Open Access
Full Length Article

ORIGINAL ARTICLE

Adaptive Non-Crashing Functional Bugs in Android Applications

Amna Asif[1], Naeem Aslam[2], Sana Akhtar[3], Muhammad Kamran Abid[4], Ujala Saleem[5]

[1,2,3,4,5]Department of Computer Science, NFC Institute of Engineering and Technology, Multan, Pakistan

ABSTRACT

Because Android applications are so widely used, it is critical to make sure they are reliable and secure. Through completely automated functional fuzzing, this research presents a novel method for Android app security. Our approach uses sophisticated fuzzing techniques to methodically investigate the operation of Android apps, with a focus on the detection of non-crashing logic flaws. While non-crashing logic defects can introduce subtle vulnerabilities jeopardizing the overall integrity of Android apps, they are typically overlooked in traditional security testing, which concentrates on discovering and mitigating crashes. The current techniques for finding these logic issues that don't crash are frequently laborious, manual, and not very thorough. As of now, a general comprehension of functional defects remains elusive, impeding the progress of methodologies and strategies aimed at mitigating such issues. To address this deficiency, we investigate the fundamental causes, and symptoms of bugs, and test oracles, as well as the capabilities and limitations of current testing methodologies, using 401 functional defects from five prominent open-source and representative Android applications. This is the first systematic study of its kind. Several of the intriguing new findings and implications that our research uncovers cast light on the topic of addressing functional defects. By applying transfer learning and RegDroid to eight prominent real-world applications, we were able to effectively identify sixteen functional flaws.

Keywords: Automated Functional Fuzzing, Dynamic Analysis, Security Vulnerability, Logic Bug Identification

Author`s Contribution
[1,2,3] Data analysis, interpretation, and manuscript writing, Active participation in data collection
[4,5] Conception, synthesis, planning of research. Interpretation and discussion

Address of Correspondence
Amna Asif
Email: amna.asif@gmail.com

Article info.
Received: July 19, 2023
Accepted: December 04, 2023
Published: December 30, 2023

Cite this article: *Asif A, Aslam N, Akhtar S, Abid MK, Saleem U. Non-Crashing Functional Buds in Android Applications. J. inf. commun. technol. robot. appl.2023; 14(1):29-43*

Funding Source: *Nil*
Conflict of Interest: *Nil*

INTRODUCTION

If you were on the closing steps of completing an essential task or found yourself completely engulfed in the thrill of playing your favorite Android application then you may have experienced your mobile phone freezing, behaving weirdly, or simply becoming unresponsive to your commands. A noteworthy majority of the inconveniencing circumstances in this area are a result of non-crashing functional bugs—errors that are not severe

but can only go unnoticed, with no error reported. These errors are subtle yet consequential, leaving a user dissatisfied and probably mentioning the name of that application to tarnish its reputation. In the very strong environment of mobile application development, developers needed to work out the security problems and modernize their software to produce the reliable and best mobile apps for their users.

The text, that was improved, alternates between real narration about a problem and a scenario of a special effect. In such a fashion, the readers will learn about non-crashing functional bugs and their real effects.

Android applications have come to dominate the mobile application landscape, which has become an integral part of our daily existence. With the increasing prevalence of these applications, there is a corresponding demand for resilient security protocols to protect user information and guarantee the soundness of these platforms. A notable obstacle in this particular context pertains to the identification of non-crashing logic flaws, which frequently evade conventional testing techniques.[1]. Non-crashing logic flaws have the potential to result in severe security vulnerabilities and unforeseen behaviors, despite not causing imminent application failure. The vulnerabilities presented by these threats significantly compromise both user privacy and system stability. [1]Traditional testing methodologies have encountered challenges in accurately detecting and mitigating these nuanced yet potentially detrimental concerns. The challenges mentioned above are tackled in this study by investigating entirely automated functional fuzzing, an innovative and effective technique for identifying non-crashing logic flaws in Android applications.[2] Fuzzing, a method of testing in which unexpected or arbitrary data is introduced into a system, has demonstrated the potential to reveal a wide range of vulnerabilities. Moreover, we investigate the fundamental causes, and symptoms of bugs, and test oracles, as well as the capabilities and limitations of current testing methodologies, using 401 functional defects from five prominent open-source and representative Android applications named as WordPress, Simple note, Antenna Pod, K9 Mail, Anki Droid, Amaze, Firefox, and New Pipe. Our investigation yielded a plethora of fascinating new results and consequences that provide insight into functional fault reduction. When RegDroid and transfer learning were used to eight common real-world apps, a total of sixteen functional flaws were successfully found.

We concentrate on GitHub-hosted open-source Android applications due to our access to their public issues. To identify representative applications, we employed two metrics:

- app features
- and app prevalence, to refine the scope.
- Objectives

This study was originally undertaken to offer a domain of Android application security via functional fuzzing. Moreover, the study aims to identify the exact or approximate side of the flaws that do not cause crashes. The specific objectives are as follows:

The specific objectives are as follows: An innovative automated functional fuzzing process will be designed that is specific to mobile apps built for the Android platform. Such an advancement will be instrumental in uncovering vulnerabilities in modern Android mobile applications. Integrate extensive research on the existing goings-on of intentional, logical mishaps within the Android application.

Highlight the necessary steps that the researchers, security professionals, and developers should take to improve the security posture for the Android applications Defining types and root causes of function breaking fails that occur while devices are not falling, investigating their distribution and frequency, measuring consequences for user experience and normal operation, and providing appropriate solutions.

Assess the consequences on user experience, performance, and security of these vulnerabilities pointing directly to the need that these disadvantages should be addressed for the best quality of app functioning and user satisfaction.

