Behind Defective Mobile AR Apps: Studying Reviews and Bugs of Android AR Software with Comparison to Prior Bug Studies

ANONYMOUS AUTHOR(S)

As Augmented Reality (AR) applications grow in popularity, understanding and addressing AR software bugs becomes crucial. AR applications, due to their interaction with the physical world, present unique challenges that differ from traditional software. In this study, we try to understand the root causes of commonly complained mobile AR bugs. In our study, we collected user reviews from Google Play market, and issue reports from open source AR software projects from GitHub. We categorized bug symptoms and root causes, and studied the correlation between them. We further studied the fixing commits of these bugs and compare their distribution with findings from prior bug studies. Our study finds that (1) AR apps users are mostly affected by dysfunction bugs such as *Hang* and Crash and these bugs are common in AR apps, (2) *API Misuse* is the mostly common root cause of AR bugs, and property setting error is the most common form of *API Misuse*, (3) a small number of API patterns and event handling practices may account for a large portion of *API Misuse*, and (4) besides AR UI symptoms and the *API Misuse* root cause, bugs in AR apps have similar characters with bugs in other Android apps.

ACM Reference Format:
Anonymous Author(s). 2025. Behind Defective Mobile AR Apps: Studying Reviews and Bugs of Android AR Software with Comparison to Prior Bug Studies. 1, 1 (September 2025), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Augmented Reality (AR) technology complements the real world in the camera with a digital layer where various virtual object models can be placed. Using real-time camera feedback, data from advanced sensors such as gyroscope, depth sensor, and onboard computational power, AR Applications enrich the real-world environment through this digital layer and provide various immersive user experiences. Estimated revenue by market size of AR as of 2024 is $40 billion where 25% of that is in mobile AR [34]. Several AR functionality-specific devices such as Ray-ban Meta [24], Meta Quest [23], Microsoft HoloLens [25], Apple Vision Pro [2], and other wearable heads-up display devices, have been actively developed and launched into market which makes AR gradually more accessible. More importantly, billions of handheld smartphones are also becoming capable of AR. As of 2024, approximately 1.73 billion [33] active AR user devices are available. In the meantime, software developers have also produced many AR applications in domains ranging from education and gaming to retail and healthcare [11]. With advancements in mobile technology and wearable devices, more users are engaging with AR experiences for entertainment, training, and practical applications like navigation and shopping.

The quality of AR applications is crucial to ensuring smooth performance, realistic visuals, and intuitive interactions. Poorly designed AR experiences can lead to user frustration, motion sickness, or lack of engagement, ultimately harming adoption rates. High-quality AR applications,

ACM XXXX-XXXX/2025/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

on the other hand, enhance immersion and usability, making them more effective and enjoyable. Furthermore, since users may use AR applications to directly interact with the physical world, their defects are more likely to cause immediate consequences in the real world. For example, erroneous output in surgery-aiding AR software [28] may mislead the doctor and lead to medical accidents, and errors in mobile AR navigation software may block the driver's / pedestrian's eyesight and cause traffic accidents.

AR applications' close interaction with the physical world also leads to many new types of bugs that did not exist before in traditional software. For example, object misplacement bugs [32] occur where virtual objects are not placed at users' intended positions in the physical world. lose tracking bugs [42] occur when placed virtual objects are drifting from their original location while users move the camera. There are also object-rendering bugs where the color, reflection, or shade of virtual objects become incorrect when placed into the physical world, especially in certain lighting conditions. These new types of bugs bring unique challenges to software quality assurance. Furthermore, some of these bugs are due to the limitations of existing AR techniques and/or frameworks instead of defects in the application code.

Given the importance, obscurity, and complication of AR bugs, we believe empirical studies to understand the landscape of AR bugs will likely (1) benefit the developers by providing a guideline for them to avoid pitfalls and common mistakes in using AR frameworks and developing AR apps, (2) benefit AR product managers by summarizing how the limitations of current AR frameworks may lead to application failures that finally affect user experience so that they can make wiser design and market decisions, and (3) benefit the software engineering researchers by providing a basis for developing the best AR software engineering practices and tool support for AR bug detection and repair.

To achieve these goals, this paper presents an empirical study that investigates the user reviews of popular Android AR apps, common symptoms of documented bugs in Android AR software projects, and the bugs' root causes. We decided to have our study focus on Android AR apps for the following reasons. First, mobile phones are the most widely used AR-enabled devices, so studying bugs in mobile AR apps can provide a representative view of AR apps. Second, the Android ecosystem is generally open-source, making it possible to study the root causes of bugs in the source code. The user reviews from the Google Play store also allow us to connect symptom categories observed in the bugs with those being complained about by users. Third, bugs in GUI-based Android applications have been well studied and categorized in a recent work [41] (referred to as the *DroidBugStudy* in the rest of the paper), so focusing on Android allows us to perform a comparison study and understand how bugs in AR apps differ from bugs in traditional mobile applications. We are also aware of a prior work by Li et al. [21] (referred to as the *WebXRBugStudy* in the rest of the paper) which studied more than 300 bugs from Web-based XR software projects. So, we can further compare our study results on mobile AR bugs with their results to find commonalities and differences in bugs from different platforms.

Our study collected 2,846 closed issues from Github open-source AR projects which use Google ARCore as their AR platform. From these issues, we identified 288 bug reports that are confirmed by the developers and are directly related to AR features. We also collect 5,440 user reviews of top AR apps from Google Play Store [7], from which we identified 635 reviews that complain about app failures or bugs. It should be noted that most top AR apps are not open source, so we cannot trace reviews directly back to the corresponding bug reports, but we can still link the two datasets with symptoms and investigate the potential root causes of the most widely complained symptoms. We further analyzed the the code and change features of the code commits associated with fixed bug reports to confirm bug root causes and understand repairing patterns. Finally, we align our findings

with two earlier studies on Web AR bugs and mobile GUI bugs to summarize commonalities and differences.

To sum up, our paper mainly makes the following contributions.

- A large dataset combining user reviews of top Android AR apps and bug reports from open-source Android AR software projects in Github.
- A categorization of AR-related bug reports based on symptoms, root causes, and code repair features.
- A correlation study of user reviews of AR apps and AR bug reports based on symptoms.
- A comparison of study findings from mobile AR bugs versus Web AR bugs and mobile GUI bugs.

The rest of the paper is organized as follows. In Section 2, we list our research questions, as well as the purposes and approaches to answer these questions. In Section 3, we describe our methodology for data collection and analysis. In the following Sections 4 and 5, we present our empirical data to answer the research questions, highlight the major findings, and discuss how some important lessons can be learned from the observations of the study results. We summarize some related works in Section 6. Finally, we conclude the study in Section 7 and point to some future works.

2 Research Questions

In this study, we aim to answer the following five research questions.

- **RQ1: What types of Android AR app bugs are mostly complained in user reviews?** **RQ1** tries to understand users' perception of Android AR bugs, and find out what quality factors are affecting users' acceptance of AR apps. To answer **RQ1**, we collected AR app reviews from the Google Play Store, identified the negative reviews that complained about app bugs, and categorize the reviews by based on the symptoms of these bugs.
- **RQ2: What are the common symptoms and root causes of Android AR bug reports?** **RQ2** tries to understand developers' perception of Android AR bugs, and find out the common types of bugs they detected and fixed. To answer **RQ2**, we collected confirmed AR bug reports from open source AR projects in Github, and categorize them based on their symptoms and root causes.
- **RQ3: How symptoms are correlated to root causes in Android AR bug reports?** **RQ3** tries to understand what are the mostly likely root causes of specific bug symptoms. Since symptoms can also be linked to user reviews, we can further investigate what types of software defects are triggering most user complaints. To answer **RQ3**, we performed correlation analysis between symptoms and root causes of AR bug reports to identify strong correlations between symptoms and root causes.
- **RQ4: What are the common code repair patterns of Android AR bugs? RQ4** tries to understand the code features of AR bug repairs and what code / API methods are often involved in them. The findings may be helpful for designing detection and repair tools for AR bugs. To answer **RQ4**, we performed code analysis on code commits associated with AR bugs reports, and report the statistics on different code-related facts of the commits.
- **RQ5: Are findings from Android AR bugs similar to those from Android GUI bugs and Web AR bugs? RQ5** tries to understand the uniqueness of Android AR bugs, and their relationship with Android GUI bugs / Web AR bugs. It will help researchers better place problems and techniques related to Android AR bugs in the research domain. To answer **RQ5**, we aligned our labeled bug symptoms and root causes with those in two previous studies, and compared our statistical results and findings with theirs.

