The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects

Christophe Rezk, *Student Member, IEEE,* Yasutaka Kamei, *Senior Member, IEEE,* and Shane McIntosh, *Member, IEEE*

Abstract—The SZZ approach for identifying fix-inducing changes traces backwards from a commit that fixes a defect to those commits that are implicated in the fix. This approach is at the heart of studies of characteristics of fix-inducing changes, as well as the popular Just-in-Time (JIT) variant of defect prediction. However, some types of commits are invisible to the SZZ approach. We refer to these invisible commits as "Ghost Commits." In this paper, we set out to define, quantify, characterize, and mitigate ghost commits that impact the SZZ algorithm during its mapping (i.e., linking defect-fixing commits to those commits that are implicated by the fix) and filtering phases (i.e., removing improbable fix-inducing commits from the set of implicated commits). We mine the version control repositories of 14 open source Apache projects for instances of mapping-phase and filtering-phase ghost commits. We find that (1) 5.66%–11.72% of defect-fixing commits of defect-fixing commits only add lines, and thus, cannot be mapped back to implicated commits; (2) 1.05%–4.60% of the studied commits only remove lines, and thus, cannot be implicated in future fixes; and (3) that no implicated commits survive the filtering process of 0.35%–14.49% defect-fixing commits. Qualitative analysis of ghost commits reveals that 46.5% of 142 addition-only defect-fixing commits add checks (e.g., null-ness or emptiness checks), while 39.7% of 307 removal-only commits clean up (unused) code. Our results suggest that the next generation of SZZ improvements should be language-aware to connect ghost commits to implicated and defect-fixing commits. Based on our observations, we discuss promising directions for mitigation strategies to address each type of ghost commit. Moreover, we implement mitigation strategies for addition-only commits and evaluate those strategies with respect to a baseline approach. The results indicate that our strategies achieve a precision of 0.753, improving the precision of implicated commits by 39.5 percentage points.

Index Terms—SZZ, fix-inducing changes, defect-fixing changes

1 Introduction

Over the lifetime of evolving software projects, defects are inadvertently introduced during initial development [7], refactoring [1], or when fixing other defects [43]. Identifying changes that are likely to induce future fixes could save developers' time and effort. Additionally, deepening our understanding of these fix-inducing changes and recognizing recurring patterns can help teams to anticipate when changes are likely to induce fixes in the future.

The SZZ approach for identifying fix-inducing changes [32] mines Version Control Systems (VCSs) and Issue Tracking Systems (ITSs) to trace a defect-fixing change back to potential fix-inducing changes that are implicated in the fix. The SZZ approach starts by *identifying* defect-fixing commits by matching a bug report from the ITS to the commit that fixes it. Defect-fixing commits are then *mapped* to implicated changes by extracting the set of removed lines and tracing them through the VCS to the commit(s) that last modified them. Finally, potential fix-inducing commits are *filtered* to eliminate those that should not be implicated in the fix (e.g., implicated changes that appeared after the defect creation date). The surviving implicated commits are labelled as *fix-inducing* commits [32].

While the SZZ approach plays a pivotal role in understanding and predicting fix-inducing changes, it is not without limitations. At its core, the SZZ approach relies on heuristics to handle noisy software repository data; however, there are commits that these heuristics cannot detect. We refer to these invisible commits as *ghost commits*, which impact (at least) two phases of the SZZ approach. First, *Mapping Ghosts* are commits that cannot be detected when connecting defect-fixing commits to potential fix-inducing ones. Second, *Filtering Ghosts* are defect-fixing commits for which no fix-inducing change survives the filtering phase.

We perform an empirical study of ghost commits in 14 open source projects from the Apache Software Foundation. First, we *quantify* the frequency at which ghost commits occur. Second, we *characterize* mapping and filtering ghosts to better understand their properties. Third, we propose and evaluate *mitigation* strategies that address the addition-only ghost commits (the most frequently occurring ghost type). Finally, we study the types of maintenance activities in addition-only ghost commits to compare with recent work on intrinsic/extrinsic bugs [27]. More specifically, we contribute the following definitions and observations:

- *Christophe Rezk is with the Department of Electrical and Computer Engineering, McGill University, Canada. E-mail: christophe.rezk@mail.mcgill.ca*
- *Yasutaka Kamei is with the Principles of Software Languages Group (POSL), Kyushu University, Japan. E-mail: kamei@ait.kyushu-u.ac.jp*
- *Shane McIntosh is with the David R. Cheriton School of Computer Science, University of Waterloo, Canada. E-mail: shane.mcintosh@uwaterloo.ca*

Mapping Ghost 1: Defect-fixing commits with no implicated commits (MG 1)

Definition: Defect-fixing commits that do not remove previously existing lines of code.

Motivation: The SZZ approach maps defect-fixing commits to fix-inducing commits by locating the most recent commit to change the lines that were removed by the fixing commit [32]. However, the SZZ approach cannot map lines that were added during a defect-fixing commit back to fix-inducing changes. In theory, a defect-fixing commit may be entirely comprised of line additions; yet these these commits may have been induced by prior changes. Hence, gaining a better understanding of defect fixes that only add lines is important for those who adopt the SZZ approach.

Quantification: 5.66%–11.72% of defect-fixing commits in the subject systems only add lines, with a median of 7.64%.

Characterization: MG 1 most often contain new *Check*s (*44.6%*), i.e., new if conditions or try-catch blocks. Often, such checks were omitted by prior changes, which ideally would have been implicated in the corresponding fixes.

Mapping Ghost 2: Commits that cannot be implicated in future fixes (MG 2)

Definition: Commits that consist of line removals only.

Motivation: Commits that only remove lines cannot be implicated by SZZ in future defect-fixing activity, since no lines remain in the codebase to which future activities can be mapped. In reality, these commits may be fix-inducing, since the removal of an incorrect line (or set of lines) can wreak havoc on a software system. Studying the frequency and characteristics of removal-only commits will show the magnitude of their potential impact on SZZ-based analyses.

Quantification: 1.05%–4.60% of commits in the subject systems contain line removals only, with a median of 2.68%.

Characterization: *Cleanup* of unnecessary code is the most frequently occurring reason (*39.7%*) for MG 2. Such cleanup activities are not risk-free. For example, the infamous left-pad incident,[1] which caused numerous NODE.JS applications to fail was caused by the removal of code.