The Research's Significance

Under this study, several crucial discoveries are brought forward ranging from intriguing findings to implications of research in the Android security domain. The main contributions include:

Detection and Rectification of Security Vulnerabilities: The research highlights the benefits of risk detection and prevention where insecurities, if left without attention,

spread and intensify quickly becoming very hard to fix when they are known and acknowledged in time. Normally, security demonstration of mobile apps is most important in the existing situation of the handling of sensitive user data in mobile applications.

Reliable and Automated Approach: Examines as a current problem how to develop an easy and automated way of checking the non-crashing logic flaws of mobile apps. The strengthening of the security system is required for the case of properly dealing with confidential information of users.

Advancement of Knowledge: The results provide a great background to add to the knowledge pool on the topic of the obfuscation of non-crashing bugs in Android apps. This is indispensable for continuous development of security means that adjusts to the ever-changing environment of mobile applications, everywhere in the world.

Practical Approaches for Security Enhancement: automates the functional fuzzing process and reinforces application security on Android systems through practical techniques. The applications of these methods offer technical guidance to researchers, security specialists, and developers to ensure the general reliability of Android applications

Advantages of Automated Functional Fuzzing: The concept of automatic functional fuzzing being revealed as a possible route to the fixing of non-crashing logic defects in Android apps is the main takeaway from the study. This set of things includes extensively testing and detecting sophisticated flaws and providing a real-time dimension thus improving the reliability and the very essence of system applications.

Automated functional fuzzing is an advantageous approach to rectify non-crashing logic vulnerabilities in Android applications owing to its capacity to conduct thorough application testing, efficiently detect intricate flaws, and simulate real-life situations—in essence, this enhances the quality and dependability of the applications.

1.1. Research Questions

Q1: What is the effectiveness of a fully automated functional fuzzing approach in identifying and addressing non-crashing logic flaws that may occur in Android applications?

Q2: How do these functional flaws manifest themselves? Which applications are impacted by these bugs?

Q3: To improve the detection of non-crashing logic flaws in Android applications, which novel methodologies can be incorporated into an automated functional fuzzing solution? In what types of test oracles do these functional flaws need to be identified?

To identify vulnerabilities, flaws, or unexpected behaviors in software, functional fuzzing entails the execution of a substantial quantity of inputs that are generated or made to appear randomly or semi-randomly.[2] By employing this methodology, security vulnerabilities and other complications that may elude conventional testing techniques Advanced testing methodology "Fully Automated Functional Fuzzing" involves the automated generation of a variety of inputs to the software system to identify vulnerabilities or flaws. By methodically investigating the functionality of Android applications, this method seeks to detect non-crashing logic flaws.[1], [4]. Non-crashing logic bugs, which are flaws in the logic of the application that may result in undesirable behavior, security vulnerabilities, or functional issues, are the main objective.

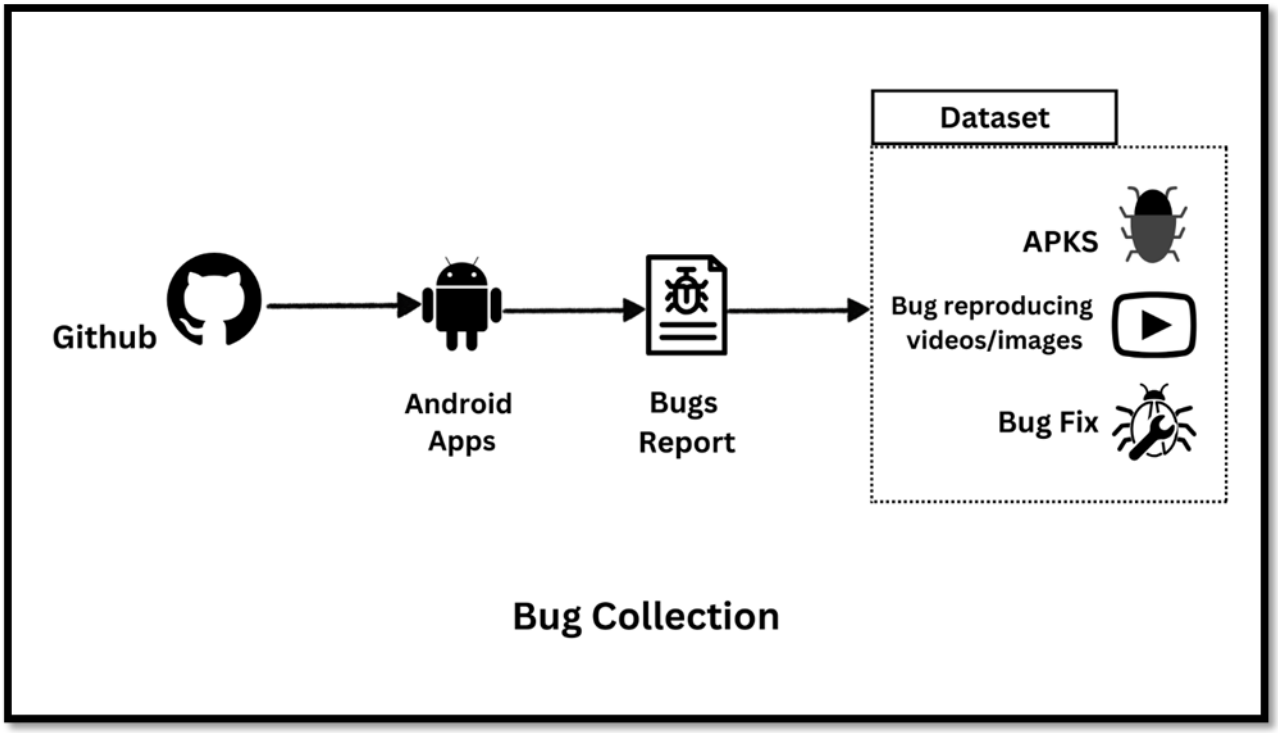

Figure 1. Bug Collection

LITERATURE REVIEW

Researching non-crashing bugs in software systems turns out to be a broad area as a systematic study of the existing literature reveals a lot of resources that were relevant to the study. The significance of these errors is then the fact that they might reduce how much a certain application could work or how it would be perceived by the users. To take the consideration of seeking through a complete critique and also involving advanced studies, we perform a detailed inspection of recent innovations, theories, and frameworks which are extensively advocated by research paper publications published in high visibility journals such as the 2023 one.

