specific event has executed (the particular app function has started). Typical scenarios include website loading, video streaming, and file downloading. At Bug #8898 of *New-*



Sorry, something went wrong.

No other group members yet

You created the group.

Lori Vallow Daybell's phone calls. Karen Read's retrial set to start. And Houston's "lost boys".

You are not subscribed to this podcast yet. Subscribe for free to access it more easily and keep its playback history.

Listen to this week's episode of the Dateline: True Crime Weekly podcast with guest host, Blayne Alexander. In Arizona, the prosecution plays phone calls Lori made to her insurance agent in the days after her husband's shooting. In Massachusetts, a jury is finally seated in the Karen Read retrial, and the prosecutor has a new strategy. The accused Gilgo Beach serial killer is back in court. Plus, a forensic anthropologist on a mission to identify the victims of the Candyman serial killer.

Find out more about the cases each week here: www.datelinetruecrimeweekly.com

Listen to Keith's podcast, Mommy Doomsday, about the Lori Vallow Daybell story here: https://podcasts.apple.com/us/podcast/mommy-doomsday/id1540849480

Listen to Josh's episode "The Widow of Woodland

Lori Vallow Daybell's phone calls. Karen Rea...
Dateline NBC

Working Offline

① Error Handling ② Offline Messaging ③ Local Playing & Viewing ④ Partially Loaded ⑤ Offline Editor

Fig. 1: Examples of Network-Related Extra App Functionalities

Pipe [10], the app crashed with error reports due to a sudden network outage while the user was watching a video.

P3: Unstable Network (37.0%, 57) This pattern includes situations like network switches (e.g., from mobile data to Wi-Fi), slow networks, and high-latency networks. In this case, the network is often not completely down, but is usually randomly on and off. *AntennaPod* (#3196) provides an exemplary instance [11], where the app crashed after searching iTunes to add podcasts on a very slow network.

Answer to Q3: Network-related faults are generally triggered by three patterns: Disconnect Before the Event, Disconnect During the Event, and Unstable Network.

C. Additional Findings

Intuitively, many NRFs stem from insufficient handling of abnormal network conditions—for example, an app that assumes the presence of online resources may crash if launched without network access. On top of that, our study further reveals that for modern apps, **the network is becoming more than a system state, but another important part of app functionalities**, due to the growing reliance on cloud services. End users now demand that apps tolerate complex and unstable network environments, prompting developers to integrate extra modules for caching, recovery, and temporary offline use. We refer to these as **network-related extra app functionalities** in this paper. Such functionalities that can only be triggered by abnormal networks, however, may be buggy themselves and introduce extra unexpected NRFs.

Figure 1 presents several real-world examples of network-related extra app functionalities: ① Many modern apps have introduced error-handling modules to deal with network problems, allowing users to retry, report issues, or try alternatives. ② For social modules, it is common to cache offline messages when the network status is not satisfactory and then try to emit them when the network is somehow recovered. While temporarily offline, users are also allowed to delete or edit their cached unsent messages. ③ In music-playing or information-browsing modules, local playing or viewing based on cache or downloads is common. To improve user experience, these modules preload some of the content in advance and later load it even if the device is temporarily offline. In this case, users would only be provided with limited functionalities (e.g., not allow them to comment or download) compared to those when online. ④ In app modules like maps, partially loaded is common under sudden offline, but leads to functional variations. For example, only paths between loaded locations are allowed, with basic shortest paths calculated without considering real-time traffic conditions. ⑤ Offline editors are often introduced as staging areas for the online editors and will be restored somehow after the network recovers. They differ from online editors in many ways: they provide only basic functionalities, scale with native services instead of cloud services, require additional local caching, and so on.

Network-related extra app functionalities, together with their underlying code logics, are triggered only under particular network states at specific points in time. This timing-dependent activation makes them difficult to exercise reliably during testing and thus becomes a significant source of NRFs.

D. Discussion

Based on the above analysis, **to conclude, network-related faults, on the one hand, come from the insufficient handling of abnormal networks, and on the other hand, come from the abnormal functioning of these network-related extra app functionalities developed for dealing with unstable networks or device offline.**

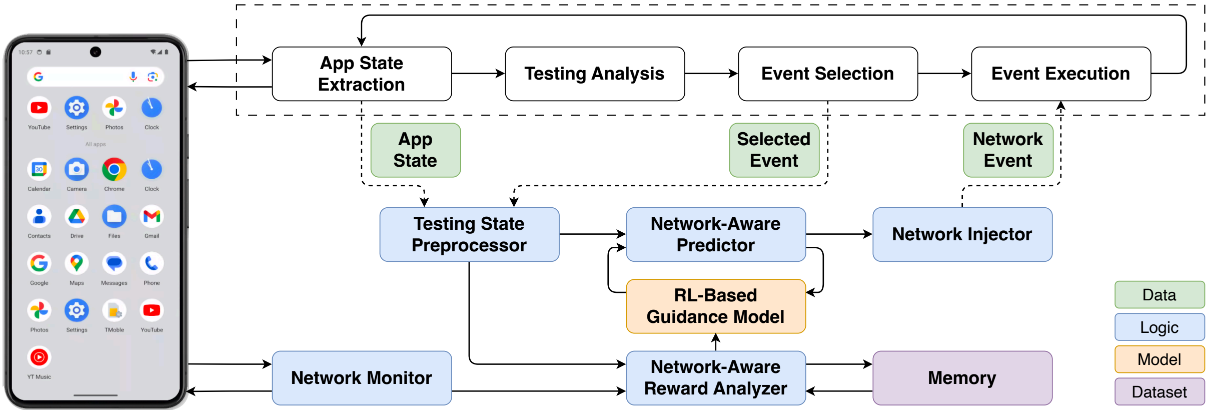


Fig. 2: The Workflow of NATE

With the increasing dependence of modern mobile apps on the network, testing network-related extra app functionalities and detecting NRFs are becoming more and more imperative for app quality assurance. In practice, however, developers are struggling to cope with the ever-expanding complexity of modern mobile apps, let alone simulate the changeful user network and design exponentially growing test cases. Meanwhile, existing Android testing approaches lack effective testing strategies to excavate the NRFs hidden in apps. Such situations motivate us to develop an automated testing approach for NRF detection.