Filtering Ghost: Defect-fixing commits with no implicated commits that survive filtering (FG)

Definition: Defect-fixing commits where all implicated fix-inducing commits are removed by the filtering phase.

Motivation: Since a series of filters are applied to the set of potentially fix-inducing commits, there may be defect-fixing commits for which all potentially fix-inducing commits are filtered out. Since these defect-fixing commits are not associated with any fix-inducing commits, it is important for those who adopt SZZ in research and practice to better understand their frequency and characteristics.

Quantification: 0.35%–14.49% of defect-fixing commits in the subject systems are FG, with a median of 5.46%.

Characterization: FG commits are most often related to the issue report date filter (35%). Deeper analysis suggests that the date filter is too aggressive because follow-up fixes are often linked to the same issue ID as the initial work.

Mitigation

Strategies: We propose mitigation strategies for MG 1 commits, which apply data flow analysis to the (set of) identifier(s) in the added lines, and then apply SZZ to the lines in the data flow path of the (set of) identifier(s).

Comparative Analysis: Our approach complements a syntax-based baseline approach [29], which applies SZZ to the lines within the closest surrounding code block. Indeed, both approaches implicate identical commits 21.1% of the time and share at least one commonly implicated commit in 73.2% of the remaining cases.

Precision Analysis: We manually analyze the implicated commits of both approaches to assess whether they truly could have been fix inducing. We find the precision of the Control Flow approach to be 0.753, while the precision of the baseline approach is 0.358. Indeed, the data flow approach is likely more precise because it avoids implicated benign lines that appear within the surrounding code block.

Maintenance Type

To better understand the types of maintenance being performed within ghost commits, we classify our sample of MG 1 commits as corrective, adaptive, or perfective, according to the taxonomy introduced by Swanson [33]. We find that the vast majority (92.4%) of MG 1 commits are corrective, while 5.7% are adaptive, and 2.1% are perfective. These observations share similarities with the recent work of Rodriguez-Perez et al. [27], who found that the fixes for bugs are often extrinsic, i.e., do not have a fix-inducing change. The fixes that we labeled as corrective maintenance are intrinsic in nature, while perfective and adaptive maintenance are often extrinsic. This indicates that 7.6% of ghost commits are extrinsic in nature, which falls within the range of rates reported by Rodriguez-Perez et al. [27].

Our findings suggest that the SZZ approach for detecting fix-inducing changes currently overlooks a considerable amount of commit activity. While these observed proportions are not exceedingly large, they still represent a sizeable proportion of defect-fixing activity for which current SZZ solutions do not apply. Future studies that rely on SZZ should explore context-aware enhancements to create more accurate and reliable SZZ data sets.

2 BACKGROUND

This section describes the stages of the SZZ approach and how ghost commits can impact SZZ-based analyses. Figure 1 contains an overview of the SZZ approach, which (i) merges issues and commits to identify defect-fixing commits (Section 2.1); (ii) maps defect-fixing commits back to prior changes that are implicated by the fix (Section 2.2); and (iii) applies a series of filters to remove implicated changes that are unlikely to have induced the fix (Section 2.3).

2.1 Identifying Defect-Fixing Commits

The first stage identifies which commits in the VCS are **defect fixing**. The assumption being that the occurrence of a fix implies the existence of a defect prior to the fix.

(I 1) Merge Issues & Conflicts: To identify defect-fixing commits, VCS entries must be linked with issue reports in the ITS. Issue reports in the ITS track the development activity backlog for a project. These reports contain rich (meta)data

1. https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/

Fig. 1. An overview of the phases of the SZZ approach. Mapping Ghosts (MG 1, MG 2) are identi ed in the Mapping phase, while Filtering Ghosts (FG) are identi ed in the Filtering phase. These phases are described in Sections 2.1–2.3.

Fig. 2. An overview of the Quanti cation and Characterization phases of our case study: extracting the data needed for SZZ, applying SZZ, and analyzing the ghost commits detected. These phases are described in Section 4.2 and Section 4.3.

about development tasks, including a Type eld, which may be “Defect” or “Enhancement,” for example.

In a nutshell, commits that are linked to issue reports of type “Defect” are considered to be defect xing. Unfortunately, the links between VCS and ITS entries are not explicitly enforced by either tool by default. Recent integrated toolsets, such as GitLab, offer work ows [2] that enforce linking between issues and commits; however, for projects that have not fully adopted a toolset like GitLab, it is a common practice for developers to manually record links between commits and issue reports in commit messages. For example, commit ae90791 from the PIG project includes the message “PIG-5118 Script fails with Invalid dag containing 0 vertices rohini” to indicate that the commit is associated with the issue report PIG-5118 . Thus, to identify defect- xing commits, one must identify the VCS-ITS linkage practices of the subject systems, then recover the VCS-ITS links using repository mining scripts.

Ghost Commit 0 (GC 0): Only the VCS commits that have a recovered link to an ITS record of type “Defect” are included in the SZZ data set. Since link recording practices are rarely enforced, developers may omit necessary links without receiving a warning from the VCS or ITS. Thus, SZZ implementations may miss defect- xing commits where links were omitted. We refer to such missing defect- xing commits as Ghost Commit 0 (GC 0).

GC 0 has been de ned and studied in prior work [3, 21, 39]. Bird et al. [3] report that linkage bias in datasets compromises the validity of software models built using those datasets. Nguyen et al. [21] nd that linkage bias in datasets exists even when strict guidelines are enforced on the development process. Wu et al. [39] propose Relink, a VCS-ITS link recovery tool to rebuild missing links and mitigate linkage bias. Given that GC 0 is already well understood, we do not investigate it further in this paper.

2. https://docs.gitlab.com/ee/topics/gitlab _ ow.html

2.2 Mapping

After VCS-ITS links have been recovered, commits that are implicated in defect- xing commits can be identi ed. This mapping step produces a database, which stores links between defect- xing and potential x-inducing commits.

(M 1) Map Defect-Fixing Commits to Implicated Changes: For each defect- xing commit, its removed lines are selected using the diff command. Next, the parent commit(s) of each removed line are identi ed using the blame command. Note that modifying a line registers as a removal and an addition. Thus, analyzing removed lines covers cases when code is removed or modi ed.