Advanced Level Work in 2023: In the year 2023, the sophisticated level of work related to Impact bugs refreshed the frontier science whitened with new colors. Through a set of studies carried out during this period, a careful examination of them results in the absorption of cutting-edge, sophisticated techniques and the ability to understand them fully as they relate to these types of software flaws. Integrating Relevant Studies:

The review section of our paper delves into the non-crashing bugs of software design reaching beyond the traditional classifications and extending into the intricate facets of erroneous logic, flawed calculations, UI-related problems, and display data errors. By combining the results of many studies and a complex scenario of non-crashing bugs and different ways of their appearance, we will have an opportunity to deliver a complete and representative picture of unique difficulties.

Frameworks from Impact Factor Research Journals: We only consider projects from such fields to make sure we have the highest level of validity. We give priority to research published in journals that have high impact factors and are found at an international level. With the help of these well-known theories and frameworks, we strive to achieve a concrete basis for our research which will be in tune with the latest practices and the approaches that are approved inside the academic field.

User interface (UI) problems are an additional classification of non-crashing defects. Issues with the visual components of the application, including faulty icons, inaccurate labeling, or inconsistent layouts, may contribute to a less-than-ideal user experience.[5]. Data

display issues are frequently encountered in non-crashing flaws as well. These concerns become apparent in the form of formatting mistakes, omissions of information, or inaccurate data representations, which have a detrimental effect on the precision and dependability of the data that is provided to users. Memory breaches are classified as non-crashing flaws because they cause the software to progressively consume larger quantities of memory as time passes. [6]Although not immediately precipitating a system breakdown, these concerns may ultimately result in compromised system performance and instability. Compatibility issues represent an additional category of non-crashing defects. Diverse functionalities may ensue due to incompatibility with particular operating systems, devices, or browsers, which may cause inconsistent performance and user discontentment.

Security vulnerabilities are crucial non-crashing flaws even though they do not result in disruptions. Consequences such as unauthorized access or data leakage may jeopardize the confidentiality and integrity of user information; therefore, the software's security posture must be promptly maintained.

Functional fuzzing is a highly effective methodology utilized in software testing to detect security vulnerabilities and issues through the exposure of a program to an extensive variety of inputs. Particularly with regard to Android application security and the detection of logic flaws that are not crashing ones, it is essential to have full awareness of the various fuzzing methodologies[15]

The following are examples of sophisticated frameworks and methodologies used to identify non-crashing functional defects in Android applications:

Automated Functional Fuzzing: Identifying vulnerabilities and non-crashing flaws by generating and inputting a large number of test cases, including those that are invalid, unexpected, or arbitrary, using the tools provided.

Dynamic program analysis consists of methods for identifying issues such as memory leakage, resource contention, and unanticipated behavior by analyzing the behavior of a program during execution.

Symbolic Execution: An analysis technique utilized to comprehend all potential paths of program execution, thereby enabling the detection of vulnerabilities and flaws that do not result in crashes.

Machine Learning and AI: detecting anomalies and potential non-crashing flaws by analyzing patterns in code or user interactions using algorithms.

Consistent factors such as input validation errors, memory leakage, and concurrency problems are brought to light by the findings. These statements suggest that improved development practices are required, such as strict adherence to coding standards, effective resource management, and thorough testing. Covert methodologies may encompass sophisticated memory management techniques or specialized diagnostic strategies to efficiently address these vulnerabilities.

2.1.Gaps

The present research gap has been found in research endeavors, which fail to address the adoption issues in the specific domains of modern Android applications. It is crucial to emphasize the importance of this gap for several reasons: It is crucial to emphasize the importance of this gap for several reasons:

Specificity for Android System:

Mobile apps' existing research has important findings although they cannot directly be applied to Android. Consequently, a new study focusing on defects specific to Android systems is essential. Such a focus is complemented by the more intricate analysis of large groups of functional problems. This makes the study more comprehensive, with an understanding of the complexities of Android apps' landscape.

***Incompleteness in Previous Studies*:**

Some approaches tend to dwell on certain app categories while others completely run out of the other apps. This omission thereby shows how there is a must for a far-reaching inquiry that emanates from a variety of Android apps. During this scrutinization, the interface and usability shortcomings throughout different app features are uncovered, meaning the tester can view the product as a whole and not just look at its parts.

User Experience and Performance Impact:

Presenting this situation appears to be an important thing when we consider the possible outcomes. Non-crashed, but still operational imperfections can negatively impact greatly the user experience, it may lead to discontentment, society's reluctance to utilize it, and poor reviews by society. In addition, the poor performance of Android applications may be attributed to lower response

times and the tendency of the applications to exhaust the battery rapidly. This is mostly due to the invisible anomalies that may be present in these applications.