III. Network-Aware Testing Enhancer

Effective fault detection in Android apps requires thorough app exploration, while existing SOTA general Android testing approaches [14]–[18] have already demonstrated impressive effectiveness in extensive practice. To leverage such superiority, we want a way to empower them for further NRF detection. However, it is a non-trivial task, facing the following three main challenges based on the investigation of our prior study:

- **C1**: How to integrate with existing SOTA general Android testing approaches in an appropriate way?
- **C2**: What kinds of network events should be injected to facilitate effective network-related fault detection?
- **C3**: How to guide the injection of network events at the right timing for efficient network-related fault detection?

To address the above challenges, we propose NATE, a Network-Aware Testing Enhancer, capable of integrating with any existing general testing approaches flexibly. The workflow of NATE is presented in Figure 2. Our approach is three-fold: ① To address C1 (Section III-A), we abstract existing general Android testing approaches at a high level and achieve lightweight integration with them. ② To address C2 (Section III-B), we design effective network events based on the triggering patterns discovered in our empirical study and further reproduction attempts of existing network-related bugs for event parameters. ③ To address C3 (Section III-C), we leverage a lightweight, flexible, and adaptable RL-based strategy. Benefit from both real-time network monitoring and the motivation of adaptive curiosity-driven testing, NATE can not only identify network-related app events automatically in real-time, but also empower testing approaches to explore the unexplored network-related extra app functionalities, with a reward function designed to balance network-relevance and the discovery of new, potentially faulty network-related states.

A. *Typical Android Testing Framework*

Most existing general Android testing approaches [15], [17], [18], [29]–[31] generally utilize loop execution. Each testing step, as depicted in the dashed box at the top of Figure 2, can be concretely abstracted into four stages: ① *App State Extraction*: The first step is to extract the current state of the App Under Test (AUT). Screenshots [15], [32] or GUI hierachies [17], [18], [30], [31] are retrieved and further preprocessed to be an identified *App State*. ② *Testing Analysis*: Based on the *App State*, different testing approaches conduct testing analysis in their own ways, such as model refinement [18] or model training [17]. ③ *Event Selection*: At this stage, testing approaches employ their own strategies to select a proper event for execution. ④ *Event Execution*: The *Selected Event* is executed and triggers app state transitions.

Based on the principle of minimal dependency, NATE only acquires the *App State* and *Selected Event* at each test step from the testing approach for analysis, and then injects network events along with the event execution phrase, so as to conduct loosely coupled cooperation with the target approach.

B. *Network Event Injection*

Based on the three triggering patterns discovered in our empirical study (defined as P1, P2, and P3 in Section II-B), three network events (denoted as E1, E2, and E3) are correspondingly designed. Since the major functionalities of network-dependent app modules require normal network conditions, while abnormal networks inevitably interfere with their normal operation, we have to meticulously control the duration parameters of each network event. Therefore, we took further reproduction attempts on the collected network-related bugs.

For P1, the disconnection durations were attempted from two to six seconds (in one-second increments), which disconnected just before the targeted event was executed. We found that due to the time acquired for app function loading, basic retries, and timeout tolerance, such faults required at least five seconds to be stably reproduced.

For P2, we tried time gaps from 0.5 to three seconds (in 0.5-second increments) between the event execution and the disconnection. We observed that it takes at least two seconds to guarantee complete loading of the app function. Aligning with E1, such faults were stably reproduced by recovering the network at least three seconds after the disconnection.

For P3, we simulated unstable networks by random network interruptions. To align with E1 and E2, the duration of E3 was also set to five seconds. We divided five seconds equally into four periods. At the beginning of each period, we interrupt the network (turn on Airplane Mode) for a random x seconds and then recover. The upper bound of x was experimented with from 0.1 to 0.5 seconds (in 0.1-second increments), where we found that 0.2 seconds is the most stable one for reproduction.

Consequently, the network events are designed as follows, and will be randomly injected by *Network Injector* of NATE:

- **E1**: Disconnect the network for five seconds.
- **E2**: Wait for two seconds and disconnect for three seconds.
- **E3**: Within five seconds, repeatedly disconnect for a random $x \in (0, 0.2]$ seconds and connect for $(1.25 - x)$ seconds.

C. *Network-Aware Guidance*

An intuitive way for network enhancement is to randomly inject network events while running a general testing approach. However, due to the lack of proper guidance, such a way is inefficient for three main reasons. First, as many app functionalities are native and network-independent, injecting network events with them is unnecessary and redundant. Second, the exploration of network-related extra app functionalities requires injecting network events with specific app events, which is inefficient via a random strategy. What's worse, random injections severely interfere with the normal operation of the network-dependent app functionalities, resulting in worse testing performance of the general testing approach. For example, some app modules become directly unavailable during a network outage. With network outages at any time, the test process of the general testing approach will be interrupted casually, leading to unexpectedly worse performance. To address the above limitations, our NATE applies network-aware guidance from two main aspects.

Network-Related Events To avoid redundant injections for unrelated ones, we need to identify network-related app events. We design a *Network Monitor* to monitor real-time network traffic changes of the AUT and collect network flow data. We can infer whether a certain app event is network-related by analyzing whether new network traffic is generated after its execution. In practice, to avoid coincidental misjudgments in rare cases, such as the coincidental background downloading, we also leverage the benefits of RL to learn from the results of multiple interactions.

Curiosity-Driven Exploration The collaboration of RL and curiosity-driven strategies has demonstrated superior performance on testing explorations [17], [31], [33]: ① Not necessitating pretraining models, RL [34] focuses on leading an agent to learn from runtime interactions on how to interact with the environment to achieve the preset goals, which is dynamic and adaptive, suitable for testing complex modern apps with dynamic changes and loadings. ② For testing, curiosity-driven strategies generally encourage the exploration of currently untested states and app events of the AUT, which is also adaptive throughout the testing process as the tested set will keep updating. ③ The advantage is further enlarged when combined with the long-term reward transfer of RL, enabling the testing approach to take a longer-term view of exploration, rather than just confining itself to only one-step feedback of the current state. Our NATE leverages such benefits, encouraging effective exploration of the network-related extra app functionalities, rather than merely enumerating every network-related app event and injecting network events.