Mapping Ghost 1 (MG 1): Since the mapping step traces lines that previously existed, defect- xing commits that do not remove or modify lines cannot be mapped to implicated commits. We refer to defect- xing commits that do not remove or modify lines as Mapping Ghost type 1 (MG 1). For example, commit 4adc8e4 [3] from the ACTIVEMQ project xes a defect by interrupting the socketHandlerThread to cleanly shut down an embedded broker.

Mapping Ghost 2 (MG 2): Invoking the blame command on removed lines cannot implicate past commits that only remove lines. However, commits that remove the wrong lines can induce future xes. We refer to commits that do not add any new lines of code as Mapping Ghost type 2 (MG 2). For example, in commit c10e8d2 [4] from the HBASE project, the removal of the createWriterInTmp method may lead to a defect x in the future.

2.3 Filtering

Next, a series of lters are applied to remove commits that could not or are unlikely to have led to the future x. This ltering stage reduces the sets of implicated commits to those that are likely to be x inducing.

(F 1) Apply Issue Report Date Filter: Implicated commits that appear after a defect has been reported are unlikely

3. https://github.com/apache/activemq/commit/4adc8e4/
4. https://github.com/apache/hbase/commit/c10e8d2/

to have induced the fix. To mitigate such noise, researchers apply filters to discard such implicated commits (f_1) [32]. To do so, we use the --after <date> flag of the blame command, where <date> is the defect creation date. However, if no modifications were made to a line after the specified date, the ^ character is prepended to the output.

(F 2) Apply Content Filters: Implicated commits that modify whitespace or comments do not alter system behaviour and are unlikely to induce a future fix. To mitigate such noise, researchers apply content filters to ignore implicated commits that update comments (f_2) or whitespace (f_3) [14].

Routine maintenance updates often modify a large number of files or lines of code (e.g., updates to coding style). Such commits are another source of noise in SZZ data. To mitigate the impact of this, researchers often filter out large commits. For example, McIntosh and Kamei [16] filter out commits that change more than 100 files or 10,000 lines (f_4).

(F 3) Apply Suspiciousness Filters: Commits that fix a large number of issues or induce a large number of fixes add noise to SZZ data. Developers routinely address issues one at a time, which is why multiple issues being fixed by a single commit is suspicious. Similarly, a commit that induces a large number of future fixes is suspicious, since it is unlikely that one change would be so problematic.

To filter out these suspicious commits, da Costa et al. [7] propose a framework. Their implementation of the framework for Apache projects uses the project-specific thresholds of the upper Median Absolute Deviation (MAD) [9] of the number of issues that a commit fixes (f_5) and the upper MAD of the number of fixes a change induces (f_6).

Filtering Ghost (FG): It is possible that, for a given defect-fixing commit, no implicated commits survive the filtering stages (f_1–f_6). We refer to these defect-fixing commits as Filtering Ghosts (FGs). FGs are problematic because no implicated commits can be associated with them. Thus, models that are trained using SZZ data will not be able to identify the commits that induce them.

3 Related Work

Fix-inducing changes have been the subject of considerable research. Since teams have limited resources, identifying changes are likely to be buggy can help with time and effort allocation. The SZZ approach [32] plays a crucial role in such allocation efforts. Below, we present the related work on fix-inducing changes and the limitations of SZZ.

3.1 Fix-Inducing Changes

The SZZ approach has been used to study properties of fix-inducing changes in several settings. For example, the seminal paper [32] used SZZ to study the day of the week when fix-inducing changes tended to appear. Eyolfson et al. [8] used SZZ to study the hour of the day when fix-inducing changes tended to appear. SZZ has also been used to detect and characterize defect-fix patterns [22], to study how long defects survive [4, 12], and to study the links between fix-inducing changes and (a) code authorship [25], (b) code clones [24], and (c) faulty defect fixes [41].

SZZ is also at the core of Just-In-Time defect prediction, a term coined by Kamei et al. [11], which describes a popular variant of change-level defect prediction. Mockus and Weiss [18] used various change properties to predict risky code changes at Bell Labs. Kim et al. [13] and Kamei et al. [11] expanded upon the set of metrics, including those that were computed using VCS and ITS data, and analyzed a broader set of projects, including swaths of open source and proprietary projects. Kononenko et al. [15] further expanded upon the metric set to include code reviewing data. JIT defect prediction has been deployed in industrial settings at Cisco [34], Blackberry [30], and Avaya [18] to name a few.

In recent years, as improvements to machine learning technology have appeared, JIT defect prediction has also been improved. To address the cold-start problem for software analytics, Kamei et al. [10] studied the efficacy of cross-project JIT defect prediction. Yang et al. [42] propose Deeper, which uses deep learning techniques to train JIT models.

While the prior work has made important contributions, it is built upon the underlying SZZ approach, which classifies changes as fix-inducing or clean. In this paper, we focus on risks to the completeness of SZZ data, quantifying and characterizing that risk in 14 open source ASF projects.

3.2 Limitations of the SZZ Approach

This paper is not the first to propose improvements to the SZZ approach. Table 1 presents an overview of past work studying SZZ limitations and improvements. Kim et al. [14] introduced an improvement to SZZ that uses annotation graphs as opposed to the annotate command. Moreover, the approach filters out style changes. Williams and Spacco [37, 38] proposed adding weights to the SZZ mapping technique, as well as using the DiffJ [5] tool to disregard formatting changes when comparing code files. Neto et al. [20] proposed an SZZ implementation that ignores refactoring changes, since those are unlikely to introduce defects. Contributing to this line of work, we propose several language-aware improvements to SZZ (see Section 6) to improve the completeness (recall) of SZZ-generated data.

Past work has also raised concerns about the risks of modelling defect data using SZZ. For example, da Costa et al. [7] evaluated several variants of the SZZ approach using suspiciousness filters based on the earliest defect appearance, the future impact of a change, and the realism of defect introduction. This work differs from that of da Costa et al. in that we focus on increasing the recall of SZZ data by defining ghost commits, and studying strategies to capture them.