Table 1. Literature Summary

Paper	Published Date	Technique	Finding	Limitation
[4]	October 5, 2021	Independent View Fuzzing.	Found 364 non-crash bugs	Lack of appropriate test oracles
[7]	August 23-28, 2021	Quantitative and qualitative analyses	52 real, reproducible crash bugs	Difficulty handling dynamic UI elements
[8]	November 30, 2023	automata-based trace analysis	Found tool weaknesses	The complexity of analyzing long event traces
[9]	July 15, 2023	static analysis approach	Detect 302 issues	No runtime monitoring
[10]	Aug 18, 2020	Machine learning algorithms	Different algorithms have different results on performance	Difficult in extracting vulnerabilities
[11]	Sep 20, 2021	Hybrid fuzzing	Analysis of different tools	Limited scalability
[12]	October 10–14, 2022	Mutation-based and Hybrid fuzzing	Evaluation of 8 fuzzers	No clear superiority of hybrid fuzzing techniques
[13]	July 26, 2023	Differential testing	Removal of false positives by automated filtering strategy	Numerical instability
[14]	Oct 14,2022	Black box fuzzing	More number of vulnerabilities detected	Limited code coverage and crawling module

RESEARCH METHODOLOGY

The principal aims of the research are twofold: to accomplish this end, a qualitative approach to user experience will be adopted by handbook checking having in mind some automated method to detect functionality defects[16]. Regarding the ability to locate and determine the volume of functional faults, it is the computer-based tools that are mostly used. These tools present a user interface that enables programmers to run these tools to observe the behavior of the application under controlled conditions while it performs a test scenario. The implementation of automated tests aims to imitate the user behavior to reveal problems, for example, unexpected system shutdowns, interface errors, and ups. and downs in performance. Undoubtedly, conventional data analysis methods, such as machine learning, qualitative analysis, and statistical approaches, are commonly utilized in the domain of functional flaw detection within Android applications.

Code and user interactions will be analyzed by machine learning algorithms to identify patterns that are suggestive of functional flaws that do not result in crashes. The implementation will involve the utilization of methodologies, including supervised learning with support vector machines, neural networks, or Random Forests. Code, documentation, and user feedback will be examined as part of the qualitative analysis to identify recurring themes and problems associated with non-crashing functional defects. Insights into the fundamental

causes and attributes of these insects will be derived through the categorization and interpretation of the qualitative data using methods such as thematic analysis and content analysis. The analysis will employ statistical methods to examine the frequency and distribution of non-crashing functional defects among various categories of Android applications.

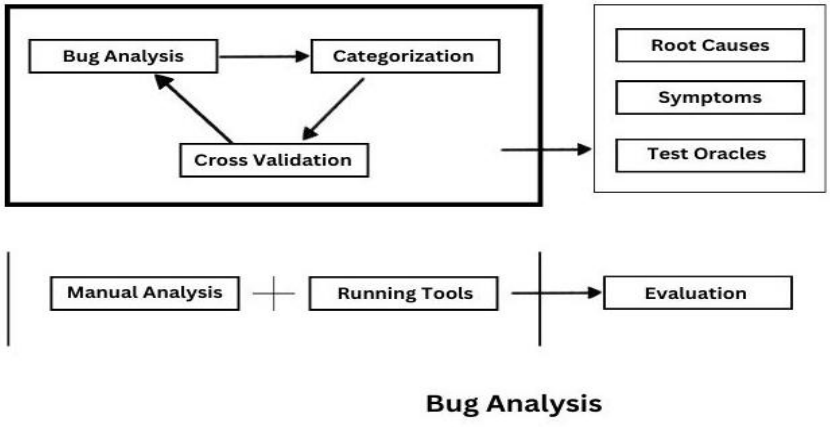

Figure 2. Methodology

Transfer learning is utilized less frequently in the analysis of functional flaws in Android applications than in other domains such as natural language processing or image recognition. Usually, in response to a lack of labeled data for the particular task at hand, transfer learning is implemented when a model that was trained on one task is repurposed or fine-tuned for a task that is related to it.[17] However, the inherent characteristics of functional flaw analysis in Android applications may not be well-suited to the pre-trained models that are frequently employed in transfer learning. Attributes of application behavior, user interactions, and software architecture that are distinct to the field of mobile application development are frequently incorporated into functional flaw analysis.

Regarding non-crashing functional flaws in Android applications, the transfer learning method has several limitations and possible biases that must be taken into account. A significant drawback is the possibility of overfitting to the source data, which occurs when the model acquires knowledge of particular patterns from the source domain that lack generalizability to the target domain. This may transpire if the model architecture is overly intricate or if the source data fails to accurately represent the target data. Furthermore, a challenge may arise when there is a domain shift occurring between the source and target domains, as the flaws may exhibit distinct manifestations in each domain, thereby impeding the model's ability to efficiently transfer knowledge.

The Challenge of Methodology: Certain Collective Biases

Although we carry out the research using rigorous methods, we should realize that there are certain limits as well as biases that we face with the exact procedures.

Transfer Learning Limitations:

Transfer learning as an efficient methodology applied to different fields has its issues too, when it comes to application to Android apps which, being complex entities, depend on a lot of related factors. Another main scope of existing barriers is the possibility of the model being fitted too much to the source data and then it may contain patterns that do not have a broad applicability to the target area. The structure of the model, which is quite complicated, or rather the data representation that doesn't keep in line with source and target information may lead to this constraint. Further, the problem of domain shifts between source and target domains may limit the transfer

of knowledge needed to differentiate the specific factors related to various forms of defects.

False Positives in Automated Testing are one of the crucial concerns of the testing field.

We tried RegDroid for automated functional fuzzing, a popular fuzzing technique that may yield false positives. Although the incorrect alerts of RegDroid show a situational standard with the others, it remains stated that the false alerts may result in detrimental effects. The false positives in the case may attract the attention of the developers but they may fail to notice the real defects because they overly devote themselves to resources used to address these issues. This will increase the chance of holding up the process, and raising the project expenses, and, at times, it might be difficult to continue the new project at the normal pace.

Applicability to Specific Apps:

The results, therefore, can be considered as a citation of an example of the grade of diversity in a set of eight interesting apps which can hardly be regarded as an objective basis for establishing a more widespread and generalized rule. Apps selected here have proven to be effective, with regular updates, offering a wide range of different tools, which can work to avoid this. On the other hand, some apps may offer characteristics and be more advanced, so these could influence the generalization of the study's results.

Manual False Positive Research:

The evaluation of false positives relied on a very complicated procedure, which meant the risk of human error could not be easily avoided. The attempt to make the process of manual interpretation more and more precise can be considered as an advantage over implementing inertia-based non-contact sensors but subjectivity remains the limiting factor.