D. *NATE Implementation*

The testing problem is formalized as a Markov Decision Process with a 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$: ① A state $s \in \mathcal{S}$ represents the current status of the AUT. ② An action $a \in \mathcal{A}$ is an app event of the current state. Both s and a are provided by the employed general testing approach. ③ The transition function $\mathcal{P}$ determines which state s_{t+1} will reach after the agent take action a_t at state s_t. Through network events injection, the transition function $\mathcal{P}(s, a)$ is further extended to $\mathcal{P}(s, a, c)$, where c indicates the network status. This extension excavated more previously neglected transitions and enables the exploration of additional states. ④ At each step, after reaching a new state s_{t+1}, a step reward r_t is provided by the reward function $\mathcal{R}$, indicating our evaluation for step t. To conduct network-aware guidance, the reward function $r_t = R(s_t, a_t, s_{t+1})$ of NATE is defined as follows:

$$r_t = \begin{cases} \text{-1}, & a_t \text{ is not network-related,} \\ \text{+2}, & a_t \text{ is network-related \& } s_{t+1} \text{ is new,} \\ \text{-0.5}, & a_t \text{ is network-related \& } s_{t+1} \text{ is visited.} \end{cases} \quad (1)$$

where a median negative reward of -1 is given when an action is not network-related; a high positive reward of +2 is awarded for actions that are network-related and guide to new states; otherwise, a low negative reward of -0.5 is given.

Our design goal for the reward function is to balance general app exploration and enhanced network-related exploration. It is two-fold in general: ① Network-Related Events Recognition: Penalize network-irrelevant events to recognize only network-related ones. ② Network-Related Exploration: Encourage exploring new network-dependent states and prevent inefficient repeat visits that frequently disrupt normal app behavior and hinder deeper exploration. Such a design encourages NATE to learn optimal timings at runtime for injecting network events, uncover previously unexplored network-dependent functionalities, and detect network-related faults.

We apply Q-learning [35], a model-free RL algorithm based on Q-function $Q^{\pi}(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t, \pi]$, with the goal of learning a policy π to maximize the expected discounted cumulative reward $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ after taking action a_t at state s_t. The training process is realized based on the Bellman equation to combine current and long-term benefits:

$$Q^{\pi}(s_t, a_t) \leftarrow r_t + \gamma \underset{a'}{max}\, Q^{\pi}(s_{t+1}, a'), \tag{2}$$

where the discounted factor γ and the learning rate α are set to 0.99 and 0.1, following the optimized practice [31]. To precisely record the evaluation results of each app event, a Q-Table is employed, with all initial values set to 0.

As presented in Figure 2, NATE works as follows at each loop step: ① The *Testing State Preprocessor* extracts the current interaction pair (s_t, a_t, s_{t+1}, a_{t+1}) from the general testing approach, where s_{t+1} is the current *App State* and a_{t+1} is the current *Selected Event*. ② The *Network-Aware Reward Analyzer* achieves the network analysis results from *Network Monitor* and compare the s_{t+1} with visited states in the *Memory* to decide reward r_t. The *Network Monitor*, which we have discussed earlier, is implemented by ADB [36] commands and employed to judge whether an event is network-related. The *Memory* of NATE is utilized to record all the visited states at the current test, based on which we can judge whether the current state is new or visited. The *Guidance Model* is then trained. ③ The *Network-Aware Predictor* evaluates (s_{t+1}, a_{t+1}) and calls *Network Injector* if the Q-value, which indicates the predicted subsequent potential contribution of injecting network events, exceeds threshold ζ. The ζ is set to 0.05 by default to ensure normal operation of the first try of each app event ($\zeta > 0$) as well as provide more tolerance and acceptable attempts for each network-related app event.

E. NATE Integration

NATE is realized in Python with ADB [36] and can be seamlessly incorporated with existing general Android approaches in a loosely coupled and lightweight manner to extend the ability for effective NRF detection. Since NATE requires only the state and event identifiers from each step of the underlying tool, integration can be achieved in two ways: ① *Intrusively*, by importing the NATE library and invoking it at each step to transmit the required data; or ② *Non-intrusively*, by monitoring the real-time log output of the target tool to retrieve the necessary step-wise identifiers and work independently with it. More details are provided online [24].

IV. Evaluation

Our evaluation focuses on the following research questions:

- **RQ1: Effectiveness Evaluation.** How effective is NATE when applied to SOTA general Android testing approaches? How much does NATE uniquely contribute?
- **RQ2: Ablation Evaluation.** How much does the network-aware guidance contribute to the effectiveness of NATE? Does such a strategy lead to better efficiency?
- **RQ3: Real-World Practice.** Do the network-related faults detected by NATE reveal real-world bugs that developers are concerned about? How are they triggered in practice?

A. Experimental Setup

Baseline Approaches. Based on the recent comprehensive evaluation [13] of the SOTA general Android testing approaches, we selected the best-performing two approaches as representatives, DQT [17] and APE [18]. DQT is based on deep RL, representing recent advancements, while APE represents the SOTA in prominent model-based testing. We employed the default settings of these approaches and integrated NATE with them by the ways discussed in Section III-E.

Benchmark Apps. We adopted the newly released App Benchmark AndroTest24 [13], which is the superset of the benchmarks of DQT [17] and APE [18]. Such 42 apps are widely recognized in prior research studies [16], [25], [31], [38], capable of instrumentation for precise fine-grained code coverage. By further checking network-related permissions [39] and picking apps with at least one new commit in the past year, 12 active network-related apps were identified. We employed their latest version with the latest commit.

Table II presents an overview of the benchmark apps, along with the app version, source repository link, GitHub stars (if available), Google Play Downloads (GPD, if available), Google Play Rate (GPR, if available), the app category, and the app scale with activities (ACTV), methods (METH), ELOC, instructions (INST), calculated by JaCoCo [37].

As depicted in Table II, the 12 apps span 10 different categories, with app scale varying from 9k to 277k ELOC, demonstrating diversity on both categories and app scales. The average ELOC of the apps reaches over 100k, while more than 80% (10/12) are available on the Google Play Store with an average rating of 4.5/5.0, revealing their representativeness of modern large, complex real-world apps. There are also well-known apps like *Signal*, *Firefox*, boasting over 100 million downloads just on the Google Play Store.

Experimental Environment. Experiments were conducted on the same physical machine. Android Emulators [40] were employed for stability and applicability. Due to the high resource requirements of the emulators, we use only four emulators in parallel to ensure smooth operation of each and more reliable experiments. To reduce the impact of emulator variability, emulators originated from the same parent, and apps were evenly distributed and bound to an emulator. An initial snapshot was also captured for each emulator to ensure the consistent start status of each test. For apps (e.g., Signal, WordPress) that require a login status for normal operation, the logged-in apps with a test account are provided together with the initial snapshot. As a result, every test begins from this initial snapshot with the account already authenticated. We selected Android 14, the most popular Android version with the largest market share worldwide at present [41]. Note that APE is not compatible with this new Android version. Therefore, we followed the setting in AndroTest24 Study [13] and ran APE on Android 9, while DQT on Android 14.