Rodriguez-Perez et al. [26, 27, 28] introduce the concept of extrinsic defects to describe defects that should not have an implicated code change. There is an interesting interplay between the extrinsic/intrinsic classification proposed by Rodriguez-Perez et al. [26], which focus on the nature of the defects being fixed, and the ghost commit concept we propose in this work, which focuses on the commits that currently slip through the mapping and filtering stages of the SZZ algorithm. We study the relationship between extrinsic/intrinsic defects and ghost commits in Section 6.

4 Study Design

The goal of our study is to better understand the extent to which the ghost commit problem impacts SZZ data of

5. http://www.incava.org/projects/diffj

TABLE 1
An overview of past work addressing SZZ Limitations.

Paper	SZZ Limitation Addressed	Contribution
Kim et al. [14]	Non-semantic changes are identified as defect-fixing	Use an automated approach with annotation graphs
Williams and Spacco [38]	Annotation graphs are imprecise at tracking lines	Use a weighted line mapping approach to track unique lines
	Non-semantic changes are inaccurately identified	Use the DiffJ tool to ignore only non-semantic changes
Neto et al. [20]	Refactoring changes are flagged as defect-inducing	Introduce a refactoring-aware SZZ implementation
Da Costa et al. [7]	Techniques to evaluate SZZ-generated data are limited	Introduce a framework to evaluate SZZ-generated data
Perez et al. [26, 27, 28]	Not all defects are introduced by a specific commit	Introduce an approach to explore different causes of defects
This paper	Certain (ghost) commits are overlooked by SZZ	Quantify, characterize, and mitigate ghost commits

Fig. 3. The mitigation and maintenance type analyses performed for MG 1 commits

real software projects. In working towards these goals, we formulate three concrete research objectives

Objective 1: Quantification. Our first objective is to measure how often ghost commits occur. While Section 2 defines ghost commits, it is unclear if they occur often enough to be of concern for users of SZZ.

Objective 2: Characterization. Our second objective is to study the properties of ghost commits. Specific development activities may be disproportionately responsible for generating ghost commits. Knowing these tendencies may help researchers to propose solutions to and practitioners to avoid generating ghost commits.

Objective 3: Mitigation. Our final objective is to propose strategies to mitigate the ghost commit problem. Ideally, these will be extensions to the SZZ approach itself.

To tackle these objectives, we conduct an empirical study of repository data from open source projects. Figure 2 provides an overview of our study approach for Objectives 1 and 2, and Figure 3 provides an overview of our approach for Objective 3. Below, we present our rationale for selecting our subject projects (Section 4.1), as well as our approaches to data extraction and analysis (Sections 4.2 and 4.3), and ghost commit mitigation (Section 4.4).

4.1 Corpus of Software Projects

We study 14 projects from the Apache Software Foundation (ASF). Similar to Munaiah et al. [19], we identify criteria that must be satisfied by our subject projects.

Criterion 1: Replicability. We want to ensure that our study can be replicated (and extended) by researchers. To reduce barriers to access of the raw data, we select subject projects whose software repositories (VCS, ITS) are freely and openly available for download. To further enable replicability, we have made our data extraction and analysis scripts publicly available. [6]

Criterion 2: System Size and Activity. We want to study large, actively maintained projects, since such projects stand to benefit the most from SZZ analyses.

Criterion 3: VCS-ITS Linkage. Like all SZZ-based studies, a key concern is the quality of the links between commits (VCS) and issue reports (ITS). Thus, we select software projects where a large proportion of commits are explicitly linked to issue reports.

To satisfy Criterion 1, we select projects from the Apache Software Foundation (ASF). The ASF provides resources to support the development of software for the public good. The VCS[7] and ITS[8] of Apache are publicly available. Selecting ASF projects for analysis satisfies many of Munaiah et al.'s Community, Documentation, and License criteria for selecting engineered software repositories for analysis.

To satisfy Criterion 2, we select 14 of the most actively developed ASF projects for analysis. These 14 subject projects have been studied in prior work [7, 17]. Table 2 provides an overview of the 14 subject projects, and shows that the size of the projects ranges between 0.087 MLOC and 4.3 MLOC. Satisfying Criterion 2 also satisfies Munaiah et al.'s History criterion. Moreover, the selected ASF projects include unit tests (Munaiah et al.'s Unit Tests criterion) and use a Cloudbees (Jenkins) instance to perform continuous integration [9] (Munaiah et al.'s CI criterion).

To ensure that Criterion 3 is satisfied, we study the VCS-ITS linkage practices of ASF projects. We find that ASF developers tend to record the issue ID within commit messages following a clear pattern. For example, below is the commit message that accompanies commit ae90791 from the Apache PIG project:

"PIG-5118 Script fails with Invalid dag [...]"

We use regular expressions to extract these issue ID references. Thus, we compute the VCS-ITS linkage rate, i.e., the percentage of commits that are associated with issue reports.

Table 2 shows that we select a VCS-ITS linkage rate threshold of 50%. This threshold helps to satisfy Munaiah et al.'s Issues criterion. A sensitivity analysis shows that a threshold of 60% would result in five fewer projects, while a threshold of 40% would only add one project. Thus, we believe that the impact of this threshold choice is minimal.

6. http://doi.org/10.5281/zenodo.4558395
7. https://git.apache.org/
8. https://issues.apache.org/
9. https://builds.apache.org/

4.2 Data Extraction

(DE 1) Extract Issue Properties: The ASF uses the JIRA ITS. We use the JIRA REST API[10] to extract the identifier (IssueID), type (Type), and reported date (RepDate) for each referenced issue of the subject projects.

(DE 2) Extract Commit Properties: We first collect a copy of the Git VCS archive of each subject system. In the past, the ASF used Subversion as its primary VCS,[11] while providing read-only Git mirrors for convenience. Nowadays, several Apache projects now use Git as their primary VCS.

Next, we extract (meta) data about the commits that appear on the trunk branch. We focus on the trunk branch because it is the main development branch in ASF projects.

For each commit on a trunk branch, we extract three key properties: (1) the CommitID; (2) the commit message (to detect whether there is an IssueID encoded within it, and extract it if it exists); and (3) the list of modified files.

(DE 3) Remove Non-Code Changes: Since we want to study defects in subject system behaviour, we focus our analysis on commits to source code files. Thus, we filter out commits that only modify .txt, .xml, and CHANGELOG files.