RESULTS AND DISCUSSION

The fundamental aim of our study was to assess the effectiveness of entirely automated functional fuzzing in identifying non-crashing logic vulnerabilities in Android applications. In pursuit of this objective, we executed a comprehensive automated functional fuzzing methodology by employing transfer learning. The approach specifically focused on performance metrics, flaw detection, and severity assessment. Our study conscientiously evaluated the overall performance of the utilized fuzzing methodology, in addition to detecting bugs and analyzing their severity. Factors such as resource utilization, execution time, and scalability were incorporated into this assessment. The operational effectiveness and pragmatic feasibility of our automated functional fuzzing methodology in tangible situations are significantly enhanced by the data points that comprise these performance metrics.

Functional defects can be validated using automated testing by executing test programs that simulate the circumstances in which the problem manifests and confirming that the anticipated behavior is not fulfilled.

Dataset

The dataset functioned as the fundamental basis for our inquiry, guaranteeing the strength and generalizability of our conclusions in a multitude of situations.

Fairness in Selections of and Testing Criteria for the Function-Compromising or Unessential Deviations or Decurrent Problems.

Reproducibility Criteria:

Our dataset should only include functional defects that can be repeatedly demonstrated to have a direct impact on daily operations. These kinds of failures require us to download the same application version of the automobile that the defect has been reported on and verify the existence of the issue based on the time of our analysis. As for the rest, bug reports require not only the fault itself but also the picture or a video to ensure trouble reproduction even if we were not able to do this.

Explicit Link to Modification:

This direct link to the separately modifying segment is eminently required. The assessment criterion makes sure that the corrections undertaken are only associated with the established concern, hence this simplifies the process of tracking the origin of the problem. This tie-up allows for understanding the background of the problem and encourages detective workflows.

Device and Platform Neutrality:

To strengthen the generalization of our conclusion from the study, we did not include issues that depend on the platform or device of a speaker. The focal functioning problems should be relevant on standard Android mobile

handsets, thus, normalizing the set data platform or device-associated complications.

The dataset comprises the quantity of Android applications as well as the way these applications are distributed among various categories The information was obtained from the app store.

Valid functional defects were compiled into a dataset of flaws. Specifically, we retained only those functional problem reports out of 1,623 total that were reported.

It fulfills all three of the subsequent criteria. The corresponding functional defect must be reproducible on the malfunctioning app version at the time of our investigation; or the bug report must include explicit videos or images that reproduce the fault, notwithstanding our inability to replicate it ourselves.

Secondly, the modification is explicitly linked to the corresponding issue. Because the modification simplifies the process of determining the source of the problem. Third, the analogous vulnerability affects standard Android mobile devices. We omitted problems that were contingent on particular platforms or devices, as they may have been platform or device-specific complications.

Table 2. No Bugs Selected

Apps	# Bugs
WordPress	68
Anki Droid	82
Amaze	30
Firefox	44
Antenna Pod	41
K 9 Mail	36
New Pipe	65
Simple Note	35
Total	401

In previous work, they created (1) the faulty application (the APK file), (2) the videos or images that reproduced the problem, and (3) the corresponding bug fix for every legitimate functional issue. We found two more defects from ambiguous reproducing information, such as absent explicit reproducing steps, flawed app versions, or videos/images that reproduce the flaws.

Table 3. Bugs Found

Type ID	Bug Type	#Bugs
T1	GUI design issue	1
T2	A function cannot proceed	1

GUI design issue is due to incorrect algorithm implementation. In software applications, algorithm correctness is of the utmost importance. Algorithms play a critical role in software development by providing direction for task execution, data processing, and decision-making. It is imperative to guarantee the accuracy of algorithms for a multitude of reasons; doing so has a direct impact on the dependability, effectiveness, and credibility of software applications. The procedure for deducing independent views and preserving their active view information while a seed test is executed is described in Algorithm 1.

