Who Should We Blame for Android App Crashes? An In-Depth Study at Scale and Practical Resolutions

LIANGYI GONG, CNIC, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China
HAO LIN, School of Software, Tsinghua University, Beijing, China
DAIBO LIU, School of Computer Science and Electronic Engineering, Hunan University, Changsha, China
LANQI YANG, University of Chinese Academy of Sciences, Beijing, China
HONGYI WANG, School of Software, Tsinghua University, Beijing, China
JIAXING QIU, School of Software, Tsinghua University, Beijing, China
ZHENHUA LI, School of Software, Tsinghua University, Beijing, China
FENG QIAN, Ming Hsieh Department of ECE, University of Southern California, Los Angeles CA, USA

Android system has been widely deployed in energy-constrained IoT devices for many practical applications, such as smart phone, smart home, healthcare, fitness, and beacons. However, Android users oftentimes suffer from app crashes, which directly disrupt user experience and could lead to data loss. Till now, the community have limited understanding of their prevalence, characteristics, and root causes. In this article, we make an in-depth study of the crash events regarding ten very popular apps of different genres, based on fine-grained system-level traces crowd-sourced from 93 million Android devices. We find that app crashes occur prevalently on the various hardware models studied, and better hardware does not seem to essentially relieve the problem. Most importantly, we unravel multi-fold root causes of app crashes, and pinpoint that the most crashes stem from the subtle yet crucial inconsistency between app developers' supposed memory/process management model and Android's actual implementations. We design practical approaches to addressing the inconsistency; after large-scale deployment, they reduce 40.4% of the app crashes with negligible system overhead. In addition, we summarize important lessons learned from this study, and have released our measurement code/data to the community.

CCS Concepts: • **Networks → Middle boxes / network appliances;**

Additional Key Words and Phrases: Android system, app crash, root cause analysis, memory management, process management

This work is supported by the National Natural Science Foundation of China under grants 62272440, 62332012, 62202266, and 62372166, National Key R&D Program of China under grant 2022YFB4500703.

Authors' addresses: L. Gong, CNIC, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China; e-mail: lygong@cnic.cn; H. Lin, H. Wang, J. Qiu, and Z. Li, School of Software, Tsinghua University, Beijing, China; e-mails: linhaomails@gmail.com, hywangthu@gmail.com, jiaxingqiu2000@gmail.com, lizhenhua1983@gmail.com; D. Liu (Corresponding author), School of Computer Science and Electronic Engineering, Hunan University, Changsha, China; e-mail: dbliu@hnu.edu.cn; L. Yang, University of Chinese Academy of Sciences, Beijing, China; e-mail: yanglanqi0518@gmail.com; F. Qian, Ming Hsieh Department of ECE, University of Southern California, Los Angeles CA, USA; e-mail: fengqian@usc.edu.

ACM 1550-4859/2024/04-ART62
https://doi.org/10.1145/3649895

ACM Reference Format:
Liangyi Gong, Hao Lin, Daibo Liu, Lanqi Yang, Hongyi Wang, Jiaxing Qiu, Zhenhua Li, and Feng Qian. 2024. Who Should We Blame for Android App Crashes? An In-Depth Study at Scale and Practical Resolutions. *ACM Trans. Sensor Netw.* 20, 3, Article 62 (April 2024), 24 pages. https://doi.org/10.1145/3649895

1 INTRODUCTION

A variety of issues could impair the user experience and system reliability of Android-based IoT and IIOT devices [57] during an Android app's execution, including slow rendering [19], quick battery drain [31], app-not-responding [12], system-not-responding [39], app crash, and so forth. To ensure the competitive edge, well-developed testing frameworks [20, 45] and checking tools [28, 49] are available to improve app quality. Despite the developers' striving to deliver high-quality products, many Android users still suffer from crashes of released apps, which are among the most disruptive application reliability issues [35]. In Android, "crash" means that an app unexpectedly exits when an unhandled exception or signal occurs [13], leading to instant service unavailability, function failure, and sometimes even data loss [47]. Moreover, app crashes in the wild are often accompanied by system failures. These failures require dedicated maintenance staff to repair, which would bring prolonged business interruptions and large manpower overhead.

Over the years, a large body of work has been done to detect and understand app crashes. Techniques such as static code scanning [8, 32, 51] and dynamic behavior analysis [20, 30, 44, 45] have been devised to locate potential defects leading to app crashes. Also, considerable in-lab researches have been conducted to uncover the possible reasons behind app crashes [7, 22, 30], but they mainly focus on detecting [21, 43, 46] and fixing app bugs [24]. Despite all efforts of bug detection, fault localization, and fixing techniques, released apps still crash occasionally after being tested thoroughly. Facing this situation, we cannot help but raise a question: who should be blamed for Android app crashes in the wild? In fact, the root causes of app crashes is actually rarely invoked, especially in the wild. Due to a lack of *real-world* measurement and analysis *at scale*, little have we understood regarding the prevalence, characteristics, and root causes of app crashes in the wild. It leaves developers unaware of how to avoid and fix bugs, and significantly hinders practical solutions to the problem.

1.1 Measuring App Crashes on Android Devices at Scale

To close the knowledge gap, we invite the developers of many popular apps to a joint study of app crashes. Finally, the developers of 10 apps[1] (four for video streaming, three for news feed, one for social media, one for e-book reading, and one for photography) accept our invitation. The 10 apps each have 10 M to 50 M daily active users, and we are allowed to instrument them through a lightweight data collector. In view of the influences our study may have on users, our data collector should run without root privileges, do not impact an app's normal behavior (unless a crash happens), and be capable of capturing detailed relevant data at crash scenes, such as CPU utilization, memory usage, call stacks, and heap information. Existing tools (e.g., Android Vitals [14], BreakPad [27]), however, fail to satisfy the above requirements simultaneously. We therefore develop CrashCapturer, a user-space tool for capturing app crash events in a timely and comprehensive manner. When implementing CrashCapturer, we find the key challenge is how to preserve the crash data, as the secondary crashes of applications occur during the saving process of crash logs, thus damages the crash scenes, causing the loss of critical information. To effectively

[1]For commercial reasons, the actual names of the ten apps are not revealed at the moment.

catch the ephemeral crash scenes, CrashCapturer preemptively intercepts crash signals and clone processes to reserve the complete crash scenes.

After integrating CrashCapturer into the 10 collaborative apps, we collect their crash data for 5 months under informed user consent (any user can choose to opt out of the study) and a well-established IRB. A total of 123 M app crashes are captured, involving 93 M Android devices and 40 major hardware models. Our measurement reveals that app crashes occur prevalently (ranging from 45% to 75%, averaging at 62%) on various hardware models studied. On average, 1.32 crashes occur on an Android device during the measurement, and on some hardware models the occurrence number per phone even reaches 13.2. If this is not stunning enough, note that the above situation involves only 10 apps; given that an Android phone typically has tens or even hundreds of popular apps installed, it will not be surprising that ≥10 crashes occur on an average phone.

We do not observe essential correlation between the occurrence of app crashes and hardware configurations, indicating that app crashes should be more of a software issue. As Android evolves from version 4.0 to 6.0 where many apparent problems have been resolved, there is an evident decline in the occurrence of app crashes; in contrast, as Android evolves from version 7.0 to 11.0, the occurrence number per phone remains stable at ∼1.2. Besides, we find that video streaming apps are remarkably more prone to crashes, probably due to their high requirements of system resources.

1.2 Understanding Root Causes of App Crashes in the Wild

To uncover the root causes of Android app crashes, we build an automatic pipeline to process the large-scale logs recorded by CrashCapturer from the measured phones. In detail, we first use a natural language processing tool to remove unrelated information (e.g., meaningless empty words and numbers) and extract keyword vectors of crash events. After that, we apply the **locality-sensitive hashing** (**LSH**) algorithm [34] to quickly hash similar crash events into the same "buckets" based on the keyword vectors. Then, crash events of one bucket are classified into the corresponding root-cause clusters using *similar-stack analysis* [22, 39], and we manually inspect the root cause of unbiased crash samples in each dominant cluster. The correctness of our analysis is validated using a different set of unbiased samples.

Eventually, we discover and summarize four major types of root causes of Android app crashes synoptically: (1) failure to access defined variables (60.2%), (2) out of available system resources (26.2%), (3) illegal operations (4.8%), and (4) compatibility errors (8.7%). *Failure to access defined variables* means that resource handles (e.g., database, socket, file, UI, and object memory) are null, and the running program fails to check the availability. We find that such perplexing problem mainly arises on the subtle yet crucial inconsistency between app's internal process management model and Android's actual implementation management. Moreover, *out of available system resources* primarily result from the exhaustion of an app's allocated memory, thread or file handles, and so on. In the article, we give a deep insight of serious crashes related to out of memory errors, which aroused by memory leak and bitmap problem (23.1%). Because of the improved app performance, for fairness reasons, any app cannot request too many system resources, the second root causes can hardly be eliminated in fact. Then, we find that crashes, attributed to the third root causes, are mainly due to illegal operations, such as invalid input parameters, changes of definition and wrong data type conversions. However, *illegal operations* hidden deeply are undetectable and hard to be found in interactivity testing. To accomplish app testings before released, full coverage measurements of code in offline are required in future. *Compatibility errors* result in apps to be unable to implement API calls on the system SDK running a lower version. The situation of the

incompatibility can be improved from the developers' perspective. Considering the diversity of Android system versions in the market, developers should conduct more comprehensive offline testings and active online crash detecting on different versions.