4.3 Data Analysis

(DA 1) Analyze GC Frequency: To analyze the frequency of each ghost type, we compute: (1) the proportion of addition-only defect-fixing commits among all of the defect-fixing commits (MG 1); (2) the proportion of removal-only commits among all of the commits (MG 2); and (3) the proportion of defect-fixing commits whose fix-inducing commits are entirely discarded by the filtering phase (FG).

(DA 2) Analyze GC Root Cause: We then set out to better understand the types of changes that are associated with each type of ghost commit. To determine the types of changes that appear in ghost commits, we apply an open coding approach [5] to classify examples of each type of ghost commit. Since an understanding of the context of the studied system is required to code changes, we choose to select one project from our corpus of studied projects to perform open coding on rather than a broad sample of changes from several projects. We analyze a project with a "typical" rate of ghost commits, i.e., a project with a rate close to the median rate in our corpus. To obtain the categories used to categorize the MG, the first author independently classified the MG, defined the taxonomy based on observed patterns, and shared this taxonomy with the other authors, who provided feedback. To estimate the degree of subjectiveness in our classification process, the second author independently classified the same ghost commits using the revised taxonomy. We then use Cohen's Kappa, a coefficient that measures inter-rater reliability, to compute an agreement score between the codes of the first and second authors [6]. Finally, cases where coders disagreed were discussed in a follow-up meeting until a consensus could be reached. In those meetings, the third author would cast the tie-breaking vote if necessary.

10. https://developer.atlassian.com/server/jira/platform/rest-apis/

11. http://www.apache.org/dev/version-control.html

4.4 Mitigation Analysis

(M 1) Apply Data Flow Analysis: For MG 1, we apply the mitigation strategy (see Section 6.1) to the added lines to identify a list of lines to be mapped (and filtered) by SZZ.

If the commit that last modified a line is a refactoring change, we continue to trace backwards to find the commit that last made a non-refactoring change to the line, following the RA-SZZ approach introduced by Neto et al. [20].

We do not currently have an approach for handling breaking changes [35]; however, they did not present an issue for us in our analysis. We suspect that this issue was not prevalent because we focus our analysis on recent changes. Indeed, Tufano et al. find that breaking changes tend to be most prevalent in old commits, where dependencies on old versions of libraries and tools may present issues.

(M 2) Apply Baseline Approach: The baseline approach we compare to is a modified version of A-SZZ, introduced by Sahal and Tosun [29]. A-SZZ considers the lines between "the first left bracket above and the first right bracket below" the added lines as a code block, then runs the `blog` command on all the functional lines of the block to implicate commits. In cases where the syntactic A-SZZ definition of a code block cuts into another method or loop, we instead consider the block to be the first enclosing method or loop.

(M 3) Perform Comparative Analysis: After applying both the Data Flow Analysis and the Baseline Approach, we compare the sets of potentially bug-inducing commits implicated for each defect-fixing commit to count the instances where both techniques yielded the same results, and whether there are implicated commits in common for the cases where they yielded different results.

(M 4) Perform Precision Analysis: For each pair of fix-inducing and defect-fixing commits from both approaches, we take a deeper look at the fix-inducing commit to assess whether it should have been implicated in the fix. Since we are not subject matter experts in the studied systems, we take a conservative approach to labelling the implicated commits. We assume that implicated commits are correctly labelled (i.e., true positives) unless it is evident that they could not have contributed to the bug (i.e., false positives). We use this data to compute the precision score ($\frac{tp}{tp+fp}$) for each technique. Note that we exclude New Entity changes from this analysis due to their inherent ambiguity.

4.5 Maintenance Type Analysis

(MT 1) Perform Maintenance Type Analysis: To study the interplay between intrinsic/extrinsic defects [26] and ghost commits, we classify ghost commits by Swanson's maintenance categories [33], where: (1) corrective maintenance rectifies a processing, performance, or implementation failure; (2) adaptive maintenance responds to changes in the data or processing environments; and (3) perfective maintenance improves non-functional properties (e.g., performance, maintainability). Corrective maintenance maps onto the concept of intrinsic defects, while adaptive and perfective maintenance are likely due to extrinsic defects.

5 Quantification and Characterization

Below, we present the quantification and characterization results for the three types of ghost commits.

TABLE 2
An overview of the subject projects and Ghost Commits' frequency.

Project	Size (LOC)	Commits	Issues	Linkage $\frac{\text{Issues}}{\text{Commits}}$	MG 1 (%)	MG 2 (%)	FG (%)
ACTIVEMQ	0.087 mil	9,945	5,138	51.66%	7.55%	2.35%	5.58%
AMBARI	3.8 mil	23,901	23,346	97.68%	7.73%	1.55%	13.59%
CAMEL	2.6 mil	31,726	17,072	53.81%	9.44%	2.14%	6.64%
CAYENNE	0.563 mil	5,897	3,248	55.08%	5.91%	2.70%	5.43%
DERBY	1.4 mil	8,180	6,791	83.02%	7.74%	4.60%	9.41%
HBASE	1.3 mil	14,099	12,701	90.08%	7.42%	2.18%	6.43%
HIVE	2.6 mil	12,548	12,132	96.68%	9.23%	1.67%	14.49%
JACKRABBIT	4.3 mil	8,517	5,563	65.32%	7.33%	3.36%	0.35%
KARAF	0.281 mil	7,042	4,689	66.59%	10.45%	1.80%	1.77%
OPENJPA	0.837 mil	4,861	3,231	66.47%	6.56%	1.72%	9.02%
PIG	0.581 mil	3,152	2,932	93.02%	8.57%	1.05%	5.49%
QPID	0.246 mil	14,181	7,659	54.01%	7.42%	3.56%	2.13%
SLING	1.1 mil	21,668	12,168	56.16%	5.66%	2.38%	1.34%
THRIFT	0.466 mil	5,305	3,340	62.96%	11.72%	2.68%	1.06%
Mean	1.4 mil	12,216	8,572	70.90%	8.05%	2.41%	5.25%
Median	997 K	9,231	6,177	65.89%	7.64%	2.68%	5.46%

TABLE 3
The categories of MG 1. Percentages are of the overall sample unless indented to indicate category values. De nitions appear in Section 5.1.