TABLE II: Overview of the Benchmark Apps

App Basic Information									App Scale (by JaCoCo [37])			
ID	App Name	Version	Source	Star	GPD	GPR	Category	ACTV	METH	ELOC	INST
A1	Signal	7.39.0	https://github.com/signalapp/Signal-Android	27.6k	100m+	4.4	Communication	90	67,413	277,200	1,641,092
A2	Firefox	138.0a1	https://github.com/mozilla-firefox/firefox	10.0k	100m+	4.6	Browser	22	64,360	245,851	1,471,339
A3	WordPress	25.7.2	https://github.com/wordpress-mobile/WordPress-Android	3.1k	10m+	4.4	Productivity	126	52,098	209,885	1,123,645
A4	DuckDuckGo	5.228.2	https://github.com/duckduckgo/Android	4.3k	50m+	4.7	Browser	105	56,969	176,054	1,073,164
A5	Thunderbird(K9Mail)	11.0	https://github.com/thundernest/k-9	12.5k	5m+	3.8	Communication	31	14,493	56,590	389,748
A6	Wikipedia	2.7.50527	https://github.com/wikimedia/apps-android-wikipedia	2.7k	50m+	4.5	Reference	61	14,191	56,135	355,777
A7	SuntimesWidget	0.16.5	https://github.com/forrestguice/SuntimesWidget	394			Daily	40	11,318	61,054	308,434
A8	AnkiDroid	2.21alpha13	https://github.com/ankidroid/Anki-Android	10.0k	10m+	4.7	Education	32	9,690	39,809	229,803
A9	NewPipe	0.27.6	https://github.com/TeamNewPipe/NewPipe	35.1k			News	14	14,687	38,249	164,002
A10	AmazeFileManager	3.10	https://github.com/TeamAmaze/AmazeFileManager	5.8k	1m+	4.5	System Tools	8	5,103	33,484	150,747
A11	AntennaPod	3.8.0	https://github.com/AntennaPod/AntennaPod	7.2k	1m+	4.8	Audio	12	5,609	31,038	131,779
A12	ConnectBot	1.9.10	https://github.com/connectbot/connectbot	2.8k	5m+	4.6	System Tools	11	1,209	9,023	70,774
Average								46	26,428	102,864	592,525

Experimental Configurations. Following the suggestions of AndroTest24 Study [13], we employ both fault detection and code coverage as evaluation metrics. According to our empirical study in Section II, stack traces of exceptions could cover over 85% of network-related bugs. The last type, *deficient app logic*, however, is not included in our evaluation as it is completely subjective and would introduce a mass of false positives. We adopt both Level F (severe fatal crashes [42]) and Level E (all potential app-related errors) Faults to include both crash and non-crash faults. The collection and analysis of faults were carried out according to the mature guidance [13]: ① runtime monitor of Logcat [43] and exception extraction, ② filter by app package, ③ deduplicate with the exception type and source code reference, and ④ categorization by scope. Apps were instrumented with JaCoCo [37], and we utilize the most fine-grained instruction coverage for more refined feedback. Experiments were conducted with one-hour tests and repeated five times each to mitigate the interference of randomness, where total fault detection and average code coverage were later calculated. For app *WordPress*, as it is not allowed to conduct Google sign-in on the self-built version due to security issues [44], we had to use the official app instead. Therefore, only fault detection is evaluated on this app.

B. RQ1: Effectiveness Evaluation

The evaluations results based upon DQT [17] and APE [18] are presented in Table III and Table IV. The first column under each metric shows the evaluation results of the approach itself, while the last column shows the one enhanced by NATE.

Fault Detection. With NATE's enhancement, DQT+NATE is able to detect 5.7 times Level F Faults and 2.4 times Level E Faults of DQT, while APE+NATE detects 2.6 times Level F Faults and 1.7 times Level E Faults of APE alone.

To ascertain the characteristics of such enhancements, we conduct further analysis on the types of detected Fatal Errors (i.e., Level F Faults), which are more severe and lead to app crashes. Figure 3 displays the distribution of the detected fatal faults from DQT+NATE and APE+NATE, along with the improvements compared to the original approaches at each fault category. Due to the considerable number of categories, only the top 14 categories are specified. Overall, with the enhancement of NATE: ① The detected fatal faults of DQT extend from five categories to 17, while those of APE extend from seven categories to 14. ② Compared to the presented distributions of the original general testing approaches [18], [23], the results under NATE intensification are more abundant, with many coming from uncommon categories, such as *RemoteServiceException* and *SocketTimeoutException*. ③ In common categories, such as *NullPointerException* and *IllegalStateException*, there is also a significant enhancement. While examining their stack context, we found that network problems triggered many of the faults indirectly. For example, a sudden network outage caused a variable to become *Null* due to the failed data fetch. Without proper checking or recovery during the subsequent variable transmission, this later led to *NullPointerException* in further passes, and the app crashed.

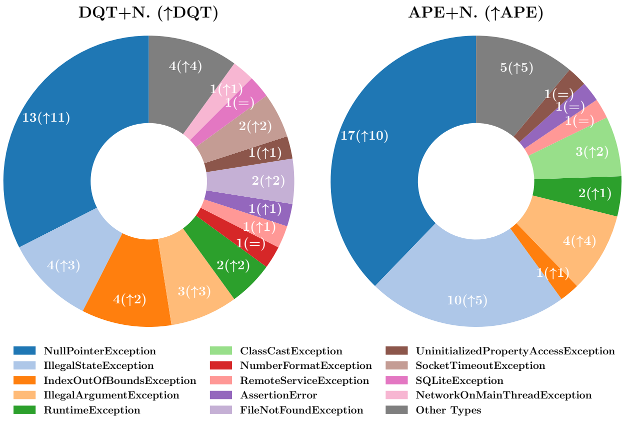


Fig. 3: Distribution of the Detected Fatal Faults

Considering the fact that these apps have undergone extensive user supervision and plenty of testing with prior testing approaches, they are unlikely to expose large-scale faults in general, as we can see in the results of the original DQT and APE. Nevertheless, with the introduction of the changeful and influential network factor, which is actually important but generally neglected by prior testing approaches, many hidden

TABLE III: Evaluation Results Based Upon DQT [17] on Android 14 (R. = RANDOM, N. = NATE)