To be specific, we pinpoint that the most crashes (36.8%) mainly stem from the subtle yet crucial inconsistency between app's internal process management model and Android's actual implementation management. In the Android system, once an app is switched to the background, it may be killed by the low memory kill mechanism of Android system, and cannot receive message notifications in a timely manner. Therefore, these popular apps employ an anti-system technology to prevent themselves from being killed by Android process management. Specifically speaking, once a key process of an app has received the kill signal from Android process, its daemon immediately generates a mirror process with a different *uid* number to escape from being killed. Because the process generation is a heavyweight operation, it cannot ensure that every process of an app would be revived in time. Crashes would occur if some processes referenced by other processes is inescapably killed by Android process management, resulting in a null pointer exception. In fact, such crashes are difficult to completely reproduce in laboratory conditions, due to the complex factors that affect the reactivation efficiency of process in real-world environments. We think that Android app developers seem to be blamed for app crashes caused by the inconsistency due to violating Android process management mechanisms, while Android OSes developers should also be responsible for app crashes owing to overly strict process management mechanisms. If popular apps are given certain message notification channels, many app crashes may be avoided.

1.3 Mitigating App Crashes with Lightweight Countermeasures

From what have been measured and analyzed in Section 3.2, we dissect our findings in depth, present a more representative approach and practical guiding advice from both developers and carriers perspectives. For developers of applications, we design a lightweight approach for app developers to conveniently fix the issue, in order to maintain a lifecycle state machine for objects to identify the subtle termination differences between processes and activities. Moreover, for carriers, we declare that in Android, insufficient offline test and improper models are responsible for poor stability and robustness, therefore, comprehensive offline test before released is essential.

Worth mentioning the fact that competition between processes is the major reason for app crashes at runtime. After updating the 10 collaborative apps with these solutions and rolling them out at scale, almost all (>98%) the targeted crashes are avoided on the Android phones of the 25 M opt-in users, reducing the total number of crashes by 40.4% with negligible system overhead.

1.4 Summary of Contributions

— We conduct the fist large-scale and in-depth measurement study of app crashes on Android devices in the wild, and confirm their prevalence for various hardware models. We also discover that Android app crashes in the wild are more of a software issue than a hardware issue.

— We present our end-to-end data collection and analysis pipeline for deeply understanding app crashes. Our collection is lightweight and has little impact on the performance of Android systems. Our analysis pipeline can automatically diagnose the root causes of app crashes.

— We provide insightful guidance for eliminating various app crashes and practically address the largest number of app crashes by the process management model inconsistency problem. After real-world deployment, our solution reduces 40.4% of the app crashes with negligible system overhead.

Our measurement code and data are released in part at https://AndroidCrash.github.io to benefit the community.

2 STUDY METHODOLOGY

In this section, we first present the CrashCapturer tool that records required in-situ data of a relevant app at crash scenes, and then describe our automatic analysis pipeline to pinpoint the root causes of the collected crashes.

2.1 CrashCapturer

In order to study app crashes thoroughly, we construct a large-scale and in-depth measurement study online, where plenty of crash information can be selected. Moreover, to find out root reasons of app crashes, numerous fine-grained system-level information is required from crash scenes. The undamaged crash scenes help us analyze the deep systemic causes behind the collapses. However, in view of the influences may have on users when collecting a mass of crash information, three critical requirements, including running without root privileges, not affecting the normal behavior of apps (unless a crash happens) and not causing significant system overhead (computation and network traffic overhead) are accommodated. A large-scale and fine-grained data collection in the wild should avoid interference with the normal utilization of phones.

We first analyze the exception handling mechanisms of Android system, as shown in Figure 1. In Android system, one process of an app is forked from the `Zygote` process in the native layer. Then its Java runtime process is started in the main method of `ZygnoteInit` class. Moreover, the Java exception handlers are set in the `commonInit` of `RuntimeInit` class (①). Here, developers can set default exception handlers to catch the exceptions of threads (②). In practice, the exception handling is implemented in the `uncaughtExceptionHandler` of `LoggingHandler` and `KillApplicationHandler` (③). Once a crash occurs in the process, handlers can record the systematic information (such as memory usage, process properties, OS information, and app information) in the `LoggingHandler` and report the crash information (such as Java file, classes, and stack information, etc) in the `KillApplicationHandler`. Furthermore, developers can also set default exception handlers in the thread (④). If a crash occurs in the thread, the default exception handlers are searched for the exception handling (⑤,⑥). When the default exception handlers are not found, the thread would call the default exception handlers in the app to handle the crash (⑦). If there is not default exception handler, a system error is reported.

Then, we investigate existing tools in the market for app crash capturing and analysis. From what we know, Android Vitals [14] plays a crucial part in collecting stack traces generated by Android apps for *transparent* crash event capturing (i.e., not affecting the app's normal behaviors when no crash happens). When implemented in the real world, it becomes inconvenient somewhere. Routinely, Android Vitals remains off by default, users need to be claimed to turn it on to allow data collection. In fact, considering privacy leakage and traffic cost, users are prone to reject turning on this function. Additionally, we find that some custom Android systems choose to shut off services related to Android Vitals. In addition, another tool called BreakPad [27] is used to analyze the registration and memory information of crashes. However, its post-automation processing is quite time-consuming, and cross-platform properties lead to a large and complex code structure, making development and maintenance more difficult.

The issues above lead to incomplete crash information collection, and significantly hinder our detailed and in-depth analysis of app crashes. Additionally, the overhead problem of crash collector maintenance runs counter to our original intent. Therefore, we build CrashCapturer, a crash data collection tool that operates under three collection requirements above, it has low system overhead and no privacy leakage risk. Otherwise, it can be easily integrated into a concerned app (it only

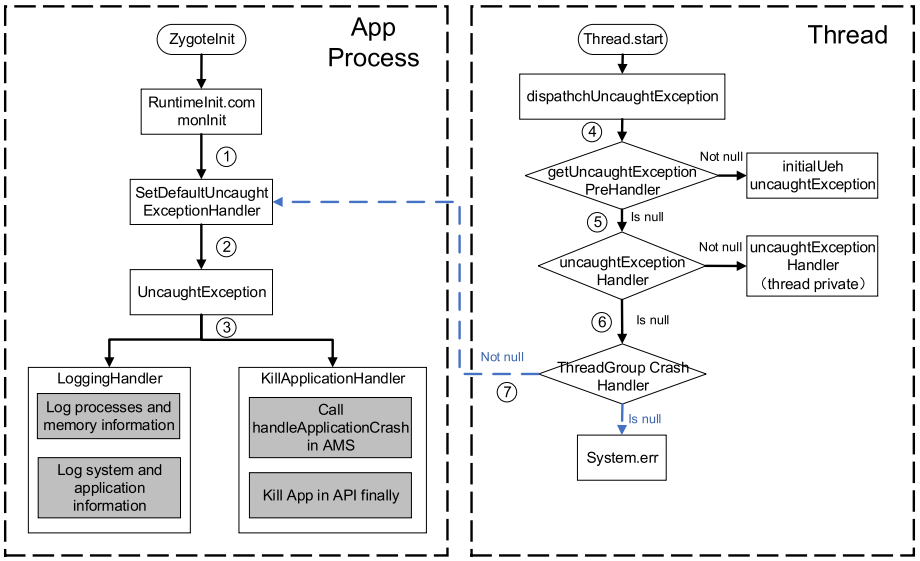

Fig. 1. Exception handling mechanisms of Android system.

requires the app to contain and initialize it at startup time); it does not work (and thus does not change the app's runtime logic) when there is no crash, but has to record the related data in a timely and comprehensive manner once a crash event occurs. In its implementation, however, we discover the major challenge to be the app's secondary crashes upon its first crash [54]. After an app crashes, the Android system will forcibly relaunch the application based on the stack information. However, this forced restart operation of the app can result in a secondary crash due to data loss in the first crash. The app secondary crash then leads to the loss of critical in-situ information such as call stacks and register values, causing all crash logs fail to write.

To address this, we put forward a clone procedure of CrashCapturer so that the crash scene can be reserved entirely as far as possible when crashing, and do not distract regular the process of system recycling at the same time. According to the exception handling mechanisms of Android system as shown in Figure 2, first of all, CrashCapturer rewrites the default crash handling function (②) so as to intercept the associated error signals prior to Android's default crash handling of a concerned app. Then, when a crash occurs in the app (④), the handler (③) preserves the crash scene by suspending all the threads (⑤) of the app's processes with `ptrace` [33]. After that, cloning the crash scene effectively and comprehensively is required, and thus related information will be collected from the cloning process (⑥). Hence, for a crash because of exhausted resources, it is inclined to crash once more due to lack of resources to write logs when the system collects data, resulting in a loss of log data. Therefore, we adopt system resource reservation (①) to save critical crash information in CrashCapturer and retain valid logs (⑦). In practise, reservated system resources include file and socket handles, some memory for retainging data. After that, writing data (i.e., CPU/memory usage, call stacks, heap information, stack frames, and register values) along with purified crash logs uses a regular filter to delete irrelevant information. After irrelevant information such as unnecessary virtual locations, line number, and so on, is removed, there would be an impact on memory reducing and traffic overhead decreasing.