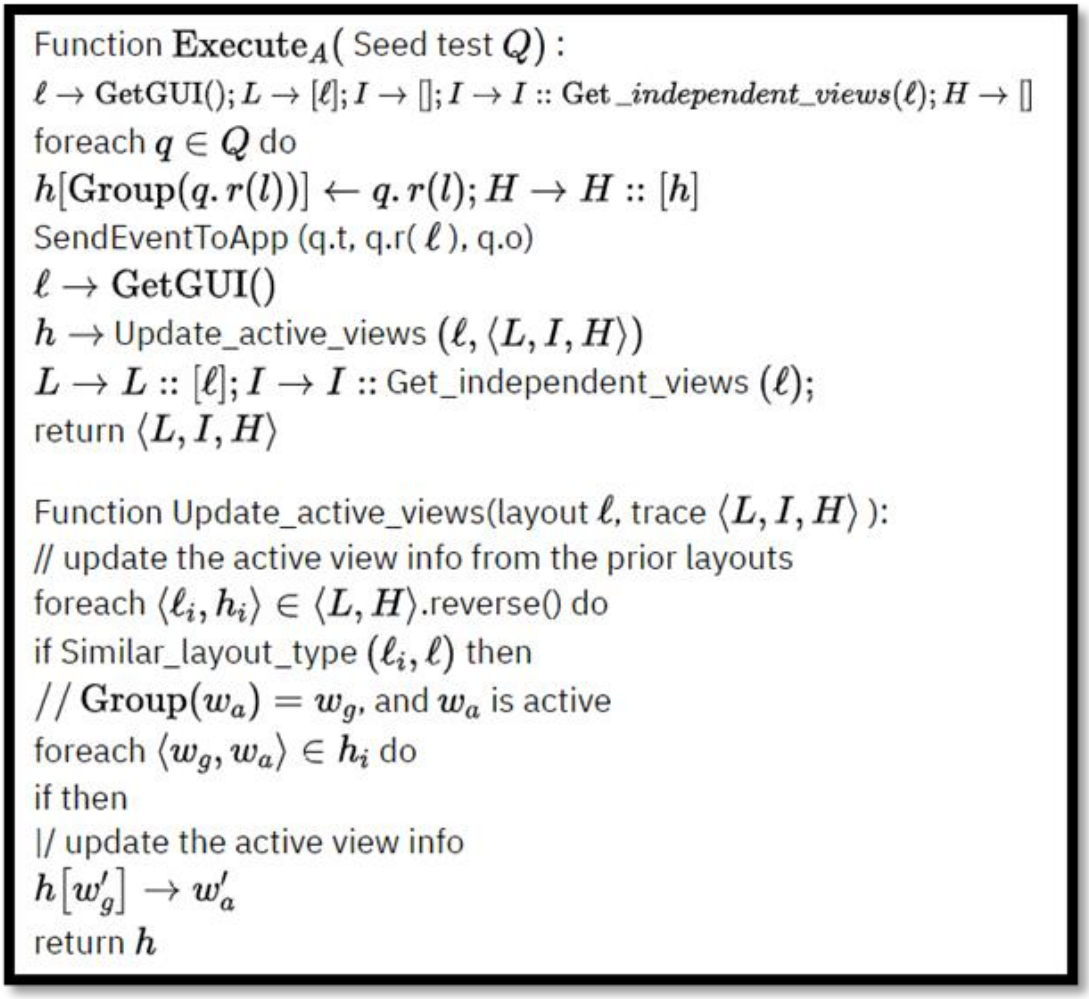

Figure 3. Algorithm

The execution trace for Q is denoted as A (Q) ΠA (Q) = ⟨L, I, H⟩; where L is the sequence of layouts, I and H record the independent views and the active view information for each layout. The receiver view of q is marked as active by the algorithm if it is contained within a group view. It then notifies the application via event q to proceed to a new layout (Lines 5-6). Subsequently, the active view information from previous GUI layouts is updated to reflect the new layout (Line 7). The internal active view information is updated by the function Update_active_views using the most recent comparable GUI layout to the current one (lines 11-16). Similar_layout_type (Line 12) is a rudimentary variant of Equivalent with; it determines whether two layouts are of the same type (i.e., whether they represent the same functional page).[4]. The term "Android-related error" refers to errors that are caused by a lack of comprehension or improper enforcement of Android-specific features. 165 out of 401 functional flaws are identified as Android-related errors.

```
1 + override fun onSaveInstanceState(outState: Bundle) {
2 + super.onSaveInstanceState(outState)
3 + outState.putString(ACCOUNT, accountUuid)
4 + outState.putLongIfPresent(FOLDER, folderId)
5 + outState.putString(FOLDERNAME, folderDisplayName)
6 + }
7 +
8 + private fun restoreInstanceState(savedInstanceState: Bundle) {9 + val accountUuid =
savedInstanceState.getString(ACCOUNT)
10 + if (accountUuid != null) {
11 + val folderId = savedInstanceState.getLongOrNull(FOLDER)
12 + val folderDisplayName= savedInstanceState.getString(FOLDERNAME)
```

Figure 2: Android-related error1 in K 9 Mail

Functional defects may result from improper management of the lifecycle of Android components (e.g., Activity, Fragment) in response to particular events (e.g., screen rotation, app background, resumption, system termination). This flaw caused the loss of the user account when the screen was rotated. The failure to save and restore the critical variables (accountUuid, folderId, folderDisplayName) that contain the account information is the cause.

```
1 private onTouchEvent(MotionEvent event) {
2 ...
3 HiTestResult hr = getHitTestResult();
4 - return onImageClick();
5 + if (isValidInstagramImageClick( )){
6 + return super.onTouchEvent(event);
7 + else {
8 + return onImageClick();
9 + }
```

Figure 4 Android-related error2 in WordPress

Overriding event callbacks is the method by which Android handles user or system events. Functional flaws may result, however, from improper handling. A malfunction occurs in which the application displays an error message "Unable to view image" when a user clicks on a thumbnail of an Instagram image. This is because the default photo viewer is incapable of previewing the image.

Oracles are frequently used in automated testing to compare the actual to the expected outcomes of test cases. To ascertain the success or failure of a test case according to predetermined criteria, automated testing tools depend on oracles. Within the context of automated testing, an "app feature agnostic oracle" denotes a component capable of validating the integrity of the software's operation without necessitating expertise regarding the features under examination. We can guarantee that the testing procedure is by the functionality under scrutiny by incorporating feature-related oracles into automated testing. This methodology aids in the detection and resolution of concerns pertaining to specific functionalities, thereby augmenting the application's overall dependability and excellence.

Table 1: Distribution of Bugs in terms of Oracle types

App	#Bugs	AFA	AFR
WordPress	68	19	49
Anki Droid	82	22	60
Firefox	43	11	32
Amaze	30	14	16
New Pipe	65	22	43
Simple Note	35	16	19
Antenna Pod	41	15	26
K 9 Mail	37	14	23
Total	401	133	268

As shown in the table, 33.3% (133/399) of functional defects are susceptible to be detected by an oracle that is independent of app features, whereas 68.7% (268/399) of the flaws necessitate an oracle that is related to app features.

When considering the application of transfer learning to identify non-crashing functional defects in Android applications, the following hypotheses may be formulated: Null Hypothesis (H0): The efficacy of flaw detection does not exhibit any discernible distinction when comparing the transfer learning approach to conventional methods. Alternative Hypothesis (H1): The efficacy of the transfer learning approach in identifying non-crashing functional flaws within Android applications surpasses that of currently employed methods.