App			Fault Detection - Level F			Fault Detection - Level E			Code Coverage - INST		
ID	App Name	INST	DQT	DQT+R.	DQT+N.	DQT	DQT+R.	DQT+N.	DQT	DQT+R.	DQT+N. (+DQT%)
A1	Signal	1,641,092	2	3	10	6	10	18	18.28	17.88	20.82 (+13.9%)
A2	Firefox	1,471,339	0	0	1	2	2	5	18.61	18.49	20.20 (+ 8.5%)
A3	WordPress	1,123,645	0	0	2	4	5	9	-	-	-
A4	DuckDuckGo	1,073,164	0	0	1	0	0	3	37.88	38.04	38.89 (+ 2.7%)
A5	Thunderbird(K9Mail)	389,748	1	1	4	3	2	7	31.56	31.52	34.97 (+10.8%)
A6	Wikipedia	355,777	1	1	4	8	12	16	36.44	31.63	36.91 (+ 1.3%)
A7	SuntimesWidget	308,434	0	0	4	1	2	5	33.55	32.32	37.18 (+10.8%)
A8	AnkiDroid	229,803	1	1	2	5	3	9	33.99	34.56	37.16 (+ 9.3%)
A9	NewPipe	164,002	0	0	3	5	13	20	38.93	39.88	46.29 (+18.9%)
A10	AmazeFileManager	150,747	1	1	3	9	10	12	28.56	28.53	30.04 (+ 5.2%)
A11	AntennaPod	131,779	1	1	4	6	8	16	46.69	41.76	51.69 (+10.7%)
A12	ConnectBot	70,774	0	1	2	4	7	8	29.22	30.50	30.58 (+ 4.7%)
Total Fault / Average Coverage			7	9	40 (×5.7)	53	74	128 (×2.4)	32.16	31.37	34.98 (+ 8.8%)

TABLE IV: Evaluation Results Based Upon APE [18] on Android 9 (R. = RANDOM, N. = NATE)

App			Fault Detection - Level F			Fault Detection - Level E			Code Coverage - INST		
ID	App Name	INST	APE	APE+R.	APE+N.	APE	APE+R.	APE+N.	APE	APE+R.	APE+N. (+APE%)
A1	Signal	1,641,092	1	1	5	3	5	10	12.51	11.48	17.94 (+43.4%)
A2	Firefox	1,471,339	0	0	0	3	4	6	21.95	22.99	24.52 (+11.7%)
A3	WordPress	1,123,645	0	0	2	5	10	16	-	-	-
A4	DuckDuckGo	1,073,164	0	0	1	2	3	3	36.95	37.06	39.69 (+ 7.4%)
A5	Thunderbird(K9Mail)	389,748	0	0	0	0	0	1	26.03	22.89	31.70 (+21.8%)
A6	Wikipedia	355,777	3	3	5	18	27	31	43.61	37.49	43.49 (- 0.3%)
A7	SuntimesWidget	308,434	3	1	5	13	11	17	42.93	41.24	46.53 (+ 8.4%)
A8	AnkiDroid	229,803	0	0	2	4	2	9	31.73	31.36	36.45 (+14.9%)
A9	NewPipe	164,002	5	4	10	26	29	36	47.95	43.82	50.47 (+ 5.3%)
A10	AmazeFileManager	150,747	3	4	6	15	14	21	30.02	30.67	34.06 (+13.5%)
A11	AntennaPod	131,779	1	0	4	7	9	17	37.53	24.30	50.98 (+35.8%)
A12	ConnectBot	70,774	1	2	5	13	14	17	30.76	31.12	31.28 (+ 1.7%)
Total Fault / Average Coverage			17	15	45 (×2.6)	109	128	184 (×1.7)	32.91	30.4	37.01 (+12.5%)

network-related faults are exposed, resulting in a significant increase in the number of faults detected.

Code Coverage. The introduction of NATE also brings improvements in code coverage, as presented in the *Code Coverage* Column of Table III and Table IV. Overall, DQT+NATE improves on all 11 apps by an average of 8.8%, while APE+NATE improves by an average of 12.5%.

Based on the findings in our empirical study (refer to Section II), our further investigations reveal that such improvements mainly come from three aspects: ① The triggering of the handling logics for abnormal networks. ② The exploration of the network-related extra app functionalities. ③ The activation of the general testing approaches on handling non-deterministic transitions [17], [18], [31]. Introducing the network factor inevitably brings more non-deterministic transitions, as the same app event may cause different jumps depending on the network status. However, this is not necessarily detrimental to the testing approaches. For DQT, which is based on deep RL with strong robustness, such attempts bring more possibilities and more comprehensive Q-value evaluation [17]. APE, although a model-based approach, does not rely on a static app model, but dynamically reduces, abstracts, and refines the app model at runtime [18]. Benefits from such mechanisms, APE has achieved more refined testing exploration with appropriate non-deterministic transition perturbations, as seen in the improvements on APE+NATE.

We also notice that on app *Wikipedia*, the improvement brought by NATE to DQT is not significant (+1.3%), or even slightly decreases on APE (−0.3%). We find that this is mainly due to the faulty network outage handling page of the tested version. Testing approaches get stuck inside it, with all app events guided to other app functionalities temporarily disabled, where they can only go back or restart the app instead, leading to a negative impact on the overall testing effectiveness.

TABLE V: The Overall Unique Contributions of NATE

App			Fault Detection		Code Coverage	
ID	App Name	INST.	Level F	Level E	INST%	+INST
A1	Signal	1,641,092	**10**	**16**	**6.41%**	105,261
A2	Firefox	1,471,339	1	8	1.57%	23,072
A3	WordPress	1,123,645	3	**17**	-	-
A4	DuckDuckGo	1,073,164	2	3	**5.31%**	56,933
A5	Thunderbird(K9Mail)	389,748	3	5	**4.87%**	18,987
A6	Wikipedia	355,777	4	**20**	1.30%	4,637
A7	SuntimesWidget	308,434	**5**	11	3.32%	10,249
A8	AnkiDroid	229,803	2	8	**4.03%**	9,265
A9	NewPipe	164,002	**8**	**30**	**4.78%**	7,835
A10	AmazeFileManager	150,747	4	**17**	**6.24%**	9,411
A11	AntennaPod	131,779	**7**	**20**	**4.51%**	5,940
A12	ConnectBot	70,774	4	8	0.77%	545
Average			**4.4**	**13.6**	**4.00%**	