As mentioned in Section 1, the 10 collaborative apps integrated CrashCapturer and released the updated versions to their users in Oct. 2020. Then in the following 5 months (Nov. 2020–March. 2021), we collected their crash data under informed user consent and a well-established IRB. On

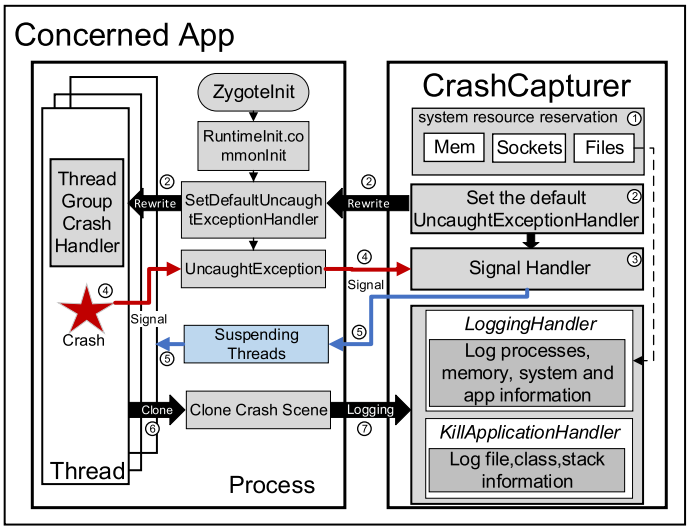

Fig. 2. Workflow of our developed CrashCapturer.

ethical grounds, no personally identifiable information was collected during the process and users could choose to opt out of the study at any time (such users turned out to be very few). Our collection involves lightweight data logging at the crash time; for even a low-end Android device, CrashCapturer incurs <1% CPU utilization, <10 KB memory usage, and <500 KB storage space. The recorded data were uploaded to our back-end server when WiFi is available.

2.2 Root Cause Analysis Pipeline

To unravel the root causes of app crashes, application or system developers usually analyze crash logs artificially. While such manual analysis with low efficiency and poor scalability is unrealistic to apply alone due to the large-scale data set. According to the observation, crash events of the same root cause probably bear resemblances to each other in their exception types, error messages, call stack patterns and other system-level symptoms (e.g., CPU utilization and memory usage) in logs. Therefore, we develop an automated analysis pipeline to improve the efficiency and effectiveness.

In detail, each crash log consists of four parts: exception type, error message, stack trace information and system-level symptoms as shown in Figure 3. The exception type refers to a crash-related exception as officially defined by Java APIs, and the exception plays a guiding role in explaining certain crash situations at a coarse-grained level. The error message is delivered from its error log of Android logging framework. It presents the error reasons and locations as a brief description of developers. Thus, the error message is constructively helpful to our analysis of crash causes. The stack trace information includes the call functions in program stack when crashing. Note that the top function in stack is generally the location where is more inclined to crash, nevertheless, this may not be the critical root cause. The system-level symptoms assist to feedback hardware resource states of system (e.g., CPU utilization, memory usage, and network environment). As a consequence, we need to build our analyzing infrastructure with four parts mentioned above, instead of one certain part, to give a comprehensive study of crash root reasons.

When applying automated analysis pipeline, we find that traditional similar-stack analysis algorithms only value the stack trace information. Hence, root causes are usually excessively specific,

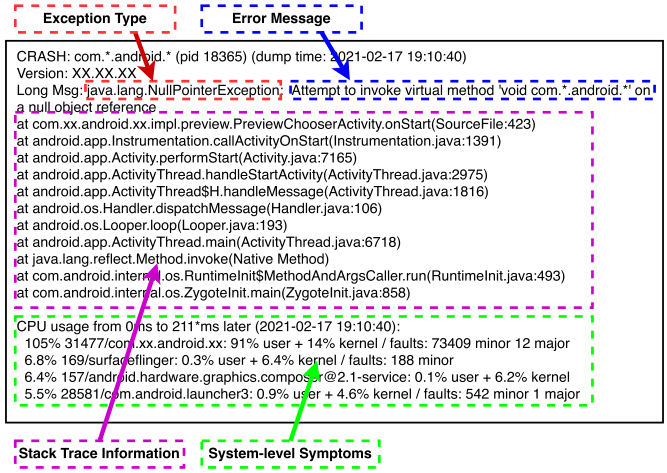

Fig. 3. Examples of our crash logs with four parts.

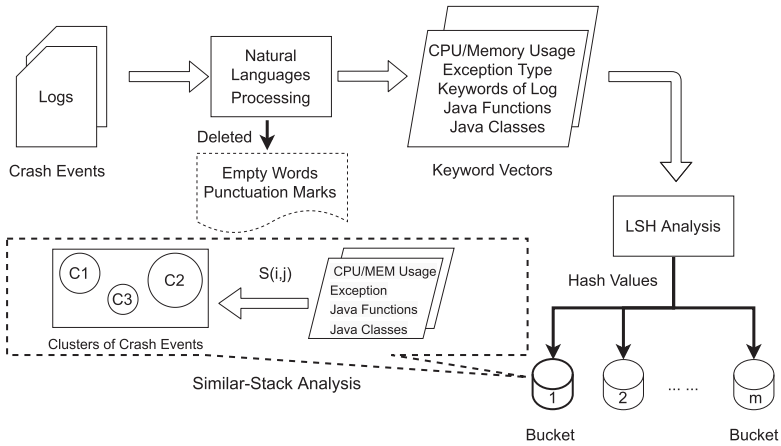

Fig. 4. Workflow of our developed pipeline for automatically analyzing the root causes of crash events.

which hinders researchers to have an insight into systematic defects. Moreover, the complexity of traditional similar-stack analysis is $log(n^2)$ at least, where n is the crash log number. Such massive calculations may take amount of time to analyze the scale of comparative crash log similarities. To deal with these, we propose a novel pipeline for automatically analyzing crash events, as shown in Figure 4. Our pipeline can significantly reduce the overhead of log matching at scale. Briefly, we first construct natural language processing technology to extract keyword vectors of crash logs. Next, locality-sensitive hashing algorithm is applied to distill crash logs into numbered buckets in $log(n)$ time. Finally, we propose an improved similar-stack analysis to form root-cause clusters of crash events by taking system-level symptoms, exception types and crash stack traces into account.

There are a good deal of empty words, punctuation marks, names of file path, and plentiful information related to crashes in error messages. However, irrelevant components play a negative role in analysis, not only cause a significant increase in computational effort, but also bias the results. To raise the efficiency of log classification, we apply natural language processing technology to

remove meaningless empty words and symbols (e.g., punctuation marks) in error messages. After removing, critical and tangible words are kept to make up keyword vectors. A typical keyword vector mainly contains CPU usage, memory consumption, the instruction set, and the app fatal signal, the app failure code, and so on. In the article, our pipeline analysis applies spaCy tool to separate words in error messages to further extract keywords of error messages. We take 10 K error messages at random to conduct experiments, the result indicates that the average string length of needed matching can be reduced by 23% using spaCy.

Despite these efforts, when dealing with large-scale log data, traditional linear search and matching methods still require a significant amount of time and computational resources. To further enhance data analysis efficiency, we employ an LSH algorithm with a complexity of $log(n)$. This algorithm directly processes keyword vectors, including exception types, keywords of the error messages, and function names of the stack traces, and then computes hash values for these keyword vectors. Subsequently, we categorize crash events into multiple buckets based on these hash values. It is significant to set the number of buckets to be 10 times of exception types [37]. If there are too many buckets, it results in overly fine-grained categorization, making it challenging to identify commonalities. Conversely, too few buckets lead to many crash events in a single bucket, which is not ideal for further analysis and makes it difficult to find the root cause. In practice, the number of buckets is set to 620 in our experiments.

After determining the number of buckets, we execute the processing flow based on the following fact: after LSH processing, logically adjacent (similar) crash events have a high probability of being placed in the same bucket, while non-adjacent events are unlikely to be assigned to the same bucket. Therefore, each bucket contains similar types of crash events, which is significant for further elaboration and analysis.

Based on the above processing, we are able to conduct further classification to the crash events in the same bucket into the corresponding root-cause cluster based on the improved similar-stack analysis. To improve classification accuracy and reduce the time complexity of high-dimensional feature classification, we employ a customized similarity calculation method for fine-grained analysis. Firstly, considering the impact of word order on similarity measurement, especially for long texts, we categorize a keyword vector into short-text and long-text types. Short-text features include keywords related to exception types, error messages, and system-level symptoms, while long-text features include Java functions within stack traces, path file names, and so on. Subsequently, we combine features belonging to different types into their respective text vectors, denoted as V_p and V_c. Next, we calculate the similarity between pairs of V_p, denoted as S_p, and pairs of V_c, denoted as S_c. Finally, to obtain an overall similarity measure, we employ a specific method to combine S_p and S_c, resulting in S.

We will now describe the calculation methods for S_p and S_c. The Jaccard Index is commonly used for comparing the similarity between finite sample sets. However, we have noticed that when certain content in the keywords is excessively repetitive, the Jaccard Index tends to yield high similarity scores because it disregards differences in text lengths. To mitigate this issue and prevent overly high similarity scores, we calculate short-text type similarity S_p by using an improved Jaccard Index. In this approach, we include a penalty for the difference in text lengths in the denominator.

$$S_p(i,j) = \frac{|V_{p,i} \cap V_{p,j}|}{|V_{p,i} \cup V_{p,j}| + \alpha * max(|V_{p,i} - V_{p,j}|, |V_{p,j} - V_{p,i}|)}, \tag{1}$$

where α is the penalty coefficient, $\alpha \in (0,1)$. The larger the α value is, the more seriously the result is affected by the text length, while the similarity of text semantics is underestimated, and

vice versa. In practice, the α is adjusted appropriately according to the experimental situation and set to 0.35 in our experiments.

As for long-text similarity S_c, we additionally consider the influence of word order. Term vector space model and Minimum Edit Distance were used to measure similarity.

$$S_c(i,j) = 1 - \frac{ED_c(i,j)}{max(V_{c,i}, V_{c,j})}, \tag{2}$$

$$ED_c(i,j) = \begin{cases} max(V_{c,i}, V_{c,j}) \quad if\ min(V_{c,i}, V_{c,j}) = 0, \\ min = \begin{cases} ED_c(i-1,j)+1 \\ ED_c(i,j-1)+1 \\ ED_c(i-1,j-1), \quad (V_{c,i} = V_{c,j}) \\ ED_c(i-1,j-1)+1, \quad (V_{c,i} \neq V_{c,j}) \end{cases} \end{cases}. \tag{3}$$

The final similarity $S(i,j)$ is derived as the weighted average between $S_p(i,j)$ and $S_c(i,j)$, where the weights are the respective cardinalities of the set V_p and V_c:

$$S(i,j) = \frac{(\sum_{n=i,j} |V_{p,n}|) \cdot S_p(i,j) + (\sum_{n=i,j} |V_{c,n}|) \cdot S_c(i,j)}{\sum_{m=p,c} \sum_{n=i,j} |V_{m,n}|}, \tag{4}$$

V_i and V_j will be classified into the same root-cause cluster if $S(i,j)$ is above a threshold, which is empirically set to 0.95 based on our manual inspection of representative stack trace examples.