Package Name	Total Reviews	Rating	Downloads
com.tasmanic.camtoplanfree	71.0K	4	10.0M
com.grymala.arplan	89.3K	4.2	5.0M
com.grymala.aruler	84.4K	3.9	5.0M
com.sensopia.magicplan	81.4K	4.1	5.0M
io.spoton.viewer	11.6K	4.2	5.0M
com.Elementals.Bike3DConfigurator	40.3K	4	1.0M
com.ar.augment	32.1K	3.8	1.0M
com.arloopa.arloopa	27.3K	4.7	1.0M
com.blippar.ar.android	26.9K	2.4	1.0M
com.puteko.colarmix	20.0K	3.5	1.0M

Table 1. Top 10 AR Apps in Our Review Dataset

3 Methodology

Our approach to this study consists of three major steps. The first step is data collection. In this step we collect user reviews on AR applications on Google Play Store, and collect a list of issues created on open-source AR applications on GitHub. The next step is data cleaning. In this step we filter out the user reviews that are either five stars, written in non-English text, or does not contain any legible text. In this step we also filter out the GitHub issues to only keep those that are closed and does not belong to any library or framework. After data cleaning we categorize the data in the third step, where two of the authors independently labeled all the data with conflicts resolved with other authors.

3.1 Data Collection

Our data collection consists of two parts: the collection of user reviews from Google Play Store, and the collection of issue reports from the Github repositories.

Collection of User Reviews. To collect user reviews from Google Play store, we first need to identify top AR apps. In particular, we first go to the AR category of Google Play Store, and sort the apps by their number of downloads. From the sorted app list, we consider only the apps that has the words "AR", "Augmented", or "Reality" in their name or summary. We perform this filtering to identify the apps with AR as their main feature, and remove the general Apps with AR features, such as Amazon and Snapchat. We further set a criterion of having at least 1000 downloads and 10 reviews. Finally, we acquire 67 apps in the sorted list which satisfy our filtering criterion above. The package names of the 67 apps is available on our project website. For each of the 67 apps, we download up to 150 most recent reviews from each app because Google is restricting review downloading now and shows only up to 150 reviews on the web portal of Google Play. From these apps, we collected 5,440 reviews in total. We further filter out all the five-star reviews because they should not mention any defects of apps, and end up acquire a dataset of 2,588 non-perfect reviews. Table 1 presents the top 10 apps in our dataset with highest downloads.

Collection of Issue Reports. To collect issue reports from Github, we first use code search in Github to fetch all the projects whose code contains `import` statements of Google AR Core Package Signature `com.google.ar.core`, so we can tell that AR Core has been used in the project. Then, from the projects, we keep only those with closed issues, and we collect all those closed issues to form a dataset of 2,846 issues from 77 apps. Table 2 presents the top 10 software projects in our dataset with highest number of issue reports.

3.2 Data Labeling and Categorization.

Data Labeling Procedure. In our study, we need to label a lot of user reviews and issue reports to categorize them and extract their characteristics. To enhance the quality of our labeling outcomes, we use the following procedure of three steps. First, for each labeling task, two authors manually go over all the data independently and create their own labels for each data item. Second, all the labels created by both labeling authors are brought together into a discussion to align them. Other

Repo Name	Total Issue Reports	Stars
introlab/rtabmap	664	3.3K
Azure/azure-spatial-anchors-samples	350	2.1K
SceneView/sceneform-android	246	666
tukcomCD2024/DroidBlossom	210	10
spe-uob/2021-ARMessaging	178	12
gluonhq/attach	156	53
isl-org/OpenBot	148	3.1K
willowtreeapps/vocable-android	142	121
appoly/ARCore-Location	82	483
giandifra/arcore_flutter_plugin	59	446

Table 2. Top 10 Software Projects in Our Issue Report Dataset

authors also provide their opinions until all the labels are aligned, so a unified labeling system is created. Third, both labeling authors re-label the data items (i.e., translate their original labels to labels in the unified labeling system), and any unresolved conflicts are given to a third author to resolve. To measure the consistency of the labeling, we further calculate the Cohen's Kappa value for each labeling task.

Labeling of User Reviews. When labeling user reviews, we first removed all the five star reviews because they unlikely contain any descriptions of defects. From the remaining 2,588 reviews, we identified 635 reviews complaining about specific misbehavior (i.e., we removed the bad reviews with just general judgments such as "Trash app.", and reviews asking for only feature changes such as "The app should support GPS." or "The app has too many ads.").

When trying to categorize the 635 reviews by their symptoms, we further refer to the symptom classification system in WebXRBugStudy [21] on the Web AR bugs and reuse their symptom categories for better alignment. For symptoms not covered in the previous study, we label them with new category tags and resolve the conflicts using the approach mentioned above.

Labeling Symptoms of Issue Reports. When labeling issue reports, we first performed a high-level classification of the issue reports into three categories: non-bugs, non-AR-bugs, and AR-bugs. The non-bugs category includes issue reports that describe feature requests, questions, comments and other software related issues, but not a buggy behavior of the corresponding software. The non-AR-bugs category includes issue reports that describe a buggy behavior, but the behavior is not related to AR features of the corresponding software. For example, errors in software log-in, network connections, system I/O and crashes happening before the AR sessions are turned on are all considered as non-AR-bugs. The AR-bugs category includes issue reports that describe buggy behavior related to AR features (i.e., those happened when the AR session is being started). Among all 2,846 issue reports, our initial labeling process identified 296 AR-specific bugs reports and 184 general bugs reports.

We further label the AR-bugs to categorize them based on their symptoms. During the labeling, we refer to the same set of category tags we developed during the labeling of user reviews, and create new category tags only when we are not able to find a proper category tag in the existing set.

Labeling Root Causes of Issue Reports. Besides symptoms, we also label the issue reports based on their root causes. Since we want to align our study with existing studies, before labeling, we try to align the root cause classification systems in both DroidBugStudy [41] (which has 16 root causes in three categories) and WebXRBugStudy [21] (which has 19 root causes in seven categories) papers. It should be noted that when labeling symptoms we refer to only the WebXRBugStudy because we focus on AR bugs and thus most of the symptoms in DroidBugStudy are irrelevant. However, the root causes are more common and apply to both AR and non-AR features, so we decide to further consider the root-cause classification system in DroidBugStudy.

In Table 3, we present how we align root causes from DroidBugStudy and *WebXRStudy* to form a unified root cause classification. From the tables, we can see that the two classifications differ

Root Cause	DroidBugStudy	WebXRStudy
API Misuse	Third Party Library Misuse (subtype of General Programming Error), Android API Misuse (subtype of Android-related error)	ARG (API Arguments), including subtypes of Wrong API Configurations, Missing Arguments, and Wrong Argument Values
General Programming Error	General Programming Error, including subtypes of incorrect assignments, missing case, multi-thread error, wrong control flow, and exception handling	
Data Asset	Android Resource Related Error (subtype of Android-related error)	
External Bugs	Third-party Library Bugs and Limitations, Android Framework Bugs (subtype of Android-related error)	DEPEND (bugs in dependencies)
Event Handling	Android Mechanism Related Error (subtype of Android-related error)	EVENT (Event Handling), including subtypes lifecyle events, system events and other events
Incompatibilities	Android Compatibility Issue (subtype of Android-related error)	COMPAT (Incompatibilities) including hardware, library, OS, and other incompatibilities
Missing Feature	Missing Feature (subtype of General Programming Error)	DUI (missing support of diversified UI)
Redundant Code		RDDOP (Redundant Operations)

Table 3. Alignment of Root Causes in DroidBugStudy, WebXRStudy, and Our Study

a lot on their granularity on different bug categories. In particular, DroidBugStudy has a more fine-grained classification for general program errors, while *WebXRStudy* has a more fine-grained classification for life-cycle events, API misuse, and incompatibility. This is reasonable because the two studies were performed on two very different data sets (Android functional bugs vs. Web-based XR bugs), and their classification systems were generated through open coding based on the distribution of root causes in their corresponding dataset.

However, the data-centric classification systems also make it very difficult to compare results of different studies and understand the characteristic of certain bug groups (e.g., Android AR Bugs) in a larger context. Therefore, in our study, we first create a union set of all root causes from both studies and then map these root causes to each other to align the two bug classifications.