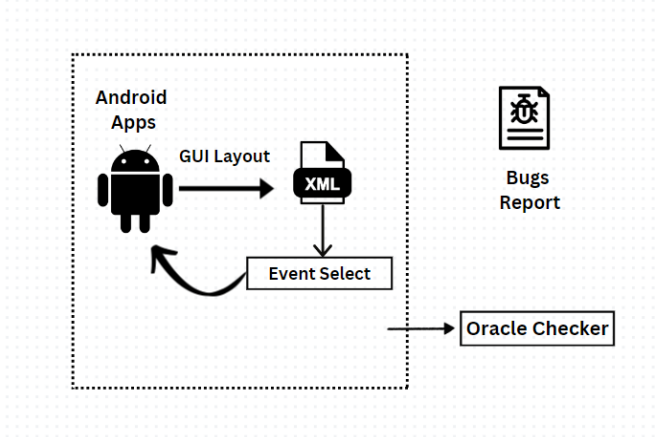

Figure 3: How RegDroid Works

RegDroid, an automated differential testing application, was utilized to identify functional flaws. RegDroid generates random graphical user interface (GUI) tests at a high level. These tests are executed independently on two app versions, denoted as $\boldsymbol{M}$ and $\boldsymbol{T}$, to determine whether the GUI pages of both versions feature comparable UI elements. RegDroid specifically operates two identical Android devices concurrently to enhance the efficacy of testing. Two essential modules are incorporated: the event selector and the Oracle validator. It is the responsibility of the event selector to generate random GUI experiments. The event generator produces the corresponding action (e.g., click, edit, scroll) for the executable GUI widget q that is selected at random from the current GUI page o of application version $\boldsymbol{M}$. The action generated is based on the widget property of q (e.g., clickable, editable, scrollable). This event i will subsequently be carried out on versions a and $\boldsymbol{V}$, correspondingly. Oracle Checker will be invoked before the implementation of the event. Iteratively, this module will be invoked until it either produces n events or inconsistency is discovered (in which case, RegDroid will relaunch the application and retest it).

Figure 4: Bar chart.

The Android resource-related error (42%), Android compatibility issues (20%); lacking cases (16%), and missing features (22%), account for the majority (78%) of UI distortion.

Table 2. Detailed stats on functional fuzzing.

App	#T. P	#Distinct Non-Crash Bugs	#Distinct Crash Bugs
WordPress	37	8	3
Anki Droid	36	6	0
Firefox	34	4	4
Amaze	35	4	6
New Pipe	33	6	7
Simple Note	32	2	1
Antenna Pod	31	4	4
K 9 Mail	22	1	2

The number #T. P. Mutants show how many true positives there are, or real working bugs. Different Non-Crash Bugs and Different Crash Bugs show how many separate bugs don't crash and crash.

Table 3. Details of False Positives

App	#F.P	#I	#L	#S
WordPress	38	35	11	27
Anki Droid	39	55	9	19
Firefox	66	8	0	28
Amaze	56	16	15	0
New Pipe	43	9	7	2
Simple Note	49	0	17	35
Antenna Pod	41	16	12	0
K 9 Mail	40	34	5	15

The full FP study results are shown in the table. The total number of FP is shown by #F. P., which is found by adding #1-O. D. E.(refined) and #T. P. More specifically, we found three groups of FPs. RegDroid was able to find 16 different bugs that worked. Two of these are new bugs that we found. 10 had been found before. It found 401 bugs, according to RegDroid. One hundred and forty-one of these bugs are true positives (35%), and two hundred and sixty are false positives (67%). There are many similar fake findings.

The consequences of false positives detected by RegDroid on software quality and development can be substantial. In the first place, they can result in the squandering of resources, as developers must invest time in identifying and resolving issues that are not true defects. Delays in other development duties and increased project costs may ensue consequently. Additionally, developers may experience frustration due to false positives, which may disrupt their productivity and have a negative effect on morale. This may result in decreased productivity and inefficiency among the development team. Moreover, a high frequency of false positives may undermine confidence in the flaw detection capabilities of RegDroid, resulting in hesitancy towards utilizing the tool and a consequent decline in its efficacy. Furthermore, the occurrence of false positives may have a detrimental effect on the quality of software as they may cause a diversion of focus from authentic problems, which may ultimately lead to the distribution of software that still contains unresolved defects. This may result in a negative user experience and reputational harm to the software. In general, RegDroid must address false positives to preserve its efficacy as a tool for detecting bugs and guarantee the quality of software in Android applications.

Even though there are some false positives, our simple version (with improvement) of RegDroid already shows that it can find hard-to-find functional bugs, which is useful because it adds to what other tools do.[18] Its false positive rate of 67% is about the same as that of the most advanced and cutting-edge functional testing tools on the market, DroidBot, and Odin, which have rates of 59% and 68%, respectively. Some things could make our study less accurate. To begin, our study got 401 working bugs from 8 apps. What we found might not be true for all apps. To lessen this risk, these apps are carefully chosen to make sure that they are diverse. These eight apps are well-known, regularly updated, and offer a range of features.

Second, the false positive research is done by hand, which means it might not be perfect.

Leverage a blend of manual and automated testing methodologies to identify and resolve functional defects that do not cause system crashes more efficiently. While automated testing increases coverage and efficacy, manual testing can help detect issues that automated tools may overlook.

Perform routine code reviews to detect and resolve potential functional flaws that do not cause crashes. Code evaluations play a crucial role in detecting potential defects by identifying inefficiencies, logical errors, and improper error management, among other issues.

Implement a resilient bug monitoring and management system to monitor and rank functional defects that do not cause crashes. This can aid in preventing issues from going unnoticed and ensuring that problems are resolved promptly.