In practice, the similar-stack analysis is capable to output hundreds of root cause clusters. Nevertheless, based on our observation, several major clusters with much larger sizes account for the most of crashes. To validate the correctness of our automatic analysis, for each of these major clusters, we manually analyze the K (empirically taken as 100) samples nearest to the cluster centroid, and the K furthest samples. The cross-reference analysis results indicate that our manual analyzed samples have all been correctly classified by the automatic analysis. This is mostly ascribed to the high similarity threshold (0.95) we set.

3 MEASUREMENT RESULTS

Based on CrashCapturer and the generous help from the developers of 10 collaborative apps, a total of 123,090,649 crash events are captured, involving 93,410,716 Android devices across 40 major hardware models as listed in Table 1. With such a large volume of data set, we have multi-fold findings on app crashes in terms of their prevalence and characteristics, as well as in-depth understandings of their root causes. In the paragraphs to follow, all the statistics related to app crashes are only of the 10 apps, and are unrelated to other apps installed in the opt-in users' phones unless otherwise specified.

3.1 Prevalence and Characteristics of App Crashes

Prevalence of App Crashes. Our study reveals that crashes occur prevalently on all the studied phone models. In the article, we utilize the prevalence and frequency to describe the app crash situation. Assuming the number of the hardware model M is m, we count the number of phones that experience at least one crash as n, and the crash number of the ith phone as c_i. Then the prevalence can be defined as $Prevalence = \frac{n}{m}$, $Prevalence \in [0,1]$ and the frequency is define as $Frequency = \frac{\sum_{i=1}^{n} c_i}{m}$. A larger prevalence value means a higher probability of app crashes on the hardware model M, and a larger frequency value indicates that the app crash may cause a serious impact on the user experience. As shown in Table 1, the prevalence ranges from 0.45 to 0.75 and averages at 0.62 during our 5-month study, implying that the majority of Android devices have ever experienced crash events with regard to the 10 collaborative apps. Moreover, we find that an average of 1.32 crash events occur to an Android device during the measurement. As illustrated in

Table 1. Statistics of App Crashes Across our Studied 40 Major Phone Models[2](Ordered from low-end to High-end)

Model	CPU	Memory	Storage	Android	%Usr	Prevalence	Frequency
1	1.2 GHz	2 GB	16 GB	4.0	0.002	0.70	2.22
2	1.2 GHz	2 GB	16 GB	5.0	0.002	0.71	4.27
3	1.8 GHz	2 GB	16 GB	5.0	0.001	0.75	13.21
4	1.95 GHz	2 GB	16 GB	4.0	0.002	0.68	3.53
5	2 GHz	2 GB	16 GB	5.0	0.0005	0.60	2.23
6	2 GHz	3 GB	32 GB	4.0	0.003	0.68	2.04
7	2 GHz	3 GB	32 GB	5.0	0.02	0.70	2.07
8	2 GHz	3 GB	32 GB	6.0	0.03	0.64	2.67
9	2 GHz	3 GB	64 GB	6.0	0.03	0.62	3.39
10	2.2 GHz	3 GB	32 GB	6.0	0.03	0.71	3.38
11	2.2 GHz	3 GB	64 GB	7.0	0.11	0.72	2.74
12	2.2 GHz	4 GB	32 GB	6.0	0.11	0.69	1.26
13	1.8 GHz	4 GB	64 GB	7.0	0.21	0.69	2.85
14	2.0 GHz	4 GB	64 GB	7.0	1.57	0.68	1.41
15	2.05 GHz	6 GB	64 GB	7.0	1.51	0.51	1.43
16	2.2 GHz	4 GB	64 GB	8.0	2.65	0.63	2.30
17	2.2 GHz	4 GB	128 GB	8.0	3.16	0.64	1.06
18	2.2 GHz	4 GB	256 GB	9.0	5.08	0.65	1.20
19	2.2 GHz	6 GB	64 GB	6.0	3.40	0.63	1.78
20	2.2 GHz	6 GB	64 GB	7.0	6.85	0.70	1.61
21	2.2 GHz	6 GB	64 GB	8.0	5.60	0.57	0.94
22	2.2 GHz	6 GB	64 GB	9.0	5.68	0.63	1.51
23	2.2 GHz	6 GB	128 GB	9.0	5.56	0.61	1.47
24	2.2 GHz	6 GB	256 GB	9.0	1.78	0.58	1.50
25	2.4 GHz	6 GB	64 GB	8.0	0.98	0.58	1.55
26	2.4 GHz	6 GB	128 GB	10.0	6.40	0.65	1.12
27	2.45 GHz	6 GB	64 GB	9.0	8.04	0.68	1.26
28	2.45 GHz	6 GB	128 GB	10.0	7.60	0.57	1.21
29	2.58 GHz	6 GB	128 GB	10.0	5.91	0.63	1.24
30	2.58 GHz	8 GB	256 GB	10.0	4.20	0.60	0.96
31	2.8 GHz	6 GB	128 GB	10.0	1.19	0.55	1.21
32	2.8 GHz	6 GB	256 GB	10.0	2.94	0.57	1.20
33	2.84 GHz	6 GB	64 GB	10.0	3.59	0.63	1.82
34	2.84 GHz	6 GB	128 GB	10.0	3.24	0.54	1.09
35	2.84 GHz	6 GB	256 GB	10.0	0.86	0.54	1.27
36	2.84 GHz	8 GB	64 GB	10.0	5.39	0.57	1.13
37	2.84 GHz	8 GB	128 GB	10.0	3.82	0.59	1.13
38	2.84 GHz	8 GB	256 GB	11.0	0.32	0.51	1.70
39	3.13 GHz	8 GB	256 GB	11.0	1.98	0.60	1.06
40	3.13 GHz	8 GB	512 GB	11.0	0.15	0.45	0.95

The three rightmost columns are the percentage of users, the fraction of phones that experience at least one crash (Prevalence), and the average number of crashes occurring per phone (Frequency) during our five-month study.

[2]The total number of specific phone models in our data set is in fact larger than 40. However, to make Table 1 tidy, we have merged models with similar hardware into a major model, leading to a total of 40 major phone models.

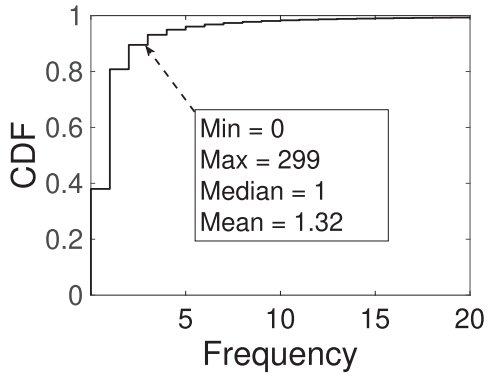

Fig. 5. Number of app crash events happening to a single phone.

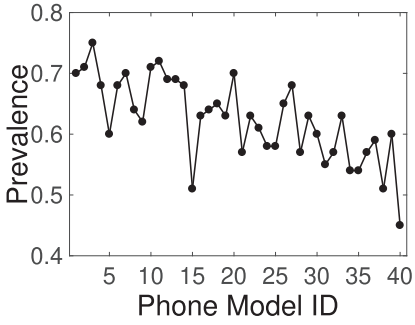

Fig. 6. Prevalence of app crashes on each model of phones.

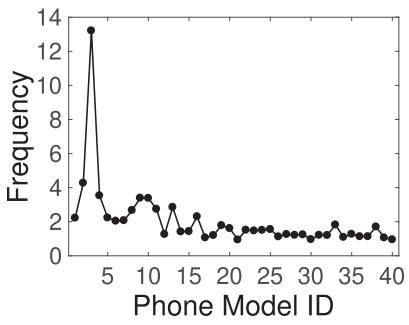

Fig. 7. Frequency of app crashes on each model of phones.

Figure 5, the occurrence number (or *frequency*) of crashes on a device is quite skewed—38% of the devices do not experience app crashes while up to 299 crashes are captured in a single device.

It is worth noting that the above situation involves only the 10 collaborative apps studied; on average, a studied phone has 5.89 of the 10 apps installed. Notably, the 10 apps are developed by professional developers and have gone through rigid tests before their releases. Since a typical Android phone usually has tens or even hundreds of apps installed [6], the actual number of crashes occurring on an average phone in the wild may well exceed 10, indicating that app crashes have in fact brought about much more severe disruptions to mobile user experience than what is stated above.

Hardware Configuration. Intuitively, better hardware configurations might help mitigate app crashes as they usually come with more sufficient system resources (e.g., more powerful CPUs and larger memory). At first glance, as shown in Figure 6 and Figure 7, newer phone models appear to bear fewer app crashes. However, as indicated in Figure 8 and Figure 9, when equipped with the same Android version (i.e., under a fair comparison), better hardware configurations do not necessarily result in fewer app crashes. This phenomenon is especially clear with respect to the 11 phone models that all use Android 10, as illustrated in Figure 10 and Figure 11. Hence, we can conclude that app crashes should be more of a software problem than a hardware problem.