The root causes from both studies are aligned to eight root causes. We performed some re-categorization to enable the alignment. For example, in DroidBugStudy, *Third Party Library Misuse* and *Android API Misuse* are subtypes of *General Programming Error* and *Android Related Error*, respectively, but they are grouped as one root cause in WebXRBugStudy. For alignment, we consider them as one root casue *API Misuse*. Also, DUI (missing support of Diversified UI) is a root cause in WebXRBugStudy. We believe the root cause *Missing Feature* from DroidBugStudy is more general and *DUI* is a special case of *Missing Feature*, so we align them into one root cause *Missing Feature*.

In our root cause study, we use this pre-defined aligned classification instead of creating a new classification from the data (although we still add new root causes that are not covered by the existing classification system), so our study results can be easily compared with both previous studies.

We calculated the Cohen's kappa scores for all of our labeling tasks. Specifically, the score is 0.845 for identifying defect-related user review, 0.736 for categorization of user reviews, 0.764 for identifying confirmed AR-related and non-AR-related issues reports, 0.720 for labeling symptoms of issue reports, and 0.708 for labeling root causes of issue reports. The scores are mostly between 0.7 and 0.8, showing substantial agreement between labeling results.

3.3 Code Commit Analysis

For a portion of the bugs confirmed by the developers, we are able to track their corresponding bug fixing commits. From these commits, we can further study how Android AR bugs were fixed by the developers. Our commit tracking includes the following three steps.

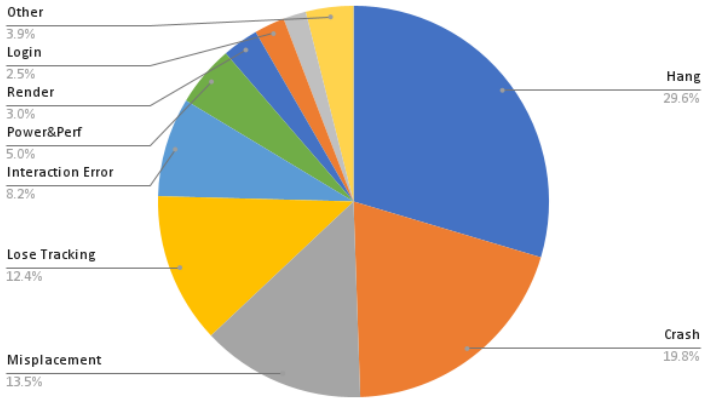

Fig. 1. Categorization of Bug Symptoms in AR App Reviews

- For each bug, we examine the comments in the corresponding issue report to identify any commit IDs or pull request IDs. For all commits or pull requests mentioned in the comments, we investigate the their code change and messages to find out whether they are fixes of the bug. It should be noted that we may end up identify multiple commits or part of a pull request (when a pull request contains multiple commits) as the fixing commits of a bug.
- If we cannot identify bug-fixing commits or pull requests in the comments, we acquire all the code commits between the date of the last comment indicating the bug has not been fixed (e.g., "Will look into it", "The bug seems to be related to ...", "Will fix it") and the date of the first comment indicating the bug has been fixed (e.g., "Closed", "Fixed in version 2.0", "Should work now"). In the acquired commits, we search for the bug report ID and the keywords in the bug report title, and try to identify bug-fixing commits from the search results.
- If we still cannot identify bug-fixing commits from the search results, we perform a final scan on up to 10 commits immediately before the date of the first comment indicating the bug has been fixed, and committed by the developer who posted that comment.

With the process above, we are able to map 227 of all 480 confirmed bugs to their bug-fixing commits. We believe this matched dataset can also be valuable for future research in the area.

Our study then tracked several key metrics from each bug-fixing commit: the number of files modified, lines of code (LOC) added or removed, and changes to API calls. Tracking API calls added or removed helps us understand a bug fix's correlation with underlying AR frameworks and libraries, and whether certain API methods are specifically error-prone.

4 Study Results

In this section, we present the study results to answer each of the research questions.

4.1 RQ1: Commonly Complained Bugs from User Reviews

To answer RQ1, we categorize 635 negative reviews from 67 top AR apps mentioning defective app behaviors, and the results are shown in Figure 1. Our categorization identifies the following major complained bug symptoms.

- **Hang**: A user review is put into this category when it contains descriptions such as "hang", "black/white screen", or "freezing". It indicates that the app goes into an abnormal state and cannot respond to any user input.

- **Crash**: A user review is put into this category when it contains descriptions such as "crash" or "unexpected quit". It indicates that the app unexpected exits with failures.
- **Misplacement**: A user review is put into this category when it contains descriptions such as "float", "block", or "object placed too far". It indicates that a virtual object is not placed in the correct location.
- **Lose Tracking**: A user review is put into this category when it contains descriptions such as "no plane" or "cannot find surface". It indicates that the app cannot properly fetch trackables (e.g., planes, images, human faces) from the physical world.
- **Interaction Error**: A user review is put into this category when it contains descriptions such as "no effect", "nothing happens", or "it should do ... but". It indicates that the app is not deliver a proper response / action after a user interaction.
- **Power&Perf**: A user review is put into this category when it contains descriptions such as "lag", "slow", or "overheating". It indicates that the app is experiencing low performance or fast energy drainage.
- **Render**: A user review is put into this category when it contains phrases such as "looks weird" or "wrong color". It indicates that a virtual object is not rendered correctly.
- **Login**: A user review is put into this category when it contains phrases such as "login error" or "cannog login". It indicates that the app shows error in its login process.

It should be noted that the number of bugs in negative reviews does not necessarily reflect the number of bugs. For example, crashes and hangs are easy-to-see errors, so their proportion may have been exaggerated in the statistics because they are more severe and users facing them are more likely to write a negative review. Despite the imprecise reflection of bug distribution, we believe the categorization can provide a general landscape on what AR app users mostly care and complain about.

From Figure 1, we have the following observations. First, *Hang* and *Crash* are the two largest categories, together accounting for almost half of the reviews. As mentioned above, this is reasonable as these two types of bugs are the most severe and the most likely to be reported in reviews. Second, *Misplacement*, *Lose Tracking*, and *Interaction Error* are the following three largest bug categories. All of them are related to augmented reality features and they have a similar share among all bugs in user reviews. Third, complaints on performance and virtual object rendering are relatively uncommon in the user reviews studied. We summarize these observations as our Finding 1.

> **Finding 1:** Mobile AR apps users complain most about *Hang* and *Crash* bugs, followed by three AR-feature-related bug symptoms: *Misplacement*, *Lose Tracking*, and *Interaction Error*.

Since it is possible that many negative reviews of a certain bug symptom come from a small number of apps, we further study the the distribution of bug symptom in different apps. Figure 2 presents how many apps receive negative reviews of each bug symptom.

From Figure 2, we can see that the top bug symptoms in Figure 1 are still most common seen bug symptoms across apps. The leading edge of *Hang* and *Crash* are not as large as in Figure 1, which somewhat confirms our earlier speculation that they affect user experience more so they are more likely to be reported. The figure also shows that dysfunction bugs (i.e., bugs causing the app to not work at all, including Hang, Crash, Lose Tracking, and part of Interaction Errors) are commonly complaint among apps, indicating that the overall quality of AR apps (or AR features) are low, and they may have not been well tested because such bugs should be relatively easily detected in testing. Other symptoms such as *Power&Perf* bugs and *Render* bugs are also common,

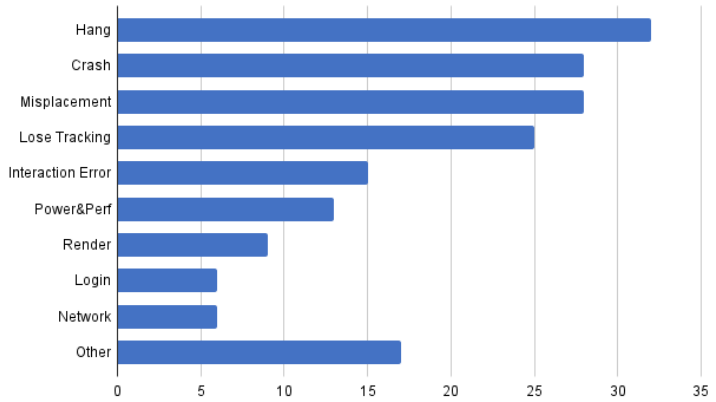

Fig. 2. Number of AR Apps with Each Bug Symptoms in Reviews

but users may not complain about them much because they are relatively minor compared with other bug symptoms. We summarize these observations as our Finding 2.