Unique Contributions. To further ascertain the unique contri-

TABLE VI: Confirmed Network-Related Real-World Bugs Detected by NATE

ID	App Name	Star	GPD	Error Type	URL	FF	Status	RT	Pattern
#1	Signal	27.6k	100m+	java.net.SocketTimeoutException	https://github.com/signalapp/Signal-Android/issues/14098	✓	Confirmed	2 days	P1, P3
#2	Signal	27.6k	100m+	java.net.ConnectException	https://github.com/signalapp/Signal-Android/issues/14006	✓	Confirmed	2 months	P1, P3
#3	Signal	27.6k	100m+	java.lang.AssertionError	https://github.com/signalapp/Signal-Android/issues/13974	✓	Confirmed	1 week	P1
#4	Firefox	10.0k	100m+	java.net.SocketTimeoutException	https://bugzilla.mozilla.org/show_bug.cgi?id=1957614	✓	Confirmed	2 months	P3
#5	Firefox	10.0k	100m+	java.net.SocketTimeoutException	https://bugzilla.mozilla.org/show_bug.cgi?id=1949963		Fixed	1 day	P1, P3
#6	Firefox	10.0k	100m+	org.mozilla.geckoview.WebRequestError	https://bugzilla.mozilla.org/show_bug.cgi?id=1949676	✓	Confirmed	1 day	P1
#7	DuckDuckGo	4.3k	50m+	java.lang.IllegalStateException	https://github.com/duckduckgo/Android/issues/5925	✓	Confirmed	3 weeks	P3
#8	DuckDuckGo	4.3k	50m+	java.net.SocketTimeoutException	https://github.com/duckduckgo/Android/issues/5831	✓	Confirmed	1 day	P3
#9	WordPress	3.1k	10m+	DiagnosticCoroutineContextException	https://github.com/wordpress-mobile/WordPress-Android/issues/21818	✓	Confirmed	1 day	P1, P3
#10	WordPress	3.1k	10m+	java.lang.NullPointerException	https://github.com/wordpress-mobile/WordPress-Android/issues/21817	✓	Confirmed	1 day	P3
#11	Thunderbird	12.5k	5m+	java.lang.NullPointerException	https://github.com/thunderbird/thunderbird-android/issues/9077	✓	Confirmed	1 day	P1, P3
#12	Wikipedia	2.7k	50m+	java.lang.NullPointerException	https://phabricator.wikimedia.org/T392660	✓	Fixed	1 day	P3
#13	Wikipedia	2.7k	50m+	java.net.SocketTimeoutException	https://phabricator.wikimedia.org/T392472	✓	Fixed	1 day	P3
#14	AnkiDroid	10.0k	10m+	java.lang.IndexOutOfBoundsException	https://github.com/ankidroid/Anki-Android/issues/18234	✓	Fixed	3 days	P1, P3
#15	NewPipe	35.1k		java.lang.NullPointerException	https://github.com/TeamNewPipe/NewPipe/issues/12198	✓	Confirmed	1 day	P2, P3
#16	NewPipe	35.1k		android.app.RemoteServiceException	https://github.com/TeamNewPipe/NewPipe/issues/12197		Confirmed	1 day	P2
#17	NewPipe	35.1k		java.net.SocketTimeoutException	https://github.com/TeamNewPipe/NewPipe/issues/12018		Confirmed	4 days	P2, P3
#18	NewPipe	35.1k		android.os.NetworkOnMainThreadException	https://github.com/TeamNewPipe/NewPipe/issues/11183		Confirmed	2 days	P2
#19	AmazeFileManager	5.8k	1m+	java.net.UnknownHostException	https://github.com/TeamAmaze/AmazeFileManager/issues/4393	✓	Confirmed	1 day	P1
#20	AntennaPod	7.2k	1m+	java.lang.NullPointerException	https://github.com/AntennaPod/AntennaPod/issues/7778		Fixed	1 day	P2
#21	AntennaPod	7.2k	1m+	java.lang.NullPointerException	https://github.com/AntennaPod/AntennaPod/issues/7777	✓	Fixed	1 day	P1

butions from NATE, we merge all the 10-round testing results from DQT+NATE and APE+NATE to construct an overall test set, and then subtract the set formed by the 10-round testing results from the original DQT and APE, as shown in Table V, with those exceeding the average are bold.

As depicted in Table V, NATE uniquely contributes 4.4 Level F Faults, 13.6 Level E Faults, and 4% app code on average at each app. On the largest app *Signal*, NATE uniquely contributes 10 Level F Faults, 16 Level E Faults, and over 6.4% app code. Given the massive scale of *Signal* with over 1.6 million instructions, it is an additional over 105k instructions of unique coverage, which is a non-negligible improvement.

C. RQ2: Ablation Evaluation

To answer RQ2, we implement a purely RANDOM ablation version: The *Network Injector* is retained from NATE, but instead of applying network-aware guidance, it injects network events with probability τ, where τ is set to 0.1 following the random injection parameter convention [23], [31].

The evaluation results are presented under DQT+R. in Table III and Table IV. Comparing NATE to RANDOM, with the help of network-aware guidance: ① DQT is able to detect 4.4 times Level F Faults and 1.7 times Level E Faults, and cover 11.5% more instructions. ② APE detects three times Level F Faults and 1.4 times Level E Faults, and covers 21.7% more instructions. It is reasonable, as we have discussed in Section III-C. The RANDOM strategy not only fails to effectively guide general testing approaches to explore network-related extra app functionalities, but also frequently interferes with the normal exploration of the general testing approach. In the case of *Wikipedia*, frequent and inopportune network event injection causes exploration to continuously get stuck on the faulty network outage handling page, resulting in lower coverage on both approaches. Another typical example is from the podcast manager app *AntennaPod*, where many functionalities are only enabled after an online podcast is added. The add process starts from a specific entry and requires a series of continuous app events to be completed under a normal network, where any failure of any step results in task failure. The RANDOM strategy constantly interrupts the process, leaving many subsequent features hard to explore.

Note that compared to the original DQT and APE, under the enhancement of the RANDOM strategy: ① Competitive fault detection results are still achieved across apps and general testing approaches. ② For code coverage, although an overall decrease is observed, improvements across several apps are still caught, such as *DuckDuckGo* and *ConnectBot*. Such findings demonstrate the importance of the network factor for modern apps, which should be taken seriously in testing.

D. RQ3: Real-World Practice

To answer RQ3, we manually synthesized the important information of the faults (such as the stack traces, the occurrence time point, the execution log of the testing approaches, and the execution screenshot sequence), tried to reproduce the network-related faults detected by NATE, prepared the issue reports, and reported them to the developers.

So far, 21 reported issues have been confirmed as real-world bugs, and six have already been fixed. To reduce false positives, we only consider an issue confirmed with explicit responses from the developer, e.g., direct replies or tag operations. Table VI presents the details of these bugs, along with the app name, GitHub stars (if available), Google Play Downloads (GPD, if available), the error type of the bug, the issue URL, whether first found by NATE or not (FF), current status, the duration between reporting and developers' first responses, and the triggering patterns of the bug.