We observe in Figure 7 that a particular phone model (Model 3) has experienced significantly more crashes (as many as 13.2 per phone) than the other models. Delving deep, the exceptionally

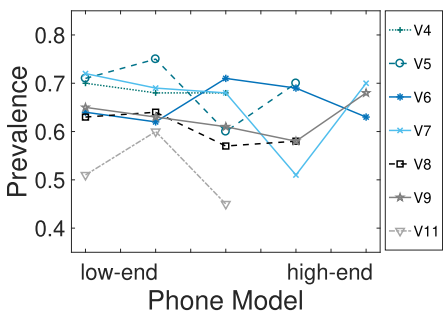

Fig. 8. Prevalence of app crashes on different phone models with different Android versions.

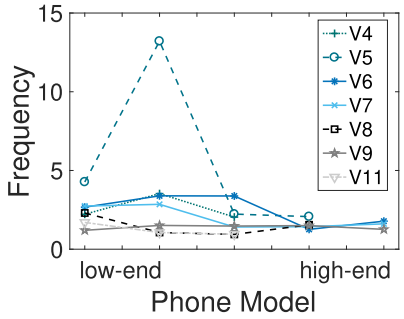

Fig. 9. Frequency of app crashes on different phone models with different Android versions.

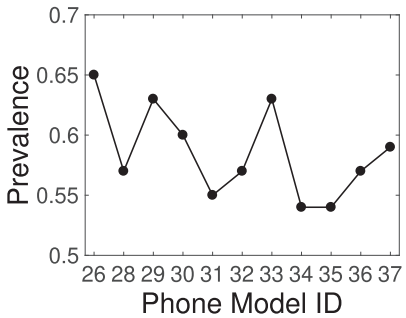

Fig. 10. Prevalence of app crashes on different phone models with Android 10.

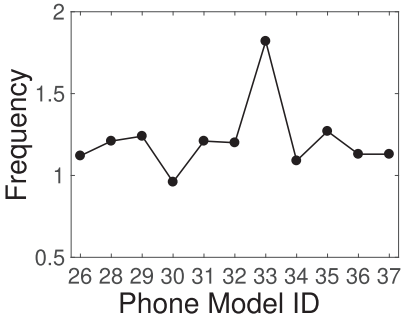

Fig. 11. Frequency of app crashes on different models with Android 10.

frequent crashes on this phone model are mainly attributed to its vendor's problematic modifications made to its Android system—the vendor poses an unreasonable constraint on the maximum number of broadcast receivers per process (which is 500 in its modified Android yet no such constraint exists in vanilla Android).

Android Version. The Android versions involved in our measurement range from 4.0 to 11.0. As the OS constantly evolves, while the prevalence of app crashes is relatively stable around 0.6 (see Figure 12), there is an evident decline regarding the frequency of app crashes (see Figure 13). Specifically, the frequency of app crashes is relatively high for Android 4.0 and 5.0 (i.e., 2.4 and 2.9, respectively), but sharply decreases to 1.7 for Android 6.0 due to the fact that many apparent crash-related issues were fixed then, such as bugs in the graphics rendering component [15]. In comparison, as Android upgrades from 7.0 to 11.0, we observe that although the situation continuously improves, the improvement is getting less and less. This indicates that some bottleneck problems remain unresolved in recent five years, thus hindering the further reduction of app crashes.

Mobile App. Our studied 10 apps belong to five categories: video streaming, news feed, social media, e-book reading, and photography. As illustrated in Figure 14 and Figure 15, these apps are subject to different prevalence and frequencies of crashes. Our major finding is that the video streaming apps have each experienced more crashes per phone than the other categories of apps, probably due to their high consumption of system resources. Overall, the above results indicate that crashes are common to various categories of apps, and those "heavyweight" apps are more subject to crashes.

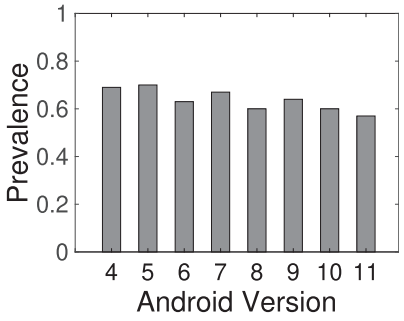

Fig. 12. Prevalence of app crashes for Android versions.

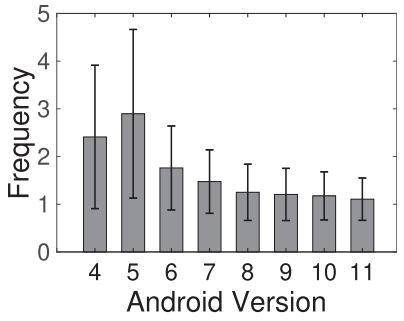

Fig. 13. Frequency of app crashes for Android versions.

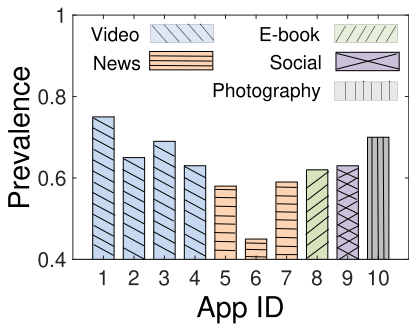

Fig. 14. Prevalence of app crashes for the five kinds of apps.

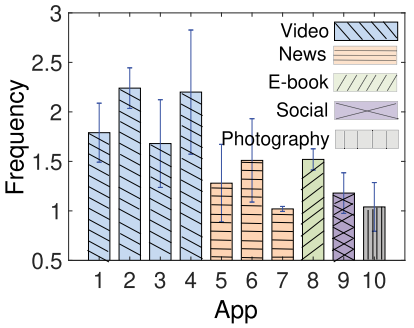

Fig. 15. Frequency of app crashes for the five kinds of apps.

3.2 Root Cause Analysis Results

We acquire 1,287 root cause clusters after leveraging the automatic pipeline detailed in Section 2.2, where 123,090,649 crash logs are collected in total. For overall considered, we take the definition of exception types, frequent keywords in error messages, and top functions of stacks from debug reports into consideration, and perform some apparent analysis. From what is clearly illustrated in Figures 16, `NullPointerException` exceeds 45% of overall crashes and OOM exception is about 20%, while the share of all other exceptions is <5% on average. In addition, after counting the high frequency keywords and the top functions in stacks of error messages, we receive that the most frequent terms in the error messages is "null object reference", and the highest rank of top function in stack is the `stop` function(). Then we look over the source code given by cooperative partner to summarize comprehensively about root causes.

After manual static analysis to the results, we note that all crashes can be summarized to several dominant clusters, which can be summarized as four major dominant clusters account for almost all crashes (>99.9%). Accordingly, the four major root causes are 1) failure to access defined variables (60.2%), 2) out of system resources (26.2%), 3) illegal operations (4.8%), and 4) compatibility errors (8.7%). The remaining "long tail" [58] clusters each takes up only a minor portion (almost always< 0.1%). Most of them are owing to app-specific defects thus not be involved in the analysis. Below we closely examine the logs corresponding to the four major root causes to obtain in-depth understandings.

Failure to Access Defined Variables: We find that the first root cause mainly stems from the failure to access defined logic variables and methods, and resource handles (e.g., database, socket,

file, and UI viewer). Literally interpreting, when a program attempts to operate a defined variable, the variable cannot be read or written, such a situation is the cause of a null point exception. The exception types from crash logs mainly include NullPointerException, IllegalAccessException, IllegalMonitorStateException, IllegalAccessError, and so on. The null pointer exception is persistent and tricky to be solved in the software engineering field, and it is being handled with extra care by senior developers [40]. There has been a great deal of work focusing on the detection of the null pointer exception in engineering codes [5], such as white-box testing and dynamic taint analysis. Even though many efforts have been put into this problem, we find the highest percentage of exceptions still for the null pointer exception in all crashes collected. As the 10 apps we cooperate with are the most popular ones in the application store and tested in depth, we cannot help but think about whether the previous methods failed or there are hidden problems hard to find for developers, resulting in such a high portion of the null pointer exception appearing.

Therefore, we conduct a deep analysis of the clusters related to null pointer exceptions, and eventually discover that the null pointer exception in the ten apps often includes the following cases: (1) Parent window and child window are present at the same time. However, the child window is still alive when the parent window is killed, at this point, the child window cannot access the *pid* of the parent window; (2) During an app loading process, the user performs a refresh or other click operations, resulting in some resources not being successfully initialized; (3) Abnormal connection conditions, such as Internet disconnections, Bluetooth disconnections, where some network-related resources are inaccessible; (4) When an application parses a file that cannot be read or written because the file is corrupted; (5) Device resources cannot be found when using hardware devices such as NFC, camera, and so on. (6) Using an uninitialized object, such as a database object, a component object, and so on.

Surprisingly, the majority of null pointer exceptions occur during process reuse (38.8%). We found that these collapsing reused processes are used in an app survival technology. In particular, in the Android system's process management mechanism, if an app is switched to the background, it may be killed by Android's low-memory kill mechanism. Once the app is killed in the background, it will be unable to receive real-time message notifications pushed by the app server, which may greatly diminish the user experience of popular apps. Therefore, app developers have introduced an anti-system technology—app survival technology—into the Android system's process management. From Android 5.0 to Android 12.0, some app survival technologies have been developed one after another, and the process management model of Android OS has also been continuously upgraded to resist app survival.

Before Android 5.0, processes fork out by apps through local methods could not be controlled by the system. When the system kills an app process, it only kills the Java process activated by the app. After the continuous improvement of Android process management, the operating system may kill the entire process group based on the *uid*, so the native process cannot escape from being killed. After all these, Android process survival technologies are divided into two aspects: keeping the process from being killed by system, or being able to revive the process after it is killed. For the first case, most apps use the method of constantly raising their own process priority to prevent being easily killed by the system, but this method has stuck with the problem of app priority competition. However, in the end, the Android system can still kill the process according to its process management strategy, which means that keeping the process from being killed fails and is not available. In order to maintain "immortality", some apps choose to intervene to change the process management mechanism and quickly revive the killed process.

