LazyCow: A Lightweight Crowdsourced Testing Tool for Taming Android Fragmentation

Xiaoyu Sun*
Xiaoyu.Sun.IEEE@gmail.com
Australian National University
Australia, Canberra, ACT

Xiao Chen
Xiao.chen@monash.edu
Monash University
Australia, Clayton, VIC

Yonghui Liu
yonghui.liu@monash.edu
Monash University
Australia, Clayton, VIC

John Grundy
john.grundy@monash.edu
Monash University
Australia, Clayton, VIC

Li Li
lilicoding@ieee.org
Beihang University
China, Beijing, Beijing

ABSTRACT

Android fragmentation refers to the increasing variety of Android devices and operating system versions. Their number make it impossible to test an app on every supported device, resulting in many device compatibility issues and leading to poor user experiences. To mitigate this, a number of works that automatically detect compatibility issues have been proposed. However, current state-of-the-art techniques can only be used to detect specific types of compatibility issues (i.e., compatibility issues caused by API signature evolution), i.e., many other essential categories of compatibility issues are still unknown. For instance, customised OS versions on real devices and semantic OS modifications could result in severe compatibility issues that are difficult to detect statically. In order to adress this research gap and facilitate the prospect of taming Android fragmentation through crowdsourced efforts, we propose LazyCow, a novel, lightweight, crowdsourced testing tool. Our experimental results involving thousands of test cases on real Android devices demonstrate that LazyCow is effective at autonomously identifying and validating API-induced compatibility issues. The source code of both client side[1] and server side [2] are all made publicly available in our artifact package. A demo video of our tool is available at https://www.youtube.com/watch?v=_xzWv_mo5xQ.

ACM Reference Format:
Xiaoyu Sun, Xiao Chen, Yonghui Liu, John Grundy, and Li Li. 2023. LazyCow: A Lightweight Crowdsourced Testing Tool for Taming Android Fragmentation . In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Xiaoyu Sun is the corresponding author.
[1]https://github.com/sunxiaobiu/LazyCow
[2]https://github.com/sunxiaobiu/RemoteTest

1 INTRODUCTION

Android fragmentation has long caused compatibility issues that may crash applications on users' Android devices and lead to bad user experiences. There are many Android OS versions and smartphone manufacturer-customized ROMs on the market. Android app developers struggle to test their applications across many different types of devices due to this *Android fragmentation.* Cai et al. [8] have experimentally showed that Android device variety is one of the main causes of incompatibility. This can severely impede the productivity of app developers, who are required to test their apps on a wide range of devices to ensure that no compatibility issues will arise. In theory, developers should gather devices with different specifications, including brands, models, SDK versions, and software/hardware configurations. However, it is not feasible for developers to have a complete set of devices that cover all possible specifications. Moreover, integrating the incompatibility testing process into the developers' daily workflow can be time-consuming. Hence, there is a pressing need to address the Android fragmentation through a lightweight, crowdsourced approach.

Most current state-of-the-art methods detect compatibility issues through static analysis techniques, as demonstrated in prior works such as Ham et al.[10], Huang et al.[11], Li et al.[13], Wei et al.[18], and Zhang et al.[21]. However, such approaches are only effective in detecting certain types of compatibility issues[14], specifically those caused by syntactic changes, leaving other more complex types of issues uncovered. For example, Sun et al. [17] have shown that CiD is unable to handle compatibility issues triggered by semantic changes. Additionally, customization of the Android OS can introduce compatibility issues that are difficult to be detected by static analysis techniques. To address this problem, we propose a lightweight crowdsourced platform to automatically distribute tests across real-world devices to detect a wider range of compatibility issues, taking advantage of dynamic testing.

In this work, we demonstrate a novel, lightweight, crowdsourced testing framework, LazyCow, that automatically distributes and executes test cases on real-world devices to trigger compatibility issues dynamically. Unlike traditional approaches that dispatch executable Android apps, LazyCow directly dispatches and executes test cases on the real-world devices. This approach is "lightweight" and provides several advantages such as reducing bandwidth, diminishing user awareness, allowing flexibility, and guaranteeing full test case execution. We evaluated LazyCow on thousands of test cases, successfully detecting 393 APIs with compatibility issues. Manual validation confirmed a 100% true positive rate, with 109 Signature-based issues and 284 Semantics-based issues that cannot be noticed by state-of-the-art static methods. Furthermore, we identified 161 vendor-specific and 47 model-specific compatibility issues, which are introduced when smartphone vendors customize the Android system and may result in severe security problems.