Overall, we have several interesting findings: ① Among the 21 bugs, 16 were first found and reported by NATE. On the four apps with very large user bases and GPD more than 50 million, NATE uncovered 10 bugs, with nine of them being first found. This demonstrates **the effectiveness of NATE in uncovering faults not easily found, traced, or reproduced by users or developers**. ② Notably, none of the 21 bugs were detected by the original testing approaches. Such a fact demonstrates **the unique ability of NATE compared to SOTA Android testing approaches**. Before NATE, prior approaches such as DQT [17] and APE [18] treated network connectivity as a simple on/off toggle and, despite extensive testing in our evaluation, failed to surface any of such 21 network-related bugs. ③ Interestingly, **users are also sensitive to network-related bugs**. There are five bugs, while also detected by NATE, already been reported by users when we tried to report. At *NewPipe*, a popular streaming app with 35.1k stars on GitHub, users reported multiple network-related bugs shortly after the new version was released, which were also detected by NATE simultaneously. ④ **Developers also expressed great attention to the reported bugs**, with 15/21 confirmed within two days, while 13 were even confirmed within one day. Several bugs were tagged as high priority by the developers and fixed soon. For example, Bug #5 was tagged as Priority *P1*, which is the highest priority among the five-level priority system of *Firefox*, indicating the plan to fix it in the current release cycle. ⑤ From the error types of these bugs, we find that **network-related faults could directly or indirectly lead to a variety of bugs, more than just intuitive ones like temporary connection failures or timeouts**. They should be taken more seriously in practice. ⑥ The last column of Table VI presents the triggering patterns that could successfully reproduce the bug in our attempts, where we found that **many bugs can actually be triggered by more than one pattern**, indicating the high practical reference value of our proposed triggering patterns and the corresponding network events. Moreover, **the root causes of these bugs also align with our study findings discussed in Section II-D**, with nine stemming from insufficient handling of abnormal networks, and 12 arising from faulty network-related extra app functionalities.

V. Threats to Validity

Internal Threats. The main internal threats come from the test oracle applied to fault detection in our empirical evaluation. To alleviate such threats, first, we followed the findings in our empirical study that over 85% of network-related faults are traceable in the stack with exceptions. We excluded *deficient app logic* to prevent introducing an unacceptable number of false positives and more significant threats, since they are entirely subjective. Furthermore, we followed the mature guidance [13] and considered both Level F and Level E faults with both crash and non-crash faults included.

External Threats. The main external threats arise from the representative of the selected general testing approaches and benchmark apps. To mitigate such issues, we chose the best-performing two general Android testing approaches on the recent comprehensive evaluation [13], with one based on recent advanced deep RL and the other on classical model testing. For benchmark apps, we systematically excluded network-unrelated or outdated ones on top of the recent AndroTest24 Benchmark [13] and employed the latest version, resulting in 12 apps across 10 categories with average ELOC over 100k.

VI. Related Work

A. General Android Testing Approaches

By convention [13], [30], [38], existing general Android testing approaches can be divided into five categories. Random Testing [14], [45] applies pseudo-random strategies to generate inputs. As a very classical and widely utilized category, Model-Based Testing [18], [23], [46]–[49] typically builds an app model of the AUT and then employs it for test guidance. APE [18] moves forward, dynamically refining an app model with runtime information rather than relying on a static one. Systematic Testing [16], [50]–[54] utilizes sophisticated techniques such as symbolic execution and evolutionary algorithms to achieve specific objectives. With the advancement of machine learning, many approaches employ Supervised Learning [15], [32], [55]–[57] and Reinforcement Learning [17], [29]–[31], [58], [59]. DQT [17] represents the recent advancement on deep RL, which exploits the testing-knowledge sharing mechanism to enable more effective testing, especially on modern large, complex apps.

B. Targeted Android Testing Approaches

As an indispensable supplement and support for general approaches, many targeted approaches [19]–[22], [60], [61] have been proposed. To deal with the changeful preference, Prefest [21] leverages both static analysis and runtime dynamic exploration to amplify test cases with app configurations. QTypist [61] and InputBlaster [60] focus on generating either valid or unusual text inputs to the text input fields, while ClipboardScope [19] proposes to protect user privacy with better access control on the clipboard.

By contrast, given the growing challenge of network-related faults on modern apps, NATE leverages the superiority of existing SOTA general Android testing approaches on app exploration and focuses on further empowering them for effective network-related fault detection.

VII. Conclusion

In this paper, we conduct the *first* empirical study on 154 real-world network-related bugs, achieving notable findings and implications on tackling such faults. Based on our study, we propose NATE, a network-aware testing enhancer guided by RL-based curiosity-driven exploration, to empower existing general Android testing approaches for network-related fault detection. Experiments demonstrate the effectiveness, efficiency, uniqueness, and compelling practical value of NATE.

Acknowledgment

We thank the anonymous reviewers for their invaluable feedback on this paper. This research is supported by the National Natural Science Foundation of China under Grant Nos. 62402213 and 62372227.

References

[1] Backlinko. (2025) Smartphone usage statistics. [Online]. Available: https://backlinko.com/smartphone-usage-statistics

[2] GitHub. (2025) Newpipe issues #4920. [Online]. Available: https://github.com/TeamNewPipe/NewPipe/issues/4920

[3] ——. (2025) Fenix issues #18040. [Online]. Available: https://github.com/mozilla-mobile/fenix/issues/18040

[4] ——. (2025) Anki-android issues #17551. [Online]. Available: https://github.com/ankidroid/Anki-Android/issues/17551

[5] ——. (2025) Signal-android issues #11202. [Online]. Available: https://github.com/signalapp/Signal-Android/issues/11202

[6] Bugzilla. (2025) Firefox for android bug #1891257. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1891257

[7] GitHub. (2025) Anki-android issues #10000. [Online]. Available: https://github.com/ankidroid/Anki-Android/issues/10000

[8] ——. (2025) Antennapod issues #1942. [Online]. Available: https://github.com/AntennaPod/AntennaPod/issues/1942

[9] ——. (2025) Anki-android issues #12711. [Online]. Available: https://github.com/ankidroid/Anki-Android/issues/12711

[10] ——. (2025) Newpipe issues #8898. [Online]. Available: https://github.com/TeamNewPipe/NewPipe/issues/8898

[11] ——. (2025) Antennapod issues #3196. [Online]. Available: https://github.com/AntennaPod/AntennaPod/issues/3196