> **Finding 2:** Dysfunction bugs are commonly complained across many apps, indicating general low quality of AR apps/features and potential lack of testing.

4.2 RQ2: Common Symptoms and Root Causes of Bug in Open Source AR Projects

To answer **RQ2**, we identified 480 bugs from 2,846 issue reports in 77 open source AR projects from Github, and labeled their symptoms and root causes. Since many AR software projects also have non-AR features (e.g., log in, configuration, traditional GUI), we separate bugs into AR bugs (296 bugs) and non-AR bugs (184 bugs). A bug is considered an AR bug if it is triggered when an AR feature of the software is used, and all other bugs are considered non-AR bugs. To keep consistency and allow more clear comparison, the symptom categories are aligned between user reviews (RQ1) and GitHub bug reports (RQ2). There are four additional major bug symptoms that are not listed in Section 4.1, so we list as follows.

- **Camera**: A bug is put into this category when it reflects anomaly of the camera (such as not focusing). This symptom also appears in user reviews but is categorized as *Others* due to its rarity.
- **Internal State**: A bug is put into this category when it reflects an internal state error without external behavior symptoms, such as a failed unit test. This symptom only appear in GitHub bugs instead of user reviews because normal users can never observe internal state errors.
- **Wrong Output**: A bug is put into this category when it reflects an erroneous output not falling into other categories (such as incorrect messages or text on the screen). We do not observe this symptom in user reviews.
- **Network**: A bug is put into this category when it reflects network errors such as an offline server. This symptom also appears in user reviews but is categorized as *Others* due to its rarity.

Figures 3 (a) and Figure 3 (b) show the distribution of symptom categories among AR bugs and non-AR bugs, respectively. From the figures, we can see that AR Bugs and Non-AR Bugs have very different profile on symptom distributions. First, *Hang* and *Crash* bugs account for a large portion (more than half) of Non-AR Bugs, but their proportions in AR Bugs are relatively small. One

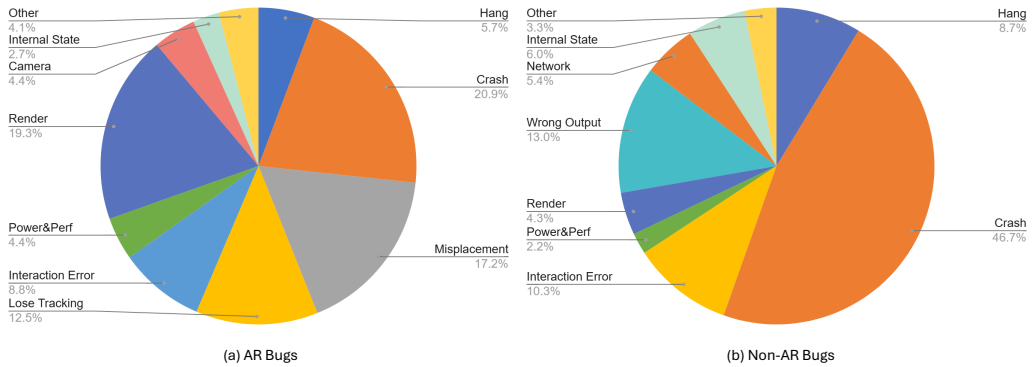

Fig. 3. Symptom Distribution in AR and Non-AR Bugs from GitHub AR Projects

potential reason is that *Hang* and *Crash* bugs often happen when the app starts so AR features have not been reached yet. Second, the three commonly complaint AR-related symptoms (*Misplacement*, *Lose Tracking*, and *Interaction Error*) appear almost exclusively in AR bugs, with *Interaction Error* as an exception because traditional GUI also may have interaction errors. Third, *Render* accounts for a large proportion of AR bugs, and *Wrong Output* accounts for a large portion of Non-AR bugs. From these observations, we summarize Finding 3.

> **Finding 3:** AR apps often have both AR and Non-AR features which have very different bug symptom distributions, so separate bug detection strategies may be used to handle them.

Furthermore, comparing Figure 3 with Figure 1, we can see that although their distribution of symptoms generally follow the same pattern, there are two major differences. First, *Hang* symptom is the most common one in user reviews, but are not so common in GitHub bugs. We believe the reason is that, normal users tend to describe bugs in a brief and superficial way, so they may also use the general terms "hang" or "black / white screen" to describe *Crash* or *Lose Tracking*. In contrast, developers in GitHub projects tend to describe bugs from a technical perspective, so they will describe a same bug as a *Crash* (e.g., because stack traces were observed in the log) or *Lose Tracking* (because they know that the app is in the tracking phase). This observation also shows that a large portion of the *Hang* symptoms complained by users may be actually *Crash* or *Lose Tracking* bugs behind the scene.

Second, *Render* bugs account for only 3% of negative reviews, but they account for more than 19% of GitHub AR bugs, more than all the other AR-related symptoms. This shows that *Render* bugs may be relatively tolerable by users so they do not complain about them much despite its commonality. Another potential reason is that *Render* bugs may not be easily noticed by normal users who do not know the ground truth rendering. For example, many GitHub *Render* bugs are about incorrect color or shade effects. Users may not be aware of them if they do not know what is the supposed color and shade effects.

> **Finding 4:** Many user complained *Hang* bugs may actually refer to *Crash* or *Lose Tracking* bugs, which may need further attention. *Render* bugs are common but do not concern users much, which may be considered when making decisions on effort investment.

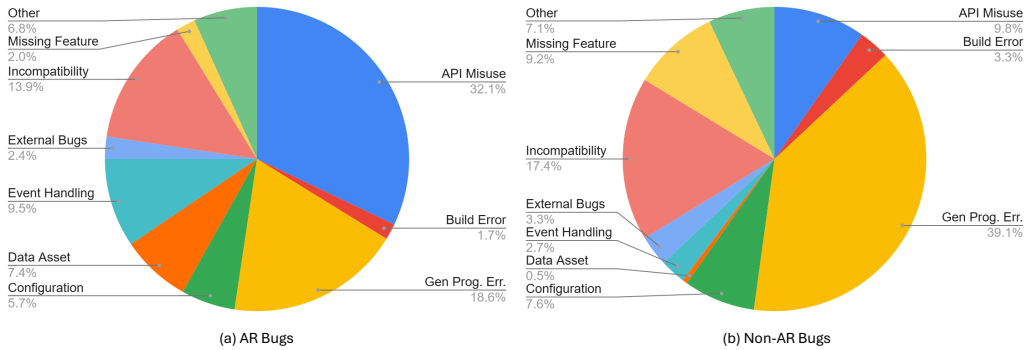

Fig. 4. Root Cause Distribution in AR and Non-AR Bugs from GitHub AR Projects

Figures 4 (a) and 4 (b) show the distribution of root cause categories among AR bugs and non-AR bugs, respectively. Referring existing studies [21, 41], we categorize root causes of GitHub bugs into the following categories.

- **API Misuse**: This root cause refers to misuse of API methods, including platform API (e.g., Android SDK and ARCore), library API (e.g., Android SceneForm), and internal API (programming interface of the project's internal modules).
- **Build Error**: This root cause refers to errors in build configuration or scripts. Note that we do not include build-time failures in our study, so these bugs are all runtime errors caused by wrong build configuration (e.g., building with a library of wrong version or configuration).
- **General Programming Error**: This root cause refers to all common code logic errors. We borrow this root cause and its subtypes (described later) from the *DroidBugStudy* [41].
- **Configuration**: This root cause refers to errors in runtime configurations such as a wrong option value for a library or the AR platform in meta files. Note that errors in build configurations are put into category *Build Error*, and incorrect configurations performed with API calls (e.g., value setting methods) are put into category *API Misuse*.
- **Data Asset**: This root cause refers to errors in resource files and asset files such as 3D models, images, and data files.
- **Event Handling**: This root cause refers to erroneous implementation of life cycle events (e.g., `ARSession.OnPause()`) and other events (e.g., from the underlying operating system).
- **External Bugs**: This root cause refers to bugs in external code such as the AR platform (including its technical limitation on identifying objects in the physical environments) and third-party libraries.
- **Incompatibilities**: This root cause refers to incompatibility between the AR app and its environment including the hardware, the operating system, and third-party libraries.
- **Missing Feature**: This root cause refers to unimplemented features in the app which cause bugs.
- **Other**: This category includes bugs with root causes cannot be confirmed and singletons that cannot be categorized.