Category	Number	Percentage
New Entity	6	4.2%
New Class	1	16.7%
New SubClass	4	66.7%
New Interface	1	16.7%
Check	66	46.5%
If Check	54	81.8%
Null Check	25	37.9%
Try/Catch Check	17	25.8%
Con guration	7	4.9%
Override	18	12.7%
Logging	12	8.5%
Expanding Class	64	45.1%

TABLE 4
The categories of MG 2. Percentages are of the overall sample unless indented to indicate category values. De nitions appear in Section 5.2.

Category	Number	Percentage
Cleanup	122	39.7%
Unused	38	31.4%
Unused Method	9	23.7%
Unused Con guration	4	10.5%
Unused Dependency	7	18.4%
Unused Class	9	23.7%
Unused Variable	9	23.7%
Redundant	4	3.3%
Duplicate	14	11.5%
Deprecated	8	6.6%
Renaming	3	2.5%
Refactoring	6	4.9%
Dead Code	3	2.5%
Entire File	46	37.7%
Undo	35	11.4%
Revert Entire Commit	14	40.0%
Partial Revert	21	60.0%
Update Settings	25	8.1%
Con guration	20	80.0%
Framework	5	20.0%
Logging	25	8.1%
Documentation	11	3.6%
Fix Race Condition	5	1.6%
Miscellaneous	84	27.4%

5.1 Defect Fixes with No Implicated Commits (MG 1)

Quanti cation: Mapping Ghost 1 is not uncommon among the studied projects. Table 2 shows that 5.66%–11.72% of all defect- xing commits are of type MG 1 (i.e., contain only added lines), with a median of 7.64%. Current implementations of SZZ cannot map MG 1 defect- xing commits back to commits that are implicated in the x.

Characterization: To gain insight into the characteristics of MG 1 defect xes, we manually code changes from the ACTIVEMQ project. We select ACTIVEMQ because its proportion of MG 1 xes is 7.55%, which is closest to the median value (7.64%). After the rst and second authors initially classify all 148 of the instances of MG 1 in ACTIVEMQ, we obtained an agreement score of = 0:314, which is considered to be fair agreement. In our follow-up meetings, several patterns of disagreements emerged, which were largely due to initial misunderstandings of the classi cation types. After the meetings, the coders came to a consensus on 145 commits, only requiring a tie-breaking vote for three commits. This suggests that the true agreement score is much greater than the one reported above.

Table 3 provides an overview of the categories of MG 1 that we discovered. New Entity changes involve either the addition of a New Class, New Subclass, or a New Interface. Checkchanges consist of branching upon checking certain conditions using if statements, try/catch statements, and/or assertions. We also record which of these changes are checks for special values like null . Con guration changes are those that update settings, such as changes to.properties les. Override changes involve overriding a method inherited from a superclass in a subclass. Loggingchanges add or edit code being used to log execution behaviour. Expanding Class changes add new functionality to an existing class.

Within our sample, we observe that Checktype changes occur the most. Of these, most (81.8%) consisted of if branching statements. Most often, these commits would add a check for null to improve the robustness of a method. For example, commit d92d3a8 xes issue AMQ-3782 by adding a check for null of reconnectTask .

5.2 Commits that Cannot be Mapped to Fixes (MG 2)

Quanti cation: Although MG 2 commits are less common than MG 1 commits, MG 2 commits still account for a considerable proportion of the change activity. Table 2 shows that 1.05%–4.60% of all commits are of type MG 2 (i.e., contain removed lines only), with a median of 2.68%.

Since MG 2 commits do not add lines that future changes can improve upon, current SZZ implementations cannot implicate MG 2 commits in future xes. Similar to MG 1, extensions to the SZZ algorithm that enable implicating MG 2 commits would likely improve the recall of the approach.

Characterization: To characterize MG 2 commits, we manually code commits from the HBASE project. We select HBASE because its proportion of MG 2 commits is 2.18%, which is the closest to the median (2.68%). After the rst and second author independently classi ed all 307 instances of MG 2, the initial agreement score was = 0.567, which is considered to be moderate agreement. During the follow-up meeting, all disagreements were resolved due to clari cations without requiring a tie-breaking vote.

TABLE 5
The filtering ghosts removed by each step of the filtering process.

	No Filter	Issue Date Filter		Content Filters					
		Date (f_1)		Comments (f_2)		Whitespace (f_3)		Size (f_4)	
	BFC	BFC	FG Drop %	BFC	FG Drop %	BFC	FG Drop %	BFC	FG Drop %
ACTIVEMQ	1,720	1,644	79.1%	1,644	0	1,628	16.67%	1,624	4.17%
AMBARI	9,904	9,892	2.75%	9,887	1.14%	9,797	20.59%	9,467	75.51%
CAMEL	2,635	2,499	77.71%	2,499	0%	2,465	19.43%	2,460	2.86%
CAYENNE	368	367	4.76%	364	14.29%	357	33.33%	347	47.62%
DERBY	1,520	1,412	75.52%	1,412	0%	1,394	12.59%	1,377	11.89%
HBASE	4,181	4,010	63.57%	4,006	1.49%	3,980	15.38%	3,912	40.24%
HIVE	4,631	4,622	1.34%	4,619	0.04%	4,592	4.02%	3,960	94.19%
JACKRABBIT	1,112	1,110	50%	1,110	0%	1,109	25%	1,108	25%
KARAF	847	844	20%	844	0%	833	73.33%	832	6.67%
OPENJPA	776	716	85.71%	716	0%	711	7.14%	706	7.14%
PIG	1,057	1,012	77.59%	1,010	3.45%	1,006	6.9%	999	12.07%
QPID	2,206	2,200	12.77%	2,198	4.26%	2,181	36.17%	2,159	46.81%
SLING	2,093	2,089	14.29%	2,087	7.14%	2,072	53.57%	2,065	25%
THRIFT	1,226	1,225	7.69%	1,223	15.38%	1,216	53.85%	1,213	23.08%
Median	1,620	1,528	35%	1,528	0.59%	1,511	20.01%	1,500.5	24.04%

Table 4 provides an overview of the categories of MG 2. Cleanup changes remove code that is not needed. As the name suggests, Deleting Entire File changes remove files from the VCS. Unused changes remove artifacts and code elements that are unused. Other types of Cleanup changes include Redundant, Duplicate, Deprecated, Renaming, Refactoring, and Dead Code. These Cleanup changes may induce future fixes if they are too aggressive, removing code that was still needed. Undo changes either Revert Entire Commit or Partially Revert a commit. Revert Entire Commit changes are unlikely to be fix-inducing, since these changes usually refer to undoing commits that were initially problematic. However, Partial Revert changes may induce future fixes, since undoing part of a commit is likely done by hand and may be prone to error. Updating Settings changes are the same as Configuration changes described under MG 1. Logging and Documentation changes are unlikely to induce future fixes, since they do not impact core system functionality. Race Condition changes, such as attempts to resolve deadlocks, may induce future fixes due to incomplete or incorrect fix attempts. We also consider Miscellaneous changes as unlikely to induce future fixes.