Effective Recommendations for Guiding Better

Testing Techniques with an Attempt to Lower the Bug Infections Rate. Words to consider: Hence, Effective, Aim, killing. Besides the key parts of the discussion, we have to keep in mind the operational part which requires offering practical recommendations to be implemented on the testing side and for the functional bugs elimination. The

following actionable suggestions are proposed: The following actionable suggestions are proposed:

Having Automated and Manual Testing is Combined. Leverage an integrated selection combining automated and manual testing techniques, for more reliable inspections. Automated testing as a technique can expand test coverage and efficiency. However, with manual testing, human intuition can correctly identify subtle issues that computer-aided testing tools could not notice. \r\n Therefore, there is a synergy functioning and the evaluation of fundamental aspects is made complete Routine Code Reviews Hold code reviews regularly to be 1 step ahead in finding and fixing functional specifications that may not result in system crashes. Code evaluation is to be considered one of the highly responsible tasks for a developer to identify the bugs in the code and miss out on logical errors and inadequate error management.

Implement Robust Bug Monitoring

Build a resilient bug tracking and assistance machine that staff will watch stringently to prioritize technical problems that do not lead to system crashes. It reduces the chance of dealing with problems after they have slipped through the cracks so that it becomes possible to react as soon as something is discovered, and success is guaranteed.

CONCLUSION

We investigated how well fully automatic functional fuzzing can find logic bugs in Android apps that don't cause crashes in this thesis. The point of our study was to find ways to improve on current methods, like human testing and automatic crash-only fuzzing, by focusing on finding logic bugs that don't cause crashes right away but can still cause major security holes and bad behavior. We looked at bug reports from popular open-source Android apps and found that functional bugs are common. They can have a big effect on how well an app works and how the user feels about it. The most common types of functional bugs we found were UI problems like crashes, freezes, and acting in the wrong way. We also looked into what causes functional bugs and found that they are often caused by things like not implementing app features correctly or completely, not handling errors well, and not testing enough. From these results, it seems that better testing and development methods are needed to stop and fix useful bugs in Android apps. In summary, our research makes a valuable contribution to the progression of automated flaw detection for Android applications. We have demonstrated that functional fuzzing can be an effective method for detecting critical vulnerabilities that might otherwise remain undetected by concentrating on non-crashing logic flaws.

REFERENCES

[1] Y. Xiong et al., “An Empirical Study of Functional Bugs in Android Apps,” Int. Symp. Softw. Test. Anal., pp. 1319–1331, Jul. 2023, doi: 10.1145/3597926.3598138.

[2] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, “Fuzzing vulnerability discovery techniques: Survey, challenges, and future directions,” Comput. Secur., vol. 120, p. 102813, Sep. 2022, doi: 10.1016/J.COSE.2022.102813.

[3] Y. Rong, C. Zhang, J. Liu, and H. Chen, “Valkyrie: Improving fuzzing performance through deterministic techniques,” J. Syst. Softw., p. 111886, Nov. 2023, doi: 10.1016/J.JSS.2023.111886.

[4] T. Su et al., “Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs,” Proc. ACM Program. Lang., vol. 5, no. OOPSLA, 2021, doi: 10.1145/3485533.

[5] C. Tao, Y. Tao, H. Guo, Z. Huang, and X. Sun, “DLRegion: Coverage-guided fuzz testing of deep neural networks with region-based neuron selection strategies,” Inf. Softw. Technol., vol. 162, p. 107266, Oct. 2023, doi: 10.1016/J.INFSOF.2023.107266.

[6] B. A. Myers, “User Interface Software Tools,” ACM Trans. Comput. Interact., vol. 2, no. 1, pp. 64–103, Jan. 1995, doi: 10.1145/200968.200971.

[7] T. Su, J. Wang, and Z. Su, “Benchmarking automated GUI testing for Android against real-world bugs,” ESEC/FSE 2021 - Proc. 29th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., pp. 119–130, Aug. 2021, doi: 10.1145/3468264.3468620.

[8] E. Ma et al., “Automata-Based Trace Analysis for Aiding Diagnosing GUI Testing Tools for Android,” ESEC/FSE 2023 - Proc. 31st ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., pp. 592–604, Nov. 2023, doi: 10.1145/3611643.3616361.

[9] Y. Zhou and W. Song, “DDLDroid: Efficiently Detecting Data Loss Issues in Android Apps,” ISSTA 2023 - Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal., pp. 703–714, Jul. 2023, doi: 10.1145/3597926.3598089.

[10] Y. Wang, P. Jia, L. Liu, and J. Liu, “A systematic review of fuzzing based on machine learning techniques”.

[11] F. Rustamov, J. Kim, J. Yu, and J. Yun, “Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection,” IEEE Access, vol. 9, pp. 131166–131190, 2021, doi: 10.1109/ACCESS.2021.3114202.

[12] C. Poncelet, K. Sagonas, and N. Tsiftes, So Many Fuzzers, so Little Time: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack, vol. 1, no. 1. Association for Computing Machinery, 2022. doi: 10.1145/3551349.3556946.
[13] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing Automatic Differentiation in Deep-Learning Libraries," Proc. - Int. Conf. Softw. Eng., pp. 1174–1186, 2023, doi: 10.1109/ICSE48619.2023.00105.
[14] A. Alsaedi, A. Alhuzali, and O. Bamasag, "Effective and scalable black-box fuzzing approach for modern web applications," J. King Saud Univ. - Comput. Inf. Sci., vol. 34, no. 10, pp. 10068–10078, 2022, doi: 10.1016/j.jksuci.2022.10.006.
[15] J. Sun et al., "Understanding and finding system setting-related defects in Android apps," ISSTA 2021 - Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal., pp. 204–215, Jul. 2021, doi: 10.1145/3460319.3464806.
[16] G. R. Mattiello and A. T. Endo, "Model-based testing leveraged for automated web tests," Softw. Qual. J., vol. 30, no. 3, pp. 621–649, Sep. 2022, doi: 10.1007/S11219-021-09575-W.
[17] J. W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," Proc. - 2019 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2019, pp. 42–53, Nov. 2019, doi: 10.1109/ASE.2019.00015.
[18] Q. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway, "Redroid: A Regression Test Selection Approach for Android Applications", doi: 10.18293/SEKE2016-223.