In the details of techniques that kill processes, killing a process in Android system is achieved by calling two APIs: *killBackgroundProcesses* and *forceStopPackage*. In the native Android system, the former is difficult to kill the target process totally. The reason for this is, in detail, after calling

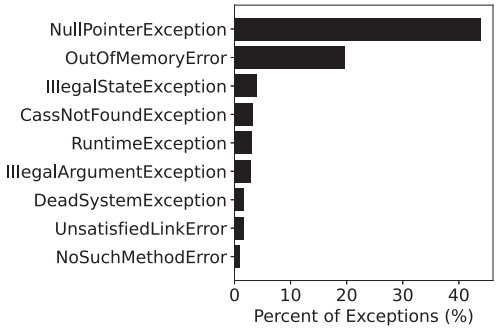

Fig. 16. Percentage of top 10 exception types.

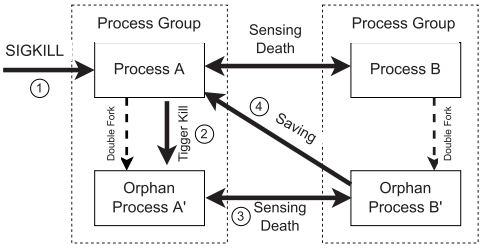

Fig. 17. Diagram of process preservation method.

the API, the system is not directly kill the process, but stops its service and restarts the stopped process when needed. Then the process will remain in the background. Meanwhile, the Android system provides another method to force kill a process (force-stop). In the implementation of force-stop, the system kills the target process group according to the *uid*, and loops 40 times to kill the group continuously, with an interval of 5 ms each time. Notably, if the app can start a bunch of new processes quickly enough after being forcibly killed, the new processes will survive after 40 loops, achieving the "immortality" of the process.

In practice, developers try to use binder messages to send the *ams* command in the native layer for starting a bunch of new processes within 5 ms, and employ a novel mechanism to increase resurrection rates. More specifically, developers use two processes to keep each other alive. The two processes receive information about each other's death by listening to file locks. To avoid two processes being killed at the same time, each of them generates a child process through forking, and the two child processes are also associated with a file lock. The process has at least one partner capable of reviving the killed process group. For example, as shown in Figure 17, two processes A and B are created, both of which are associated with each other through a file lock, so that one process can start the other one that is killed. At the same time, process A and process B go through two times forks to produce an orphan process A' and an orphan process B'. Similarly, process A' and B' also establish a file lock association between them. In this way, if process A is killed, process B will immediately perceive it. Moreover, process A and A' belong to the same process group, so the operation of killing process A will trigger the system to kill process A'. When process A' dies, process B' will immediately perceive it and start A process. Therefore, these four processes form an iron triangle among each other, ensuring their survival rate.

Although the above solution seems perfect, the actual situation on Android-based devices in the wild is very complex. Some process groups may not be fully resurrected within 5 ms due to ANR/SNR and other issues. Once the critical processes (A/A', B/B') cannot be resurrected, they will be completely killed. If the app is restarted (i.e., its activity is restarted) before all the processes are killed by the Android system, the process and its resources (particularly the static in-memory resources within singleton objects) will be reused (although they have been released) rather than being reinitialized. As a result, the app is then likely to perform illegal accesses to the released in-memory resources within singleton objects, leading to app crashes. However, such a situation is difficult to detect in offline testing.

Out of Available System Resources: The second root cause is out of available system resources. In Android, system resource has a correlative dependence to handles and memory resources. For handles, their insufficiency prevents the operating system from allocating new handles to the next

activity. We note that third-party libraries always apply numerous handles and threads for logging and performance acceleration. Massive handle acquirement brings about inadequate handles and may cause `OutOfMemoryError`, `IOException`, `SQLException`, `StackOverflowError` exceptions (3.1). As for memory resources, OOM (out of memory) crash (23.1%) is generally aroused by a bitmap and memory leak. Although the mechanism of memory allocation and reclamation with **garbage collection** (**GC**) [52] are carefully designed and updated timely, bitmap and memory leaks are still challenging problems for developers.

A bitmap problem is observed under resource-constrained scenarios, for example, when high-workload tasks are performed or high-resolution images are loaded. The size of a bitmap image is usually larger than a regular image, leading to occupying more memory [38]. In practice, there does not seem to be any silver bullet for addressing the bitmap problem. Additionally, we summarize several exceptions that would lead to memory leak: (1) static related leak: the lifecycle of static objects along with the whole process and is unable to be reclaimed [55]; (2) resources leak: such as cursor and I/O stream shut down, bitmap recycle with cache are not released in time [16]; (3) handler leak: handlers hold the reference to the activity thus hard to recycle [18]; (4) registering reference methods: the registered objects keep reference to the activity [17]; (5) cached objects are not removed [10]. OOM exception occurs if memory resources are insufficient and unable to allocate, which will also lead to memory leaks. Lastly, we confirm our findings that video apps are more prefer to crash as their unpredictable video size and high-pixel frames of pictures.

Illegal Operations: The third root cause is ascribed to illegal operations. As shown in Figure 16, we collect exception types from crash logs and exceptions inherited from RuntimeException including `NullPointerException`, `OutOfMemoryError`, `IllegalStateException` and so on. Illegal operations are aroused by inputting invalid parameters (`IllegalArgumentException`), math operation error (`ArithmeticException`), array operation error (`ArrayStoreException`, `ArrayIndexOutOfBoundsException`, `NegativeArraySizeException`), converting data type incorrectly (`ClassCastException`, `NumberFromatException`), and so on. Multi-process competition may also cause `IllegalThreadStateException` and `IllegalStateException` when missing corresponding relationships of variates. While these exceptions are not reported during the code compilation phase, only found to be responsible during the actual application. Thus, we can hardly propose feasible solutions from the external environment. It turns out to be issue for future development to deliver enhanced calibration and legitimacy testing, and also improve the coverage of the testing environment [9]. Additionally, all five types of apps exhibit more or less the same exception effect.

Compatibility Errors: The fourth root cause is due to compatibility errors. Based on the function of APIs delivering a program with clear instructions [56], users expect to upgrade third-party code "without thinking" when the system is upgraded. In fact, Android applications are generally compatible forward with newer versions of the Android system, which indicates that the general rule for an app is to keep compatibility down as far as possible. Thus out-of-step changes to APIs, such as inconsistent interfaces of the system SDK and third-party library, may bring about failures to call at runtime, and even report a crash directly. More specifically, when the API of a lower version system calls a third party with a higher version of the API library, the `NoSuchMethodError` or `UnsupportedClassVersionError` crash occur. When it comes to the state that the native library of the system and application is different strikes information `UnsatisfiedLinkError`: Native method not found, which belongs to Android development error. And source code lacking the initialization of application class may trigger `NoSuchMethodExeption`, `ClassNotFoundException` or `FileNotFoundException`: Did not find the method, class or file. It is even more challenging to fix these defects, since there is no silver bullet for bugs and defects in software engineering.

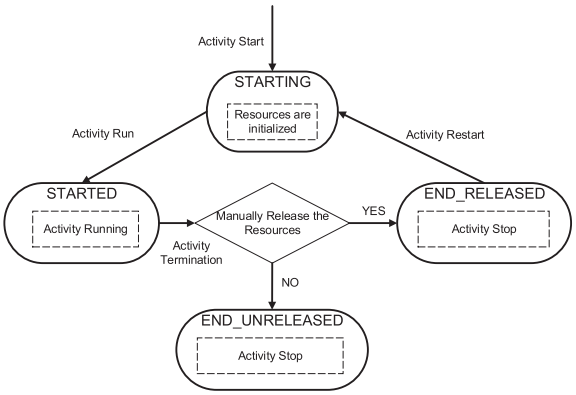

Fig. 18. The lifecycle state machine of class `SingletonPlus`.

4 PRACTICAL COUNTERMEASURES

Our multifaceted findings on root causes of app crash in Section 3 provoke us to consider more about the current technologies on the market and the solutions combating Android app crashes. Accordingly, in this section we present insightful guidance for addressing several root causes in Section 4.1, and we design practical approaches to achieve the offline testing of multiple threads. Our methods have enhanced the impact on app testing and real-world running in Section 4.2.

4.1 Guiding Principle

In this study, we demonstrate that collaborating with multiple app vendors offers a unique opportunity to reveal the landscape of suboptimal mobile app performance, even when the performance issues stem from the underlying framework/OS. The representativeness of our study is based on two ground-truth facts: (1) the same app will run on heterogeneous devices and platforms, and (2) the apps themselves are highly diverse and popular; then we discuss our opinions for enhancement from both developers and carriers perspectives, instead of extracting room of improvement from each root cause and summing it up. In addition, we notice that app vendors often have more incentives to collaborate with researchers (compared to device/OS vendors), likely because even professional app developers may not have the full-stack expertise in particular on the device/OS internals.

For developers: Recall that there are defects in source codes for first two root causes (i.e., most failure to access defined variables and insufficient system resource errors). Secondly, OOM (out of memory) as well as several kinds of exceptions (e.g., resource leak, handler leak, and etc) are caused by the large amount of handles and threads the third-party applied for logging and performance acceleration. As today's smartphones have more physical memory, developers may get a false impression that memory resources would not become a critical issue. In particular, developers oftentimes ignore the multi-tasking nature of modern mobile OSes. When system is multi-tasking, it imposes a limit on per-app memory allocation. Larger content sizes (e.g., high-resolution images and videos) also make today's mobile apps more "memory-hungry". Therefore, mobile developers should remain vigilant to excessive resource usage that may cause poor app performance or even crashes.