2 MOTIVATION

Crowdsourced testing has been a hot research topic for many years, with various studies conducted on its application to Android app testing [9, 15, 16]. Several industry leaders, including Global App Testing [5], Digivante [4], test IO [7], and QA Mentor [6], offer crowdsourced testing services that allow users (e.g., app developers) to test their mobile apps with thousands of professional testers from around the world. However, these all need crowd workers, which can make them time-consuming, prone to errors, and unable to automatically detect Android compatibility issues without human intervention. Moreover, users cannot customize test scripts to their particular needs, leading to undetected compatibility issues.

From the academic perspective, crowdsourced app testing has also been on the rise, with several studies exploring novel approaches for achieving better results. For instance, Wu et al.[20] proposed a method of recording user interactions and replaying them through crowdsourced testing services to identify bugs. However, this technique can be time-consuming, given that real user interactions with apps are involved. Li et al.[12] developed CoCoTest, a crowdsourced testing platform that leverages collective intelligence to recommend bug reports to workers. Unfortunately, this approach can be ineffective since crowd workers may submit low-quality reports.

Current crowdsourced testing platforms involve human intervention, resulting in different levels of professionalism, making them error-prone and time-consuming. In addition, the standard approach to crowdsourced testing is to test the entire Android app on crowdsourced devices, which can miss some app code. This limitation highlights the need for exploring the possibility of distributing test cases, which are directly executable code snippets, to real-world Android devices. To fill this research gap, we propose a novel platform that automatically generates and distributes tests to real-world devices without human intervention to detect Android compatibility issues.

3 OUR APPROACH

Our main objective is to provide a lightweight crowdsourced testing platform for automatically executing unit tests on real-world Android devices. To achieve this, we have designed and developed a prototype tool called LazyCow, which works on a client-server model. Figure 1 depicts the architecture of LazyCow, where the client is installed on multiple Android devices to handle the execution of test cases. The client determines the number and time of test cases to be executed and then sends the results back to the server for further analysis. On the other hand, the server is responsible for collecting, packaging, and dispatching test cases to the clients, and analyzing compatibility issues based on the results obtained from different devices. We provide a detailed explanation of each component in the following subsections.

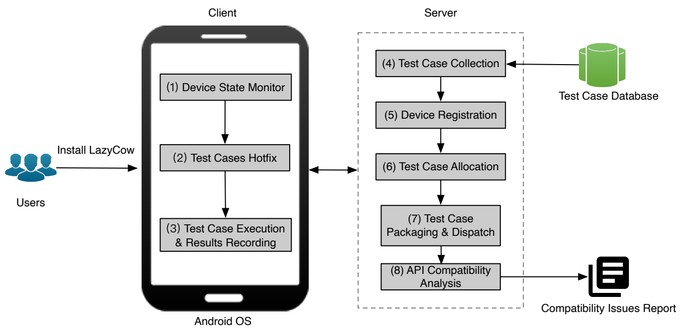


Figure 1: The working process of our LazyCow approach.

3.1 Client Side

We created a client application for LazyCow that can be installed on Android devices. The client app continuously monitors the status of the device to determine the appropriate time to run the test cases (e.g., when the device is not in use). It then communicates with the server to download and execute the test cases, and sends back the execution results for further analysis. Figure 1 shows the three modules included in the client: *(1) Device State Monitor*, *(2) Test Cases Hotfix*, and *(3) Test Cases Execution & Results Recording*.

(1) Device State Monitor. To avoid disrupting the user experience, LazyCow detects the state of the devices to identify an appropriate time to execute the test cases. We define a suitable time as a moment that meets the following three conditions:

i **Phone State:** To identify whether the user is interacting with the device, we utilize the methods *android.os.PowerManager#isDeviceIdleMode* and *android.os.PowerManager#isScreenOn*. The suitable time for running the test cases is defined as the time when the user is not interacting with the device.

ii **Memory Usage:** We use *android.app.ActivityManager#getMemoryInfo* to obtain the device's memory usage. A suitable time is identified if the memory usage is below 25%.

iii **Battery State:** We utilize *android.os.BatteryManager* to check the battery state, examining if it is charging and has enough battery life (above 60%).

After identifying a suitable time, the client would send a request to the server to download the test cases for testing.

(2) Test Cases Hotfix. To dynamically dispatch incremental tests on Android devices without requiring app reinstallation, we utilize a hotfix technique that supports updating classes, files, and resources with minimal impact on the user experience. To achieve this, we integrate LazyCow with *Tinker* [3], a hotfix solution that supports updating classes, libraries, and resources without requiring APK reinstallation when downloading test cases from the server. This approach minimizes the impact on users when updating incremental test cases. Tinker's repair principle is based on class loading and it supports the addition and replacement of classes and resources. Figure 2 illustrates Tinker's repair principle, which is based on the DEX subpackage scheme and the principle of multiple DEX loading. After comparing the differences between the new and base APKs, updated classes and resources are merged into a *patch.dex* file. The *patch.dex* file is then combined with the applied classes.dex, replacing the old DEX file to complete the hotfix process.