From the figures, we have the following observations. First, *API Misuse* is the most common root cause for AR bugs, and *General programming Error* is the most common root cause for non-AR bugs, further confirming the different characteristics of AR and non-AR bugs. Second, *Incompatibility* is a major root cause for both AR and non-AR bugs, showing its commonality across different code

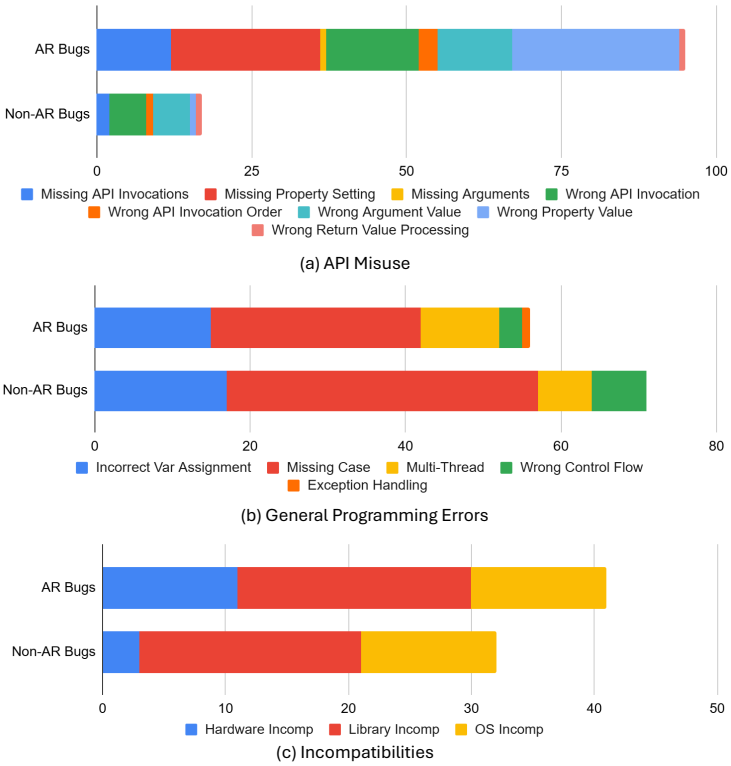

Fig. 5. Subtype Distribution of Major Root Causes in AR and Non-AR Bugs

and features. Third, *Event Handling* and *Data Asset* are two root causes more specific to AR Bugs, as they appear frequently in AR bugs, but not so frequently in Non-AR bugs. Summaring these observations we have Finding 5.

> **Finding 5:** AR bugs and non AR bugs have very different distribution of root causes. *API Misuse*, *Event Handling*, and *Data Asset* are three root causes more specific to AR bugs.

In Figure 4, we can see that *API Misuse*, *General Programming Error*, and *Incompatibilities* and three most common root causes across all bugs. Since these root cause categories cover multiple cases, we further divide them into subtypes. In particular, we divide *API Misuse* into eight subtypes: *Missing Method Invocation*, *Missing Property Setting*, *Missing Arguments*, *Wrong Method Invocation*, *Wrong Method Invocation Order*, *Wrong Argument Value*, *Wrong Property Value*, and *Wrong Return Value Processing*. In particular, we separate property settings (where an assignment to an API field or a simple setter method is called) from other method invocations because they are very common and much simpler in nature. We also separate "Missing" errors and "Wrong" errors, where the former indicates that a method invocation or property setting is missing, and the latter indicates that a wrong method has been called or a wrong value has been set. Another specific subtype is *Wrong Return Value Processing* where the code incorrectly process the return values from an API invocation, such as forgetting to cast the return value to a subclass, or extracting the wrong part from it.

Following the same definition in DroidBugStudy [41], we divide *General Programming Errors* into five subtypes: *Incorrect Variable Assignment*, *Missing Case*, *Multi-Thread*, *Wrong Control Flow*, *Exception Handling*. Note that *Missing Feature* has also been categorized into *General Programming Errors* in DroidBugStudy, but we decide to move it out as a separate root cause because it is more requirement related. Finally, for *Incompatibility*, we divide it into *Hardware*, *OS*, and *Library* based on the source of the incompatibility, following the WebXRBugStudy [21].

From Figure 5, we have the following observations. First, *Missing API Property Setting* and *Wrong Property Value* are two most common subtypes of *API Misuse* in AR bugs and they are very rare in non-AR bugs, showing that property setting is an error-prone part in AR feature implementation, and developers often miss property settings or setting wrong values. Second, *Missing API invocation* is another subtype common in AR bugs but uncommon in non-AR bugs. We further observe that AR feature implementation often involves patterns of multiple API calls, and missing an API call in a pattern often triggers bugs. Third, the subtype distribution of *General Programming Errors* in AR bugs and non-AR bugs are very similar, re-affirming that these root causes are general across bug types and features. Fourth, the distributions of *OS Incompatibility* and *Library Incompatibility* are similar across AR and non-AR bugs, but *Hardware Incompatibility* is more seen in AR bugs. This is because AR features use more hardware components (e.g., sensors, cameras), so they are more likely to trigger hardware incompatibilities. Summarizing these observations we have Finding 6.

> **Finding 6:** *Missing Property Setting* and *Wrong Property Value* are two mostly seen root cause subtypes, showing that property setting is an AR-specific error-prone programming task and may need more support from tools and documentation.

4.3 RQ3: Root Causes of Commonly Complained Bugs

To answer **RQ3**, we correlated symptoms with the corresponding root causes and put the date in Tables 4 and 5. The tables show the root-cause distribution of bugs with specific symptoms, with highlight on larger numbers. From the tables, we have the following observations. First, *Crash* symptom is mostly caused by *General Programming Errors* and *Incompatibilities*. In AR bugs, it is also often caused by *API Misuse* and *Event Handling*. These root causes are generally not AR-specific so enforcing traditional quality assurance practice such as unit / system testing and static code checking should help with reducing *Crash*. Second, *Misplacement* is the most complained AR-specific symptom, and is mainly caused by *API Misuse* and *General Programming Errors*. Looking deeper, 17 of the 25 *API Misuse* cases belong to *Missing Property Setting* and *Wrong Property Value* subtype, and most *General Programming Error* cases are related to wrong calculation of coordinates through projection formula. Furthermore, *Render* symptoms in AR bugs are mostly caused by *API Misuse*, 20 of those 28 cases belong to *Missing Property Setting* and *Wrong Property Value* subtype. These number show that *API Misuse* especially property setting related errors are mostly relevant to the placing and rendering of virtual objects. Third, some *Lose Tacking* and *Misplacement* symptoms are caused by external bugs (5 bugs), including limitation of the AR framework. However, more of those symptoms are caused by other software errors, indicating that software engineering challenges may be a large factor affecting the limited adoption of AR techniques. Fourth, *Render* symptoms in non-AR bugs are mainly caused by *General Programming Error* because these *Render* issues are related traditional GUI instead fo AR UI, further showing the correlation between *API Misuse* and AR UI bugs. Summarizing the above observations, we have Finding 7 and 8.

	API Misuse	Build	Gen. Prog. Err.	Config.	Data Asset	Event Handl.	External	Incomp.	Miss. Feat.
Hang	1	1	4	1	1	4	0	4	0
Crash	11	3	14	2	4	9	0	15	0
Misplacement	25	0	10	2	5	4	1	2	0
Lose Tracking	7	0	4	6	1	5	4	6	0
Interaction Error	10	0	8	0	1	1	0	1	4
Power&Perf	0	0	1	2	0	1	2	4	1
Render	28	0	9	1	10	2	0	2	1
Camera	4	1	1	0	0	0	0	6	0
Internal State	4	0	2	0	0	1	0	0	0

Table 4. Root Cause Distribution over Symptoms in AR Bugs

	API Misuse	Build	Gen. Prog. Err.	Config.	Data Asset	Event Handl.	External	Incomp.	Miss. Feat.
Hang	2	0	7	1	0	0	2	3	1
Crash	3	5	32	8	1	4	3	22	1
Interaction Error	1	1	5	1	0	0	0	1	7
Power&Perf	1	0	0	1	0	0	0	0	0
Render	2	0	5	0	0	0	0	0	0
Wrong Output	1	0	16	2	0	0	0	2	3
Network	2	0	1	0	0	0	1	2	2
Internal State	3	0	5	0	0	1	0	1	1

Table 5. Root Cause Distribution over Symptoms in Non-AR Bugs

Finding 7: Property-setting-related errors are the main root cause behind AR UI symptoms including the placement and the rendering of virtual objects.