[12] StatCounter. (2025) Mobile operating system market share worldwide. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[13] Y. Lan, Y. Lu, M. Pan, and X. Li, "Navigating mobile testing evaluation: A comprehensive statistical analysis of android GUI testing metrics," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 944–956. [Online]. Available: https://doi.org/10.1145/3691620.3695476

[14] Google. (2023) Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[15] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 1070–1073. [Online]. Available: https://doi.org/10.1109/ASE.2019.00104

[16] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 469–480. [Online]. Available: https://doi.org/10.1145/3377811.3380382

[17] Y. Lan, Y. Lu, Z. Li, M. Pan, W. Yang, T. Zhang, and X. Li, "Deeply reinforcing android GUI testing with deep reinforcement learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 71:1–71:13. [Online]. Available: https://doi.org/10.1145/3597503.3623344

[18] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of android applications via model abstraction and refinement," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 269–280. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00042

[19] Y. Chen, R. Tang, C. Zuo, X. Zhang, L. Xue, X. Luo, and Q. Zhao, "Attention! your copied data is under monitoring: A systematic study of clipboard usage in android apps," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 63:1–63:13. [Online]. Available: https://doi.org/10.1145/3597503.3623317

[20] K. Zhao, X. Zhan, L. Yu, S. Zhou, H. Zhou, X. Luo, H. Wang, and Y. Liu, "Demystifying privacy policy of third-party libraries in mobile apps," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1583–1595. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00137

[21] M. Pan, Y. Lu, Y. Pei, T. Zhang, and X. Li, "Preference-wise testing of android apps via test amplification," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 4:1–4:37, 2023. [Online]. Available: https://doi.org/10.1145/3511804

[22] S. Yang, S. Chen, L. Fan, S. Xu, Z. Hui, and S. Huang, "Compatibility issue detection for android apps based on path-sensitive semantic analysis," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 257–269. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00033

[23] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 245–256. [Online]. Available: https://doi.org/10.1145/3106237.3106298

[24] GitHub. (2025) Network-aware testing enhancer. [Online]. Available: https://github.com/SEG-DENSE/NATE

[25] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89

[26] Google. (2025) Connect to the network. [Online]. Available: https://developer.android.com/develop/connectivity/network-ops/connecting

[27] ——. (2025) Manifest.permission. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission#INTERNET

[28] Bugzilla. (2025) Welcome to bugzilla: The issue tracker for firefox and other mozilla products. [Online]. Available: https://bugzilla.mozilla.org/home

[29] D. Ran, H. Wang, W. Wang, and T. Xie, "Badge: Prioritizing UI events with hierarchical multi-armed bandits for automated UI testing," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 894–905. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00083

[30] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 65:1–65:29, 2022. [Online]. Available: https://doi.org/10.1145/3502868

[31] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[32] F. Y. B. Daragh and S. Malek, "Deep GUI: black-box GUI input generation with deep learning," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 905–916. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678778

[33] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 423–435. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00048

[34] R. S. Sutton and A. G. Barto, *Reinforcement Learning - An Introduction (second edition)*, ser. Adaptive Computation and Machine Learning. MIT Press, 2018.

[35] C. J. C. H. Watkins and P. Dayan, "Technical note q-learning," *Mach. Learn.*, vol. 8, pp. 279–292, 1992.

[36] Google. (2025) Android debug bridge (adb). [Online]. Available: https://developer.android.com/studio/command-line/adb

[37] EclEmma. (2025) Jacoco java code coverage library. [Online]. Available: https://www.eclemma.org/jacoco/

[38] F. Behrang and A. Orso, "Seven reasons why: An in-depth study of the limitations of random test input generation for android," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 1066–1077. [Online]. Available: https://doi.org/10.1145/3324884.3416567

[39] Google. (2025) Permissions on android. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[40] ——. (2024) Run apps on the android emulator. [Online]. Available: https://developer.android.com/studio/run/emulator

[41] StatCounter. (2025) Android version market share worldwide. [Online]. Available: https://gs.statcounter.com/android-version-market-share

[42] Google. (2025) Crashes. [Online]. Available: https://developer.android.com/topic/performance/vitals/crash

[43] ——. (2024) Logcat command-line tool. [Online]. Available: https://developer.android.com/studio/command-line/logcat

[44] GitHub. (2025) Wordpress-android/google configuration. [Online]. Available: https://github.com/wordpress-mobile/WordPress-Android?tab=readme-ov-file#google-configuration

[45] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 224–234. [Online]. Available: https://doi.org/10.1145/2491411.2491450

[46] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable automated model-based GUI testing for android enhanced by reinforcement learning," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 135:1–135:5. [Online]. Available: https://doi.org/10.1145/3551349.3559505

[47] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 115–127. [Online]. Available: https://doi.org/10.1109/ASE.2019.00021

[48] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 559–570. [Online]. Available: https://doi.org/10.1145/2884781.2884853

[49] Y. M. Baek and D. Bae, "Automated model-based android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 238–249. [Online]. Available: https://doi.org/10.1145/2970276.2970313

[50] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 481–492. [Online]. Available: https://doi.org/10.1145/3377811.3380402

[51] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 419–429. [Online]. Available: https://doi.org/10.1145/3238147.3238225

[52] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[53] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 461–471. [Online]. Available: https://doi.org/10.1109/ISSRE.2015.7381839

[54] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 599–609. [Online]. Available: https://doi.org/10.1145/2635868.2635896

[55] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable UI tests," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 269–282. [Online]. Available: https://doi.org/10.1145/3236024.3236055

[56] Y. Köroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: qlearning-based exploration of android applications," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 105–115. [Online]. Available: https://doi.org/10.1109/ICST.2018.00020

[57] N. P. B. Jr., M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, C. Julien, G. A. Lewis, and I. Segall, Eds. ACM, 2018, pp. 133–143. [Online]. Available: https://doi.org/10.1145/3197231.3197243

[58] Y. Zhao, B. Harrison, and T. Yu, "Dinodroid: Testing android apps using deep q-networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 122:1–122:24, 2024. [Online]. Available: https://doi.org/10.1145/3652150

[59] C. Degott, N. P. B. Jr., and A. Zeller, "Learning user interface element interactions," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 296–306. [Online]. Available: https://doi.org/10.1145/3293882.3330569

[60] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 137:1–137:12. [Online]. Available: https://doi.org/10.1145/3597503.3639118

[61] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile GUI testing," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1355–1367. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00119