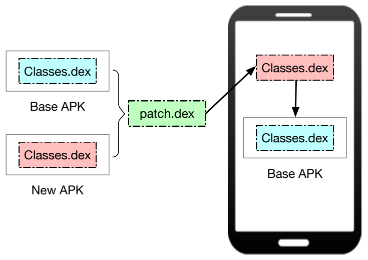


Figure 2: The repair principle of Tinker.

To implement the hotfix process, we adopt the concept of replacing the new DEX with the full amount of instant runs. To achieve this, we package the test cases in the DEX files of an APK. During the hotfix process, we calculated the differences between the old and new test cases and placed them in a patch package. This patch package was then synthesized and delivered to the device for hotfix. The incremental test cases in the patch package were placed in the directory below Tinker. Using Tinker's Classloader, the new test cases in the patch package could be loaded based on the hotfix principle.

(3) Test Cases Execution & Results Recording. After downloading the test cases to client devices, LazyCow uses *reflection calls* to retrieve and sequentially execute them from the DEX files. The test cases are written in the format of Java Unit tests, and LazyCow automatically runs them based on the annotations of each test method.

The JUnit [1] framework is the most widely used unit testing framework in Java, with five annotations for test execution callbacks: *@BeforeClass*, *@Before*, *@Test*, *@After*, and *@AfterClass*. Test methods are annotated by the *@Test* annotation, and LazyCow also supports constraining the execution flow of specific methods with the *@Before* (or *@After*) and *@BeforeClass* (or *@AfterClass*) annotations. LazyCow first performs static analysis to resolve annotations from each method and then uses reflection calls to invoke methods in the sequence of *@BeforeClass* → *@Before* → *@Test* → *@After* → *@AfterClass*.

After executing the test cases, LazyCow handles any exceptions that may occur using a *try-catch* block. It collects execution results whenever a test case fails or succeeds, along with relevant information (e.g., the stack trace information when a test fails), and sends it back to the server for further analysis.

3.2 Server Side

The test case database on the server is collected from multiple sources such as the AOSP codebase [2] and Github app code repositories. These test cases are then packaged and dispatched to registered clients in a load-balanced manner. After the clients execute the test cases, the server gathers the outputs for further analysis to identify potential compatibility issues. The server-side modules, as shown in Figure 1, include *(4) Test Case Collection*, *(5) Device Registration*, *(6) Test Case Allocation*, *(7) Test Case Packaging & Dispatch*, and *(8) API Compatibility Analysis*.

(4) Test Case Collection. The server constantly updates and manages a test case database for testing on client devices. Three sources of test cases can be collected, including:

- Test cases that are already included in the Android Open Source Project (AOSP) codebase [2], authored by Android OS developers.
- Test cases generated by automatic test case generation tools such as JUnitTestGen [17].
- Users of LazyCow can write customized test cases to fulfill their specific requirements. For instance, in continuous integration during app development, developers may want to verify whether certain APIs create compatibility issues on specific Android devices.

(5) Device Registration. Upon installation, the LazyCow client app registers the client device with the server. The registration process involves collecting device information such as the device's manufacturer and model, SDK version, device language, and screen size, which will be utilized to optimize the distribution of test cases. It is important to note that the LazyCow app does not collect any personal private data, such as device IDs, but rather assigns a unique ID to each device for identification purposes.

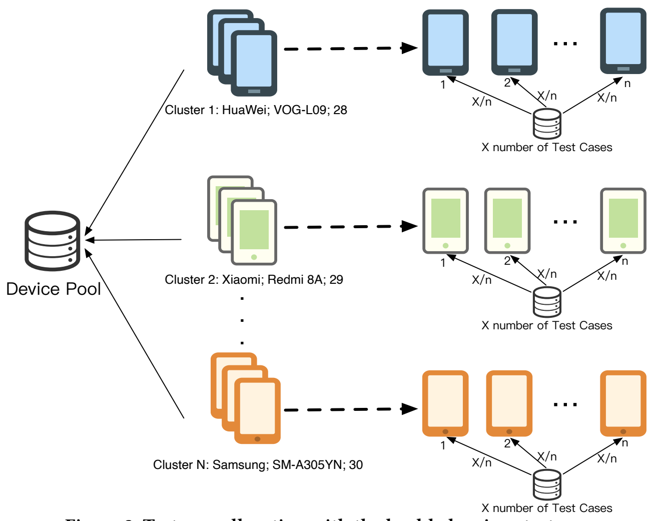

Device Pool
Cluster 1: HuaWei; VOG-L09; 28
Cluster 2: Xiaomi; Redmi 8A; 29
Cluster N: Samsung; SM-A305YN; 30
1
2
n
X/n
X/n
X/n
X number of Test Cases
X number of Test Cases
X number of Test Cases

Figure 3: Test case allocation with the load-balancing strategy.