Finding 8: Many AR imprecision symptoms (i.e., *Lose Tracking* and *Misplacement*) are not caused by limitation of the AR platform but code errors in the application, highlighting the crucial role of software quality assurance in AR adoption.

4.4 RQ4: Characters of AR Bug Fixes

To answer **RQ4**, we further studied the AR and non-AR bugs for which we can find their bug-fixing commits. Figure 6 shows the the number of lines and files changed on code files and resource files in the bug-fixing commits of AR / non-AR bugs. From the figure we can see that bug-fixing commits of AR and non-AR bugs have similar number of revised code files and code lines, but the bug-fixing commits of AR bugs have many more revised non-code files and lines than non-AR bugs. Therefore, we have Finding 9.

Finding 9: AR bug fixes are more likely to involve non-code files such as resource, asset, and configuration files. So techniques tracing dependencies between code and non-code files could be very helpful for fixing AR bugs.

API Misuse and *Event Handling* are two major root causes for AR bugs and they are platform-related, so findings on concrete API methods and events are more likely to generalize. Therefore, we further investigated the top five misused API and top five incorrectly handled events in AR bugs and list them in Tables 6 and 7. From the tables, we can see that the top misused API methods and mishandled events cover a large portion (27 of 94 *API Misuse* cases and 17 of 28 *Event Handling* cases) of relevant cases. Furthermore, initialization and cleaning-up issues dominate event handling errors. Based on these observations, we have Finding 10.

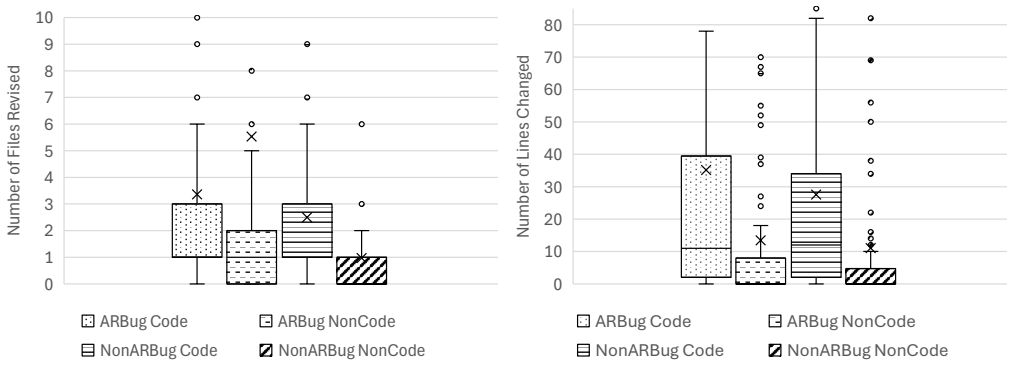

Fig. 6. Number of Updated Code and Non-Code Files and Lines in AR and NonAR Bug-fixing Commits

API	Freq.	Most Common Reason
com.google.ar.sceneform.ux.ARFragment.getScene()	7	Forgetting to call the method when adding objects, often causing Null Pointer Exceptions.
com.google.ar.sceneform.Node.setRenderable(...)	7	Forgetting to set renderable or passing wrong arguments on virtual objects, causing them to be invisible, at wrong location, or missing rendering effects.
uk.co.appoly.arcorelocation.LocationScene.setAnchorRefreshInterval(...)	5	Forgetting to set the property or passing a wrong value will cause lose tracking or misplacement errors.
com.google.ar.core.Session.createAnchor(...)	4	Wrong arguments are passed, causing lose tracking and misplacement errors.
com.google.ar.sceneform.rendering.RenderableInstance.setCulling(...)	4	Forgetting to set the property or passing a wrong value will cause wrong rendering effects.

Table 6. Top Five Misused API Methods

Life Cycle Event	Freq.	Most Common Reason
com.google.ar.core.ARSession.OnDestroy	5	Missing or Incomplete clean up of objects or resources
com.google.ar.core.ARSession.Reset	5	Forgetting to initialize or clean up objects or resources
com.google.ar.core.ARSession.OnCreate	3	Forgetting to initializing scenes or objects
com.google.ar.core.OnPause	2	Missing or Incomplete clean up of objects or resources
com.google.ar.core.OnFrameUpdate	2	Missing required operations

Table 7. Top Five Incorrectly Handled Events

Finding 10: A large portion of *API Misuse* and *Event Handling* cases are related to a small number of API methods and initialization / cleaning up operations. Pattern or AI-based helpers and reminders may help largely reduce such errors.

4.5 RQ5: Comparison with Previous Studies

Finally, to answer **RQ5**, we compare the distribution root causes of the bugs in our study and those from two previous studies [41] [21].The results are presented in Table 8. In the table, if prior studies do not report bugs with specific root cause, we leave the cell as blank because it is not clear whether such bugs were not found or such bugs were uncommon so they were categorized into the *Other* category.

Root Causes		AR Bugs (Our Study)	Non-AR Bugs (Our Study)	Droid Bug Study [41]	Web XR Bug Study [21]
API Misuse	Total	95 (32.1%)	18 (9.8%)	24 (6.1%)	27 (7.3%)
	-Missing API Invoc.	12 (4.1%)	2 (1.1%)		
	-Missing Property Set	24 (8.1%)	0		
	-Missing Args	1 (0.3%)	0		4 (1.1%)
	-Wrong API Invoc.	15 (5.1%)	6 (3.3%)		
	-Wrong API Invoc. Order	3 (1.0%)	1 (0.5%)		
	-Wrong Arg Value	12 (4.1%)	6 (3.3%)		7 (1.9%)
	-Wrong Property Value	27 (9.1%)	1 (0.5%)		16 (4.3%)
	-Wrong Return Value Proc	1 (0.3%)	2 (1.1%)		
Build		5 (1.7%)	6 (3.3%)		
Gen. Prog. Err.	Total	55 (18.6%)	72 (39.1%)	151 (38.4%)	
	-Incorrect Assign	15 (5.1%)	17 (9.2%)	19 (4.8%)	
	-Missing Case	27 (9.1%)	40 (21.7%)	62 (15.8%)	
	-Multi-Thread	10 (3.4%)	7 (3.8%)	34 (8.7%)	
	-Wrong Control Flow	3 (1.0%)	7 (3.8%)	26 (6.6%)	
	-Exception Handling	0	1 (0.5%)	10 (2.5%)	
Configuration		17 (5.7%)	14 (7.6%)		
Data Asset		22 (7.4%)	1 (0.5%)	62 (15.8%)	
Event Handl.	Total	28 (9.5%)	5 (2.7%)	45 (11.4%)	76 (20.7%)
	-Lifecycle Event	28 (9.5%)	4 (2.2%)	45 (11.4%)	56 (15.2%)
	-OS Event	0	1 (0.5%)		17 (4.6%)
	-Other Event	0	0		3 (0.8%)
External		7 (2.4%)	6 (3.3%)	22 (5.6%)	15 (4.1%)
Incomp.	Total	41 (13.9%)	32 (17.4%)	27 (6.9%)	78 (21.2%)
	-Hardware	11 (3.8%)	3 (1.6%)		35 (9.5%)
	-OS	11 (3.8%)	11 (6.0%)		11 (3.0%)
	-Library	19 (6.4%)	18 (9.8%)		17 (4.6%)
	-Other	0	0		15 (4.1%)
Missing Feature		6 (2.0%)	17 (9.2%)	29 (7.4%)	36 (9.8%)
Redundant Code		0	1 (0.5%)		11 (3.0%)
Other		20 (6.8%)	12 (6.5%)	33 (8.4%)	125 (34.0%)
Total		**296**	**184**	**393**	**368**

Table 8. Comparison of Root Cause Distribution with Existing Studies

From the table, we have the following observations. First, compared with DroidBugStudy, our study shows many more *API Misuse* cases in AR bugs, but similar number of *API Misuse* cases in non-AR bugs, indicating that AR features highly rely on API usages and AR developers may not be always familiar with APIs. Our non-AR bugs show similar proportion of root causes from *General Programming Errors* with DroidBugStudy, showing their consistent distribution in different software and bugs. Our study also show many more *Incompatibility* cases in our bug dataset, indicating that AR apps are more likely to suffer from Incompatibilities due to more complicated dependencies on hardware and libraries. Second, compared with WebXRBugStudy, our study also shows more *API Misuse* cases, potentially because web-based software rely less on directly API method calls, and the property settings as well as *General Programming Errors* may have been categorized to the *Other* category. Our study also shows fewer *Incompatibility* cases than WebXRBugStudy, potentially because web-based XR system has an additional layer of dependency: the HTTP server and the browser. To sum up, we have Finding 11.