A potential way to eliminate the process reuse exception (the most app crashes) is to align an activity's lifecycle with the lifecycle of the underlying process. We instead propose an efficient solution, by enabling the currently stateless singleton objects to "perceive" the lifecycle states of their associated app activities. Then, singleton objects can properly manage resources by themselves,

so as to fundamentally address the problem. Specifically, we build a novel class `SingletonPlus`, which is an activity lifecycle-aware singleton that can automatically and correctly handle the in-memory resource management throughout an app's activity lifecycle. In detail, a `SingletonPlus` object maintains a lifecycle state machine that synchronizes with the app activity's current lifecycle state and in-memory resource state, as shown in Figure 18. To achieve this, the object registers handlers for the corresponding activity lifecycle state transition signals. Accordingly, upon activity start, the state machine first enters the `STARTING` state, and then switches to the `STARTED` state after the object's resources are initialized. Upon activity termination, if developers manually release the in-memory resources, the state machine changes to the `END_RELEASED` state; otherwise, it goes to the `END_UNRELEASED` state. In the above way, when the activity starts and the resources have been released (i.e., the previous state is `END_RELEASED`), we reinitialize released resources to avoid crash-prone illegal resource access; otherwise, we do not perform reinitialization to avoid unnecessary overhead. The above-described approach is easy to adopt. App developers can simply inherit the `SingletonPlus` class when building their own singleton classes to avoid the crashes.

For carriers: Next, we uncover that because of the insufficient workload of offline testing and improper models for carriers [25], applications are subject to poor stability and robustness. Thus, we bring up advice triggering the third, and fourth root causes as well as the first one from the carriers' perspectives to realize strong stability and robustness of offline testing and models.

First of all, for the errors in codes, some software shows their vulnerabilities. To address the code defects, the app developer community oftentimes devises various fixes to address system bugs and enhance system performance. While our experience about the illegal operation and exceptions indicate that these seemingly "ad-hoc" solutions may reflect issues deeply in the framework/OS. And SDK/OS vendors are responsible for ultimately patching the underlying system should carefully investigate them.

Moreover, for API incompatibility, we suggest that interface calls should be more appropriate for carriers to construct, and carriers consider more about the complexity of the connection between applications and systems. Moreover, besides the above improvement guidelines, a compatibility test should run both online and offline after adding a new third-party library to application.

Further more, we consider certain crash reasons (e.g., process preservation technology and compatibility errors) are contributing to the high complexity of app development. For example, the reliance on each other appears in process preservation leads to resource management error. Generally, such inter-dependencies are difficult to test in an offline environment. To have a comprehensive and in-depth insight into the process competitive issue, we propose that for the carriers, sufficient tests including white-box tests to observe how the internal code works are essential. Meanwhile, application operators should fully understand the mechanism of Android system, and be more aware of the competition relationship when processes are concurrent and multiplexed. Other than those, making a model to test the maximum competition case before delivering products is quite vital for carriers.

4.2 Large-Scale Deployment and Evaluation

After communicating with developers of applications about our guidance principles and simulation of experimental effects on the current popular crashes collected, our instructions and solutions are approved, also well received. Further more, to evaluate the effectiveness of guiding principle, once again we work with the developers of the 10 collaborative apps to integrate our optimizations to these apps. We send the above findings and samples feedback to the collaborative developers, then they revise the code accordingly and repackage it for release. The updated apps are released to opt-in users for further crash data collection for 31 days. This time, the total number of opt-in users is 25 million.

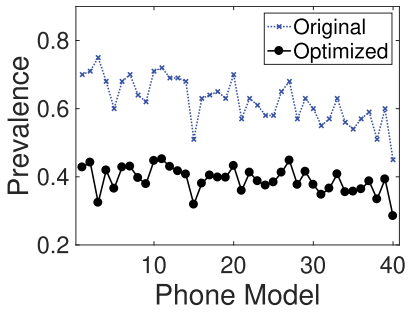

Fig. 19. Prevalence of app crashes in a whole month without and with our countermeasures.

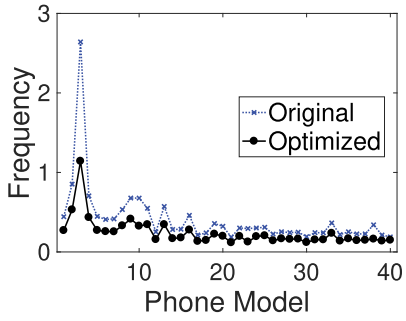

Fig. 20. Frequency of app crashes in a whole month without and with our countermeasures.

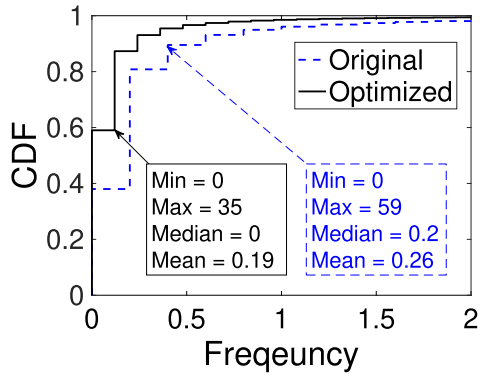

Fig. 21. Number of app crashes on a single phone in a whole month w/o and w/ our countermeasures.

Thanks to our countermeasures, the crash events are reduced by 40.4%, as compared to the previous situation. We observe that all the crashes incurred by premature release of static resources within singleton objects have been eliminated; the vast majority (~85%) of crashes caused by multi-process competition failures are also avoided; many NoSuchMethodError and UnsatisfiedLinkError crashes are also fixed. After re-statistical analysis, we find that the proportion of four major root causes has undergone new changes. To be specific, the percentage of failure to access defined variables decreased from 60.2% to 36%, and the percentage of out of system resources increased from 26.2% to 44%, and the percentage of illegal operation and compatibility errors are now 6.72% and 13.28% respectively. Obviously, memory management is now the most critical to Android app reliability.

As shown in Figures 19, 20, and 21, both the prevalence and frequency of app crashes are substantially decreased. In particular, crash reduction was observed in all the studied phone models, indicating that our countermeasures are general. At the same time, to understand the overhead of our countermeasures, we perform benchmarks on all the 40 major phone models as listed in Table 1. The results show that our design incurs little overhead to a low-end Android device: 2% CPU utilization, 10-KB memory, and no storage space.

5 RELATED WORK

Capturing App Crashes. Prior work has proposed both static and dynamic approaches to capture (potential) app crashes. For example, static code scanning is extensively utilized to examine

software flaws that may well lead to app crashes, such as buggy code patterns [8, 51] (e.g., null pointer exceptions and concurrency race conditions) and improper resource usages [11, 53]. However, these efforts cannot reveal app crashes at run time. Thus, researchers have also conducted dynamic behavior analysis to capture app crashes in real time, by injecting UI events [1, 20, 30, 45], simulating symbolic execution [2, 44], and so forth. Clearly our study takes the dynamic approach, while paying special attention to large-scale deployment in the wild. To this end, we develop the user-space tool CrashCapturer and integrate it into 10 collaborative apps for crowd-sourcing measurement.

Dissecting App Crashes. Both small-scale analysis on real devices [26, 29, 41] and large-scale analysis on emulated devices [7, 22] have been made to identify the root causes of app crashes. For instance, Gomez et al. [26] unravel nine categories of root causes for context-sensitive app crashes, which are tested with only five devices. Using a number of Android emulators, Cai et al. [7] identify the root causes of crashes incurred by app incompatibilities for 62,894 apps, and Fan et al. [22] distill 11 common causes of app crashes incurred by Android framework exceptions. Different from the above studies, we analyze real-world app crash data at scale, reveal hundreds of root causes, and point out the four major root causes that lead to the majority of app crashes.

Addressing App Crashes. There are both manual [36] and automated [21, 42, 50] methods designed to avoid or mitigate certain kinds of app crashes. For example, Kong et al. [36] manually extract the fixing templates for 17 types of exceptions that could trigger app crashes. Besides, Tan et al. [50] develop a search-based automated repairing tool `Droix`, which integrates various open-source utilities [3, 4] to patch/replace the programs that may result in app crashes. Our work is complementary to the above from a distinct perspective: instead of directly fixing the problematic code in Android apps (which is usually a heavyweight or error-prone job), we design lightweight approaches to help the developers conveniently avoid ~40% app crashes.

6 LESSONS LEARNED

Collaboration with multiple popular app developers offers a unique opportunity to study the mobile ecosystem. Researchers use many ways, including developing their own crowdsourcing apps [31], collaborating with device/OS vendors [39], and collecting data from carriers [48], to study mobile apps' performance in the wild. In this study, we demonstrate that collaborating with multiple app vendors offers a unique opportunity to reveal the landscape of suboptimal mobile app performance, even when the performance issues stem from the underlying framework/OS. The representativeness of our study is based on two facts: (1) the same app will run on heterogeneous devices and platforms, and (2) the apps themselves are highly diverse and popular. In addition, we find that app vendors often have more incentives to collaborate with researchers (compared to device/OS vendors), likely because even professional app developers may not have the full-stack expertise in particular on the device/OS internals.

App-level fixes may help address issues deep in the framework/OS. The app developer community oftentimes devises various fixes to address system bugs or enhance system performance. Our proposed app-level fixes in Section 4 are simple yet effective, practical, and immediately deployable. Our experience indicates that these seemingly "ad-hoc" solutions may reflect issues deep in the framework/OS, and should be carefully investigated by SDK/OS vendors, who are responsible for ultimately patching the underlying system.