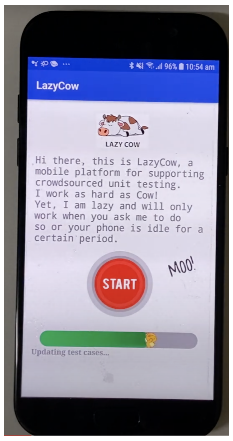
(6) Test Case Allocation. We design a test case allocation algorithm to distribute test cases evenly among all registered devices. This algorithm ensures load-balancing and is illustrated in Figure 3. Initially, LazyCow groups registered devices into clusters based on their device information such as manufacturer, model, and Android SDK version. For example, devices from Huawei with model VOG-L09 and API level 28 are grouped into cluster 1. Each cluster comprises devices with identical specifications. Then, test cases are distributed evenly to all devices within each cluster. This approach guarantees that each test case runs on multiple devices with different specifications and avoids redundant executions on devices with identical specifications, except for the cases where explicitly specified.

(7) Test Case Packaging & Dispatch. After assigning the test cases to each client, LazyCow packages and dispatches them to their respective clients. To apply the code changes (i.e., assigned test cases) to the client without reinstalling the LazyCow app, LazyCow integrates the hot-swap technique [19]. To achieve this, LazyCow monitors changes in files (i.e., test cases) and runs a custom Gradle task that generates .dex files for the modified classes only. Next, another Gradle command is used to package the newly generated .dex files into an APK and send it back to the client. The LazyCow client then reloads these newly assigned test classes and invokes them using reflection calls.

(8) API Compatibility Analysis. Once the test cases have run on different Android devices, LazyCow retrieves and stores the execution results in a database. The results contain information about the success or failure of each test case on the device, including any relevant exception information or error messages (e.g., Assertion error message) if applicable.

The API Compatibility Analysis module then examines the results across all devices to detect API-related compatibility issues. An Android API is considered to have compatibility issues if its execution results are inconsistent across different Android devices. Specifically, LazyCow identifies a compatibility issue with a given Android API if any of the following criteria are met: (1) A test case fails on certain devices but runs successfully on others, or (2) The test case throws different errors or exceptions on different device configurations (e.g., *NoSuchMethodError* on some versions and *SecurityException* on others). Based on the comparative analysis results, LazyCow flags vendor-specific, model-specific, and Android version-specific compatibility issues for Android APIs. These have been long-standing challenges that existing approaches, such as CiD, FicFinder, etc., have not yet addressed specifically for detecting compatibility issues in Android devices.

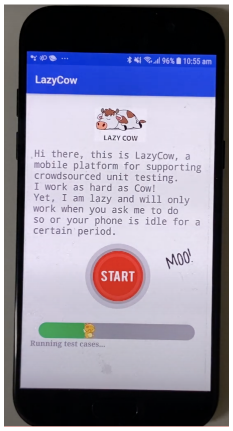
To further elaborate the working process of LazyCow, we present the screenshots of test case allocation and execution process in Figure 4.



Hi there, this is LazyCow, a mobile platform for supporting crowdsourced unit testing.
I work as hard as Cow!
Yet, I am lazy and will only work when you ask me to do so or your phone is idle for a certain period.

Updating test cases...

(a) The UI Page of Test Case Allocation.



Hi there, this is LazyCow, a mobile platform for supporting crowdsourced unit testing.
I work as hard as Cow!
Yet, I am lazy and will only work when you ask me to do so or your phone is idle for a certain period.

Running test cases...

(b) The UI Page of Test Case Execution.

Figure 4: The UI Pages of Test Case Allocation and Execution.

4 EVALUATION

4.1 Experimental settings

We investigated the effectiveness of detecting compatibility issues in Android devices with LazyCow. We used 11 Android smartphones from various manufacturers with different Android OS versions. These devices were obtained from real-world users who downloaded and installed LazyCow through online advertising. We recruited 11 participants, contributing 11 Android devices. Also, we prepare test cases dataset that contains 5,401 test cases (covering 5,401 unique Android APIs). To examine LazyCow's efficiency of dispatching and executing unit tests, we install LazyCow on all devices and record the number of test cases successfully executed and the execution time for each run to evaluate LazyCow's performance.