Table 4 shows that the largest proportion of MG 2 commits are Cleanup. Most often, these commits remove code that is no longer needed. For example, commit e5123cc removes the startCatalogJanitorChore method, which is believed to be unused.

Prior work [31, 40] studied revert commits in a variety of open source and proprietary settings, reporting that 1%–5% of commits are revert commits. We find a larger proportion (11.4%) of commits undo prior commits in our sample of MG 2 commits. We suspect that this is because 60% of our undo commits are not explicitly labelled as reverted (i.e., they were not produced using the git revert command). Since the prior work focuses on explicitly labelled revert commits, the most comparable figure in our study would be the 4.56% (= 40% × 11.4%) of MG 2 commits that Revert Entire Commits, which falls within the range of prior work. This suggests that the scope of revert commits is broader than previously analyzed. An analysis of non-explicit revert commits might be an interesting direction for future work.

Fig. 4. An example of a FG.

We observe that 1.95% of MG 2 commits are refactorings. This rate is similar to that of Tufano et al. [36], who observed that only 1.8% (= 9% × 20%) of removed instances of code smells were removed through refactorings.

We also observe that 9.4% of MG 2 commits remove entire classes and 9.4% of MG 2 type commits remove entire methods. In the context of removal of Self Admitted Technical Debt (SATD), Zampetti et al. [44] found that on average, 30.2% and 14.0% of SATD is removed by deleting entire classes and methods, respectively. We attribute this difference in the rates of entire class and method removals to our differing study contexts (i.e., all removal-only commits vs. SATD-removing commits). However, we believe that the results are complementary enough to indicate that large removal operations occur frequently enough to justify dedicated analysis approaches.

5.3 Defect-Fixing Commits with No Implicated Commits That Survive Filtering (FG)

Quantification: Filtering Ghosts make up a considerable proportion of changes among the studied projects. Table 2 shows that 0.35%–14.49% of defect-fixing commits are of type FG (i.e., have all of their implicated fix-inducing commits filtered out), with a median of 5.46%. Current implementations of the SZZ algorithm filter out all commits that are implicated by FG defect-fixing commits. Extensions to the SZZ algorithm that enable pinpointing other fix-inducing commits that could lead to FG defect fixes would again improve the recall of the approach.

Characterization: To characterize FG commits, we compute how many FG defect fixes are being removed by each filter f_1–f_4 (none of the FG commits in the studied projects are due to f_5 or f_6). Table 5 shows that across all studied projects, the largest proportion of FG commits is due to the date filter (f_1), with a median of 35%.

To investigate why so many FG commits are being removed by the date filter, we conduct a deeper inspection. We initially suspected that many of these FG would be due to inconsistencies in time-keeping between the VCS and ITS; however, this was not the case. Figure 4 provides an example of a FG (commit 3a356b5) from the PIG project. The SZZ approach implicates one potential fix-inducing commit IC (f22c685) in the future fix in commit BFC. However, the issue report that documents the defect (PIG-942) that is associated with BFC was created on Sept. 3rd, while the potentially implicated commit IC appeared later on Sept. 19th. Thus, IC is filtered out of the set of implicated commits for BFC. However, a comment on the issue report from Sept. 23rd explains that the initial fix attempt in commit IC contains problems that the later BFC commit addresses. In this case, the comment points out that IC introduces the potential for a null pointer exception, which is certainly a defect that matters for SZZ-based analyses.

> Mapping and filtering ghosts are not uncommon in ASF projects. Future SZZ implementations will likely benefit from mitigation of ghost commits.

6 Exploring Mitigation Strategies

In the prior section, we discovered that mapping and filtering ghosts do occur in the repositories of large and active ASF projects. In this section, we present the results of our mitigation analysis for MG 1 commits (the most frequently occurring type of ghost commit) and propose strategies for mitigating other types of ghost commits that should be explored in future work.

6.1 MG 1 Mitigation Analysis

Comparative Analysis: We find that data flow analysis implicated exactly the same commits as the baseline approach [29] in 15 of the 71 MG 1 commits from the ActiveMQ project (21.1%). A deeper examination of these implicated commits reveals that they mostly occur when the entire enclosing method was last modified by the same commit. For example, commit f7c7993 adds an if-check `if (from.equals(to))`. Our control flow analysis blames line 192 containing the enclosing method declaration `public static Converter lookupConverter(Class from, Class to)`, while A-SZZ blames all the lines in the method (192–206). In this case, the implicated commit is the same since the entire method was added by the same commit (1802116).

In cases where the lines immediately surrounding the added lines were last modified by different commits than the method/class declaration, the two techniques yield different results. Yet at least one common commit is implicated by both techniques in 41 of the remaining 56 cases (73.2%).

We are unable to implicate commits for non-Check New Entity changes and for 50% of Override changes. Nonetheless, our data flow analysis also reveals that at least one refactoring commit is incorrectly implicated as fix-inducing 46.5% of the time. This is due to an inherent shortcoming of SZZ and stresses the importance of implementing an automated Refactoring Aware SZZ implementation [20].

Precision Analysis: We find that our data flow analysis has a precision of 0.753, while A-SZZ has a precision of 0.358. One reason for this difference in precision is the data flow approach's ability to implicate lines outside the code block immediately surrounding the added lines. For example, commit d92d3a8 adds a null check for `reconnectTask` on lines 148–150. The data flow approach traces line 129, which updates `reconnectTask`'s value. This line is outside the `try` block surrounding the null check.