Finding 11: The root causes of bugs in mobile AR software share some similarity with existing bug studies, such as the similar distribution of *General Programming Errors* in non-AR features with traditional Android apps, and the high proportion of incompatibilities similar to Web XR Software. The specialty of mobile AR apps is that their AR features heavily rely on platform and third-party API libraries which developers may not always be familiar with.

4.6 Threats to Validity

The major threat to the internal validity of our study comes from errors in our data collection and labeling process. To reduce the threat, we carefully checked all of the scripts used for data collection, and have two of the authors independently label all datasets with other authors helping on resolving conflicts. The Cohen's Kappa scores of our labeling tasks are mostly between 0.7 and 0.8, showing substantial agreement. The major threat to the external validity of our study is whether our study results can be generalized. To reduce this threat, we use a large set of AR apps and GitHub AR projects with high downloads and high number of stars. We further align and compare our study results with prior bugs studies. The consistency (on the distribution of general programming errors, incompatibilities, and other root causes such as missing features) supports that our study results is to some extent generalizable.

5 Lessons Learned

In this section, from the observations and findings of our study, we further summarize lesson learned for different parties and propose actionable items.

Lessons learned for AR app developers. Our study shows that the most complained bug symptoms in AR apps are still dysfunction bugs (e.g., *Hang*, *Crash*) and these bugs are common among top AR apps, showing that the general quality of AR apps is low. Furthermore, many of these symptoms are caused by traditional root causes such as *General Programming Errors* and *Incompatibility*, which may be easily detected through random testing and smoke testing. Therefore, AR app developers should invest more effort on software quality and testing their apps. Writing more unit tests and perform random testing or monkey testing on their apps may help detect many bugs and largely enhance their user satisfaction. The commonality of bugs caused *Incompatibility* also shows that using multiple devices or emulators in testing is especially important for AR apps. In contrast, *Render*-related bugs are not often complained so they can be handled with lower priority.

Lessons learned for AR platform developers. Our study shows that *API Misuse* is the most common root cause of AR bugs, so AR platforms have a large room to improve, especially on providing more examples and documentation on how and when to use API methods. These examples may also be leveraged by large language models or co-pilot as training data and later better help AR app developers. Our study also finds that app developers often forget (or do not know they need) to set properties or invoke API methods. One API design to avoid such missing cases is to inject conditional API access into parameters. For example, if client developers often need to call function `B(...)` after function `A(...)` (not always, otherwise B can be incorporated into A), the platform developers can put the invocation to B at the end of A's code, and add a flag parameter to `A(...)`. Therefore, `A(...)` becomes `A(..., boolean callB)`, and client developers are enforced to consider whether B needs to be called.

Lessons learned for Software Engineering Researchers. Our study shows that many bug symptoms in mobile AR apps should have been detected with automatic testing tools, and many API-related bugs should have been avoided by co-pilot or API guidance tools. However, these existing techniques seem to have not helped AR developers a lot, potentially due to the challenges of adopting them for AR apps. Automatic testing of AR apps is extremely difficult due to the requirement of physical environment setup. Some recent frameworks [6] [14] allow automatic testing with VR scenes or videos, but it is still not clear whether they can trigger bugs as in real world usage scenarios. The vague user interface of AR apps places another challenge for automatic testing. Although co-pilot and LLMs can often provide correct API usage examples or detect API usage errors, AR apps are different in that there is often not an absolute correct way of using API. Whether and how to access API often depends on the desired way to interact with physical world

and rendering effect. App developers often needs to determine API usage while watching how the app perform in real physical world, which limits the usability of API usage recommendation tools. Our study calls for novel testing and code recommendation techniques to address the above challenges and study their effectiveness. Actually, our dataset can be used for such research tasks, and the symptom / root cause distribution we find may also be used to develop mutation-based benchmarks for various bug detection tools.

6 Related Works

Mobile and XR Bug Studies. Besides the two studies (DroidBugStudy and WebXRBugStudy) we compare in our paper, there are also other recent studies on mobile and XR bugs. Johnson et al.[16] studied the reproduction of bug reports in Android apps, revealing factors that influence whether developers can reproduce reported issues. Hogan et al. [10], examined the distinctive performance problems in VR applications, characterizing common performance bottlenecks. Guo et al. [9] conducted a large-scale empirical study of Oculus VR applications and uncovered significant security and privacy weaknesses. Li et al. [20] studied visual inconsistency issues in VR applications and developed a noval technique to detect such issues. Tang et al. [36] explored app-review–driven collaborative bug finding, showing that user-generated reviews can be leveraged to discover and cluster recurring issues.

Augmented Reality Software Development. AR techniques have been adopted in different domains such as education, entertainment and sports [29] [13] [3] [4]. Large indoor areas also adopting indoor navigation tools to guiding users to desired location [12] [17] [37] [15] [26]. AR enables doctors and surgeons [28] visualize complex 3D models of MRI, CT Scans [27], and Ultrasound scans [22] offering detailed view of internal organs assisting in improved diagnosis and surgical planning [35] [38]. With smart glasses such as Vuzix Blade [39], the hardware market is growing at an unprecedented pace. Matching pace with the hardware arena, several SDk platforms including Google ARCore [6], Apple ARKit [14], Vuforia [31], and Wikitude![40] are also offering distinct features and capabilities.

Software engineering techniques, especially testing techniques, have also been developed for AR app. ARCHIE [18] [19] collects user feedback and system state data to identify and debug issues in AR applications in real-world settings. XR testing mapping study [8] provides the first systematic mapping of software testing for XR, tools and datasets. Model-Based Testing for AR [30] explores model-based testing notions tailored to AR app behaviors and scene graphs. Youkai [1, 5] describes a cross-platform framework to script tests for VR/AR apps across engines/devices.

7 Conclusion

As Augmented Reality (AR) applications gain popularity, understanding and addressing their unique software bugs is increasingly important. In this study, we analyzed user reviews from Google Play and issue reports from open-source AR projects on GitHub, categorizing bug symptoms and root causes, then examining their correlations and fix commits. Our findings reveal that AR users are most affected by dysfunction bugs like hangs and crashes, API misuse—particularly property setting errors—is the most frequent root cause, a small number of API patterns and event-handling practices account for many of these issues. In the future, we plan to explore automated detection techniques for API misuse patterns in AR apps to prevent these issues earlier in development. Additionally, we plan to work on longitudinal studies to examine how AR bug characteristics evolve as AR apps and platform get mature.

Data Availability. Our dataset, including the reviews, issue reports, and all scripts used in our study is available in the uploaded replication package.

References

[1] Stevão A. Andrade, Fátima L. S. Nunes, and Márcio E. Delamaro. 2020. Automated Test of VR Applications. In *Advances in Computer Graphics*. LNCS, Vol. 12221. Springer. https://doi.org/10.1007/978-3-030-60703-6_18

[2] Apple. 2024. Apple Vision Pro Augmented Reality Headset. Apple Inc.. Available: https://www.apple.com/vision-pro, [Accessed: October 20, 2024].

[3] AroundAR. 2024. Around AR: Augmented Reality Solutions and Experiences. https://aroundar.com/ Accessed: October 29, 2024.

[4] dfl.de. 2024. Video on Augmented Reality Experience. https://www.youtube.com/watch?v=vYqOG_Tzi4I Accessed: October 29, 2024.

[5] Thiago Figueira and Adriano Gil. 2022. Youkai: A Cross-Platform Framework for Testing VR/AR Apps. In *HCI International 2022 – Late Breaking Papers: Interacting with Computers*. Springer, 3–12. https://doi.org/10.1007/978-3-031-21707-4_1

[6] Google. 2024. Google ARCore: Augmented Reality Platform. https://developers.google.com/ar Accessed: October 29, 2024.