4.2 Results - The effectiveness of LazyCow in discovering Compatibility Issues.

After analyzing and comparing the execution results of test cases, LazyCow identified 393 Android APIs that may have compatibility issues. We further discover that among the 393 identified compatibility issues, 109 of them belong to signature-based issues and 284 are semantic-based issues. In addition, we find that LazyCow is able to detect 161 APIs with vendor-specific compatibility issues and 47 model-specific compatibility issues. Our approach has been proven effective in automatically identifying and confirming compatibility issues caused by APIs, not only based on their signature but also on their semantics, surpassing the current state-of-the-art techniques.

4.3 Results - The comparison of LazyCow with existing tools.

Comparison with JUnitTestGen. LazyCow outperforms JUnitTestGen in detecting compatibility issues. The reason for this difference is that JUnitTestGen only tests emulators that use the original Android OS, which overlooks many compatibility issues caused by vendor/model customization. This finding demonstrates that LazyCow can identify a wider range of compatibility issues than existing dynamic approaches and is promising in complementing these approaches.

Comparison with Google CTS. LazyCow outperforms Google CTS in detecting more compatibility issues. One major reason for CTS's failure to detect compatibility issues, particularly those caused by vendor/model customization, is the lack of sufficient testing context, such as various parameter values. In contrast, LazyCow can detect such compatibility issues because it relies on JUnitTestGen to mine existing Android API usages and generates API-focused test cases that retain the execution context in real-world applications.

5 CONCLUSION

We have introduced LazyCow, a novel, lightweight prototype tool that uses crowdsourced testing techniques to identify compatibility issues caused by Android fragmentation. Our experimental results indicate that: (1) LazyCow is capable of automatically executing test cases on real-world Android devices; (2) Our approach is effective in automatically detecting and confirming API-induced compatibility issues.

REFERENCES

[1] 2021. *JUnit.* https://en.wikipedia.org/wiki/JUnit#:~:text=JUnit%20is%20a%20unit%20testing,xUnit%20that%20originated%20with%20SUnit.&text=junit%20and%20junit.

[2] 2021. *Source Code of the Android Open Source Project.* https://cs.android.com/android

[3] 2021. *Tinker.* https://github.com/Tencent/tinker

[4] 2022. *Digivante.* https://www.digivante.com/crowdsourced-testing-referral/?utm_campaign=SoftwareTestingHelp%20Referral%20Campaigns&utm_source=software-testing-help&utm_content=crowdtesting

[5] 2022. *Global App Testing.* https://go.globalapptesting.com/app-testing-for-engineering-qa

[6] 2022. *QA Mentor.* https://www.qamentor.com/qa-services/crowdsourced-testing-services/

[7] 2022. *test IO.* https://goo.gl/rGQPWF

[8] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 216–227.

[9] Sebastian Elbaum and Madeline Hardojo. 2004. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis.* 65–75.

[10] Hyung Kil Ham and Young Bom Park. 2011. Mobile application compatibility test system design for android fragmentation. In *International Conference on Advanced Software Engineering and Its Applications.* Springer, 314–320.

[11] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 532–542.

[12] Haoyu Li, Chunrong Fang, Zhibin Wei, and Zhenyu Chen. 2019. CoCoTest: collaborative crowdsourced testing for Android applications. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 390–393.

[13] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 153–163.

[14] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and characterizing silently-evolved methods in the android API. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE, 308–317.

[15] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas Schmidt, and Balachandran Natarajan. 2004. Skoll: Distributed continuous quality assurance. In *Proceedings. 26th International Conference on Software Engineering.* IEEE, 459–468.

[16] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. 2002. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis.* 65–69.

[17] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. 2022. Mining android api usage to generate unit test cases for pinpointing compatibility issues. In *37th IEEE/ACM International Conference on Automated Software Engineering.* 1–13.

[18] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* 226–237.

[19] wikipedia. [n. d.]. Hot swapping. https://en.wikipedia.org/wiki/Hot_swapping. Online; accessed 28 January 2022.

[20] Guoquan Wu, Yuzhong Cao, Wei Chen, Jun Wei, Hua Zhong, and Tao Huang. 2017. AppCheck: a crowdsourced testing service for android applications. In *2017 IEEE International Conference on Web Services (ICWS).* IEEE, 253–260.

[21] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. 2015. Compatibility testing service for mobile applications. In *2015 IEEE Symposium on Service-Oriented System Engineering.* IEEE, 179–186.