The subtle differences between PC and mobile app lifecycles remain a challenge for developers. We find that 38.8% of observed crashes are caused by developers' prematurely releasing static in-memory resources upon activity terminations. This is likely due to developers' lack of a clear picture of mobile apps' lifecycle and process management, which is different from

apps running on PC (and servers). While such a gap has been investigated by prior work [23], our study reveals its surprising magnitude in the wild, almost 15 years after mobile apps formed their industry. Our findings suggest that solving this issue requires efforts from multiple entities: not only developers, but also mobile SDK designers.

Mobile developers should still be cautious with resource usage. By practically addressing the crashes incurred by memory model inconsistency, we successfully reduce 40.4% of app crashes on the 10 collaborative apps. For the remaining 59.6% crashes, we find that they are mostly caused by resource under-provisioning and illegal resource usage. As today's smartphones have more physical memory, developers may get a false impression that memory resources may not become a critical issue. In particular, developers oftentimes ignore the multi-tasking nature of modern mobile OSes, which imposes a limit on per-app memory allocation. Larger content sizes (e.g., high-resolution images and videos) also make today's mobile apps more memory-hungry. Therefore, mobile developers should remain vigilant to excessive resource usage that may cause poor app performance or even crashes.

7 CONCLUDING REMARKS

In this article, we present our experiences in understanding and combating app crashes on Android devices at scale in the wild. By developing the CrashCapturer measurement tool and integrating it into 10 very popular Android apps, we obtain in-depth and comprehensive findings on app crashes in terms of their prevalence, frequency, evolution, and root causes in the wild. In particular, we discover that most of crashes originate from the knowledge gap between the developers of Android apps and OSes, and thus we design practical solutions to effectively fix them and uncover more crashes in real world, which have registered commercial deployment and testing. In the future, we will seek collaborations with phone vendors and OS developers for a more thorough understanding of apps' abnormal and malicious behaviors on diverse Android IoT and IIOT devices and platforms in the wild.

REFERENCES

[1] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. 2010. Diagnosing mobile applications in the wild. In *Proceedings of the ACM SIGCOMM Workshop on Hotnets*. 1–6.

[2] Saswat Anand, Mayur Naik, Mary Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM ESEC/FSE*. 1–11.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.

[4] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SOAP*. 27–38.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[6] Buildfire. 2021. Mobile App Download and Usage Statistics. Retrieved from https://buildfire.com/app-statistics/

[7] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in android. In *Proceedings of the ACM ISSTA*. 216–227.

[8] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *Proceedings of the NASA NFM*. 3–11.

[9] Peng Chen, Quansheng Zhang, Shuzhi Wang, and Xiaofeng Dong. 2011. The research of fastboot of embedded linux based on state keeping and resuming. In *Proceedings of the International Conference on Electrical and Control Engineering*. 4173–4176.

[10] Raymond Chen. 2006. A Cache with a Bad Policy is Another Name for a Memory Leak. Retrieved from https://devblogs.microsoft.com/oldnewthing/20060502-07/?p=31333

[11] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. 2016. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the IEEE/ACM ICSE*. 820–831.
[12] Android Developer. 2023. Android App-Not-Responding (ANR) Event. Retrieved from https://developer.android.com/topic/performance/vitals/anr
[13] Android Developer. 2023. Android Crashes. Retrieved from https://developer.android.com/topic/performance/vitals/crash
[14] Android Developer. 2023. Android Vitals Performance Monitor. Retrieved from https://developer.android.com/topic/performance/vitals
[15] Android Developer. 2023. Bug Fixes in SurfaceTexture. Retrieved from https://android.googlesource.com/platform/frameworks/base/+/19b6bcfd83eb7fb92ebd06d2fec89e308311f1d0%5E%21/#F4
[16] Android Developer. 2023. Inspect your app's memory usage with Memory Profiler. Retrieved from https://developer.android.com/studio/profile/memory-profiler
[17] Android Developer. 2023. Manage your app's memory. Retrieved from https://developer.android.com/topic/performance/memory
[18] Android Developer. 2023. SwitchingApps. Retrieved from https://developer.android.com/topic/performance/memory-overview#SwitchingApps
[19] Android Developer. 2023. The Slow Rendering of Android. Retrieved from https://developer.android.com/topic/performance/vitals/render
[20] Android Developer. 2023. UI/Application Exerciser Monkey. Retrieved from https://developer.android.com/studio/test/monkey
[21] Malinda Dilhara, Haipeng Cai, and John Jenkins. 2018. Automated detection and repair of incompatible uses of runtime permissions in android apps. In *Proceedings of the IEEE/ACM MOBILESoft*. 67–71.
[22] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the IEEE/ACM ICSE*. 408–419.
[23] Umar Farooq and Zhijia Zhao. 2018. RuntimeDroid: Restarting-free runtime change handling for android apps. In *Proceedings of the ACM MobiSys*. 110–122.
[24] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing Q&A sites (T). In *Proceedings of the IEEE/ACM ASE*. 307–318.
[25] Timothy Ghanem and Samer Zein. 2020. A model-based approach to assist android activity lifecycle development. In *Proceedings of the ISMSIT*. 1–12.
[26] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *Proceedings of the IEEE/ACM MOBILESoft*. 88–99.
[27] Google. 2023. Breakpad. Retrieved from https://github.com/google/breakpad
[28] Julian Harty and Matthias Müller. 2019. Better android apps using android vitals. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*. 26–32.
[29] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for android applications. In *Proceedings of the IEEE/ACM AST*. 77–83.
[30] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the ACM EuroSys*. 1–15.
[31] Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. 2016. Defdroid: Towards a more defensive mobile os against disruptive app behavior. In *Proceedings of the ACM MobiSys*. 221–234.
[32] Infer. 2023. Infer Static Analyzer. Retrieved from https://fbinfer.com/
[33] jambit GmbH. 2023. ptrace. Retrieved from https://man7.org/linux/man-pages/man2/ptrace.2.html
[34] jamesbriggs. 2022. Locality Sensitive Hashing (LSH): The Illustrated Guide. Retrieved from https://www.pinecone.io/learn/locality-sensitive-hashing/
[35] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. 2014. What do mobile app users complain about? *IEEE Software* 32, 3 (2014), 70–77.
[36] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. 2019. Mining android crash fixes in the absence of issue-and change-tracking systems . In *Proceedings of the ACM ISSTA*. 78–89.
[37] Giorgi Kvernadze. 2020. Locality Sensitive Hashing for MinHash. Retrieved from https://giorgi.tech/blog/locality-sensitive-hashing/
[38] Microsoft Learn. 2023. About Bitmaps. Retrieved from https://learn.microsoft.com/en-us/windows/win32/gdi/about-bitmaps
[39] Mingliang Li, Hao Lin, Cai Liu, Zhenhua Li, Feng Qian, Yunhao Liu, Nian Sun, and Tianyin Xu. 2020. Experience: Aging or glitching? Why does android stop responding and what can we do about it?. In *Proceedings of the ACM MobiCom*. 1–11.

[40] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, et al. 2023. Virtual device farms for mobile app testing at scale: A pursuit for fidelity, efficiency, and accessibility. In *Proceedings of the ACM MobiCom*. 1–17.

[41] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for android apps. In *Proceedings of the ACM ESEC/FSE*. 233–244.

[42] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *Proceedings of the IEEE/ACM ICSE-SEIP*. 269–278.

[43] Hamed Mirzaei and Abbas Heydarnoori. 2015. Exception fault localization in android applications. In *Proceedings of the ACM MOBILESoft*. 156–157.

[44] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[45] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: A practical tool for automated testing of android applications. In *Proceedings of the IEEE/ACM ICSE*. 15–18.

[46] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and scalable fault detection for mobile applications. In *Proceedings of the ACM MobiSys*. 190–203.

[47] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: Automatically revealing data loss bugs in android apps. In *Proceedings of the ACM ISSTA*. 141–152.

[48] Sanae Rosen, Ashkan Nikravesh, Yihua Guo, Z Morley Mao, Feng Qian, and Subhabrata Sen. 2015. Revisiting network energy efficiency of mobile apps: Performance in the wild. In *Proceedings of the IMC*. 339–345.

[49] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Communication of the ACM* 61, 4 (2018), 58–66.

[50] Shin Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *Proceedings of the IEEE/ACM ICSE*. 187–198.

[51] Shin Hwei Tan and Ziqiang Li. 2020. Collaborative bug finding for android apps. In *Proceedings of the ACM/IEEE ICSE*. 1335–1347.

[52] Wikipedia. 2023. Garbage collection (computer science). Retrieved from https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[53] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. 2016. Relda2: An effective static analysis tool for resource leak detection in android apps. In *Proceedings of the IEEE/ACM ASE*. 762–767.

[54] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1054–1076.

[55] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical static memory leak detection for managed languages. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 87–97.

[56] Weizhao Yuan, Hoang Huu Nguyen, Lingxiao Jiang, and Yuting Chen. 2018. Poster: LibraryGuru: API recommendation for android developers. In *Proceedings of the IEEE/ACM ICSE-Companion*. 364–365.

[57] Gaofeng Zhang, Yu Li, Xudan Bao, Chinmay Chakarborty, Joel J. P. C. Rodrigues, Liping Zheng, Xuyun Zhang, Lianyong Qi, and Mohammad R. Khosravi. 2023. TSDroid: A novel android malware detection framework based on temporal & spatial metrics in IoMT. *ACM Transactions on Sensor Networks* 19, 3 (2023).

[58] Boyan Zhou, Quan Cui, Xiu-Shen Wei, and Zhao-Min Chen. 2020. BBN: Bilateral-branch network with cumulative learning for long-tailed visual recognition. In *Proceedings of the IEEE/CVFCVPR*. 9716–9725.

Received 8 June 2023; revised 7 November 2023; accepted 20 February 2024