[7] Google. 2024. Google Play Store: Apps, Games, and More. https://play.google.com/store Accessed: October 29, 2024.

[8] Ruizhen Gu, José Miguel Rojas, and Donghwan Shin. 2025. Software Testing for Extended Reality Applications: A Systematic Mapping Study. *Automated Software Engineering* (2025). https://doi.org/10.1007/s10515-025-00523-7 Early access.

[9] Hanyang Guo, Hong-Ning Dai, Xiapu Luo, Zibin Zheng, Gengyang Xu, and Fengliang He. 2024. An empirical study on oculus virtual reality applications: Security and privacy perspectives. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[10] Jason Hogan, Aaron Salo, Dhia Elhaq Rzig, Foyzul Hassan, and Bruce Maxim. 2022. Analyzing Performance Issues of Virtual Reality Applications. *CoRR* abs/2211.02013 (2022). https://doi.org/10.48550/arXiv.2211.02013 arXiv preprint.

[11] Josh Howarth. 2024. 24+ Augmented Reality Stats (2024-2028). https://explodingtopics.com/blog/augmented-reality-stats Accessed: April 20, 2024.

[12] Bo-Chen Huang, Jiun Hsu, Edward T.-H. Chu, and Hui-Mei Wu. 2020. ARBIN: Augmented Reality Based Indoor Navigation System. *Sensors* 20, 20 (Oct. 2020), 5890. https://doi.org/10.3390/s20205890

[13] Immersiv.io. 2024. ARISE: Augmented Reality Solutions for Enhanced Experiences. https://www.immersiv.io/arise/ Accessed: October 29, 2024.

[14] Apple Inc. 2024. ARKit: Augmented Reality for iOS. https://developer.apple.com/augmented-reality/arkit/ Accessed: October 29, 2024.

[15] Insider Navigation. 2024. AR Indoor Navigation Solutions. https://insidernavigation.com/ar-indoor-navigation/ Accessed: October 29, 2024.

[16] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 321–332. https://doi.org/10.1109/SANER53432.2022.00048

[17] Rhuta Joshi, Anuja Hiwale, Shivani Birajdar, and Renuka Gound. 2019. *Indoor Navigation with Augmented Reality*. Springer Singapore, 159–165. https://doi.org/10.1007/978-981-13-8715-9_20

[18] Sarah M. Lehman, Semir Elezovikj, Haibin Ling, and Chiu C. Tan. 2020. ARCHIE: A User-Focused Framework for Testing Augmented Reality Applications in the Wild. In *IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. https://doi.org/10.1109/VR46266.2020.00013

[19] Sarah M. Lehman, Semir Elezovikj, Haibin Ling, and Chiu C. Tan. 2023. ARCHIE++: A Cloud-Enabled Framework for Conducting AR System Testing in the Wild. *IEEE Transactions on Visualization and Computer Graphics* 29, 4 (2023), 2102–2116. https://doi.org/10.1109/TVCG.2022.3141029

[20] Shuqing Li, Cuiyun Gao, Jianping Zhang, Yujia Zhang, Yepang Liu, Jiazhen Gu, Yun Peng, and Michael R Lyu. 2024. Less cybersickness, please: Demystifying and detecting stereoscopic visual inconsistencies in virtual reality apps. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2167–2189.

[21] Shuqing Li, Yechang Wu, Yi Liu, Dinghua Wang, Ming Wen, Yida Tao, Yulei Sui, and Yepang Liu. 2020. An exploratory study of bugs in extended reality applications on the web. In *2020 IEEE 31st International symposium on software reliability engineering (ISSRE)*. IEEE, 172–183.

[22] Medivis. 2024. Medivis: AR Body Navigation Solutions. https://www.medivis.com/body-navigation Accessed: October 29, 2024.

[23] Meta. 2024. Meta Quest Headset Series. Meta Platforms, Inc.. Available: https://about.meta.com/quest, [Accessed: October 20, 2024].

[24] Meta. 2024. Ray-Ban and Meta Collaboration for AR Smart Glasses. Meta Platforms, Inc. and Ray-Ban. Available: https://www.meta.com/ray-ban-stories, [Accessed: October 20, 2024].

[25] Microsoft. 2024. Microsoft HoloLens Mixed Reality Headset. Microsoft Corp.. Available: https://www.microsoft.com/en-us/hololens, [Accessed: October 20, 2024].
[26] MobiDev. 2024. Augmented Reality Indoor Navigation App Development. https://mobidev.biz/blog/augmented-reality-indoor-navigation-app-development Accessed: October 29, 2024.
[27] Navaneeth V Nair, Naeema Ziyad, Gopika Madhu, Navya Prasad, and M V Rajesh. 2023. Visualizing MRI and CT Scans Using Mixed Reality. https://www.youtube.com/watch?v=KTN0O4n1Xv8 Accessed: October 29, 2024.
[28] Novarad Corporation. 2018. 3D Medical Images Using Augmented Reality. https://www.youtube.com/watch?v=M3yY_b8jT54 Accessed: October 29, 2024.
[29] Orange 5G Lab. 2024. Immersiv.io: A New Way to Watch Football with Augmented Match and 5G. https://5glab.orange.com/en/realisations/immersiv-io-a-new-way-to-watch-football-with-augmented-match-and-5g/ Accessed: October 29, 2024.
[30] André Porfirio, Antônio Roberto R. Araújo, Gustavo Gava, Jefferson Silva, Erick Souza, and Rodrigo Souza. 2022. An Approach for Model Based Testing of Augmented Reality Applications. In *RCIS 2022 Workshops (CEUR-WS.org, vol. 3201)*. https://ceur-ws.org/Vol-3201/
[31] PTC Inc. 2024. Vuforia: Augmented Reality SDK. https://developer.vuforia.com/ Accessed: October 29, 2024.
[32] Tahmid Rafi, Xueling Zhang, and Xiaoyin Wang. 2022. PredART: Towards Automatic Oracle Prediction of Object Placements in Augmented Reality Testing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. ACM, 1–13. https://doi.org/10.1145/3551349.3561160
[33] sifted.eu. 2024. The future of augmented reality in four charts. https://sifted.eu/articles/future-augmented-reality-data-brnd Accessed: May 29, 2024.
[34] statista.com. 2024. AR VR - Worldwide. https://www.statista.com/outlook/amo/ar-vr/worldwide Accessed: October 20, 2024.
[35] Rui Tang, Long-Fei Ma, Zhi-Xia Rong, Mo-Dan Li, Jian-Ping Zeng, Xue-Dong Wang, Hong-En Liao, and Jia-Hong Dong. 2018. Augmented reality technology for preoperative planning and intraoperative navigation during hepatobiliary surgery: A review of current methods. *Hepatobiliary amp; Pancreatic Diseases International* 17, 2 (April 2018), 101–112. https://doi.org/10.1016/j.hbpd.2018.02.002
[36] Xunzhu Tang, Haoye Tian, Pingfan Kong, Saad Ezzini, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2024. App review driven collaborative bug finding. *Empirical Software Engineering* 29, 5 (2024), 124.
[37] Treedis. 2024. Indoor Navigation Solutions by Use Case. https://www.treedis.com/solutions-by-use-case/indoor-navigation Accessed: October 29, 2024.
[38] Raul N. Uppot, Benjamin Laguna, Colin J. McCarthy, Gianluca De Novi, Andrew Phelps, Eliot Siegel, and Jesse Courtier. 2019. Implementing Virtual and Augmented Reality Tools for Radiology Education and Training, Communication, and Clinical Care. *Radiology* 291, 3 (June 2019), 570–580. https://doi.org/10.1148/radiol.2019182210
[39] Vuzix Corporation. 2024. Vuzix Blade Smart Glasses. https://www.vuzix.com/products/blade-smart-glasses Accessed: October 29, 2024.
[40] Wikitude. 2024. Wikitude: Augmented Reality SDK. https://www.wikitude.com/ Accessed: October 29, 2024.
[41] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1319–1331.
[42] Xiaoyi Yang, Yuxing Wang, Tahmid Rafi, Dongfang Liu, Xiaoyin Wang, and Xueling Zhang. 2024. Towards Automatic Oracle Prediction for AR Testing: Assessing Virtual Object Placement Quality under Real-World Scenes. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.