Another reason for the difference in precision is that the lines in the code block are often unrelated to the defect being fixed. This results in a higher rate of false positives.

> A context-aware, data flow based approach implicates commits more precisely than a purely syntactic approach.

6.2 MG 1 Mitigation Strategies

Broadly speaking, the proposed mitigation strategies require language-aware extensions to the SZZ approach. Below, we describe our approach to mitigate each of the categories of MG 1 from Table 3.

Algorithm 1 Null Check Mitigation

```
1: nullCheckVariable = variable being null checked
2: range = additionLineNumber scanSize
3: linesToTrace = fg
4: for line in range do
5:     if line contains nullCheckVariable then
6:         Append line to linesToTrace
7:     end if
8: end for
9: return szz(linesToTrace)
```

Check: For each Check-type MG 1 commit, we first locate the identifier being checked and identify the line(s) that introduce or modify its value. For example, commit 4adc8e4 from the ACTIVEMQ project adds a null-check for the `socketHandlerThread` identifier on lines 470–473. Data flow analysis reveals that `socketHandlerThread` was introduced on line 451, which we add to the list of lines to be processed by SZZ. In cases where the line introducing the variable are not in scope, we follow the A-SZZ approach, tracing the surrounding block.

A key limitation of the approach is its reliance upon a (heavyweight) data flow analysis rather than solely mining the software repositories. Semantic knowledge of the subject system (i.e., a context-aware approach) is required when analyzing a change and deciding which change introduced the identifier being checked, e.g., when blaming the method declaration instead of lines surrounding the modified code. Different SZZ users may have different needs depending on the cost of false negatives (i.e., the importance of mitigating ghosts) and false positives (i.e., the rate of false alarms).

Maintenance type analysis reveals that all Check-type MG 1 commits are corrective, which is expected since the addition of a check implies addressing an intrinsic defect. Algorithm 1 outlines our mitigation strategy for null checks, with a computational cost proportional to the breadth of the scanned area.

Algorithm 2 New Entity Mitigation

```
1: refLineNumber = line referring to new entity
2: range = refLineNumber scanSize
3: linesToTrace = fg
4: for line in range do
5:     Append line to linesToTrace
6: end for
7: return szz(linesToTrace)
```

New Entity: We propose an SZZ-inspired sub-approach, where other classes, methods, and variables which refer to the new entity refer are first mapped to the new entity through static analysis of the source code and then filtered based on their likelihood of leading to a defect. SZZ could then be applied to the filtered set of other entities to identify potential fix-inducing changes. For example, commit f6a5c7b adds the class XBeanFileResolver to help convert relative paths by verifying whether a provided path is a URL to an XBean file (boolean isXBeanFile(String configUri)). Our proposal would apply SZZ to call sites of this method. Algorithm 2 shows our mitigation strategy for New Entity changes, with a computational cost computational cost proportional to the breadth of the scanned area.

While this direction is exciting, a key limitation of this mitigation strategy is that the implementation would require an in-depth parse of the source code of a project. Current SZZ implementations only rely on lightweight parses of project source code (e.g., to identify irrelevant comment and whitespace changes). Adding this layer of complexity may be too costly to justify the benefits for all projects; however, for projects where the implications of false negatives are severe (e.g., safety critical systems), it may be worthwhile.

Another, less immediately concerning limitation is that the solution does not account for dynamic language features, such as reflection and dependency injection. Like any static analysis, the proposed solution would inherit the classic static analysis limitations. Hybrid static and dynamic analyses could be used to address these limitations, but would impose an even higher analysis cost.

When manually exploring this strategy in our sample, we find that four of the six New Entity commits also involve the addition of a check. In the example above, the new XBeanFileResolver class is immediately used by an if check in the same commit, which we can implicate using Algorithm 1. Maintenance type analysis reveals that five of the six New Entity changes are corrective (intrinsic), while the remainder are adaptive. Approaches to mitigate extrinsic defects [26] are important for New Entity changes.

Configuration: For Configuration changes, we must convey an even deeper understanding of the context of the change to the SZZ algorithm. For instance, an understanding of the properties being updated and/or the external tool/framework being called is needed to implicate changes in this category. To illustrate, consider commit 9c75fe7, which updates the JMSXUSER_ID message property so that that it appears when browsing the message via JMX. A deeper understanding of the JMX API would be required to implicate fix-inducing commits for this defect-fixing change.

This type of ghost commit requires deep investment in project-specific details, which may not transfer to other projects. Indeed, Configuration changes can be so specific to a niche that investigating them would require a complete understanding of the studied projects.

Complicating matters further, maintenance type analysis reveals that six of the seven Configuration changes are adaptive. This suggests that the bulk of configuration fixes do not have a commit to implicate. Given their relative infrequency and low rates of corrective maintenance, mitigation of Configuration ghosts are unlikely to yield much value.

Algorithm 3 Override Mitigation

```
1: overriddenMethod = method being overridden
2: range = class hierarchy threshold
3: linesToTrace = fg
4: for class in range do
5:     if class is superclass of overriddenMethod then
6:         Append overridden method declaration to linesToTrace
7:     end if
8: end for
9: return szz(linesToTrace)
```

Override: We propose applying SZZ to the superclass variant of the method being overridden. For example, commit 51ef021 overrides the getPercentUsage() method, which belongs to the StoreUsage subclass, to fix a bug. Our proposal would apply SZZ to the superclass variant from Usage. Algorithm 3 outlines our mitigation strategy for Override changes, with a computational cost proportional to the depth of the class hierarchy being searched.

A key concern with this solution is how quickly the set of implicated commits may grow. For complex hierarchies with several variants of an overridden method, the set of lines being fed to SZZ may quickly grow, essentially trading a false negative problem for a false positive one. To counter this, the range setting can constrain the search space.

In the example above, applying our strategy leads us to implicate commit 6d8e2c5, which originally added getPercentUsage() in the superclass. The method was later overridden in commit 51ef021 to add percentUsage = caclPercentUsage(), which refreshes the setting when retrieved over JMX.

Ten of the 18 Override changes also involve the addition of a check, where Algorithm 1 applies. In the example above, a null check of store is added to the overridden method.

Turning to the maintenance type, we find that six of the eight non-Check Override changes are corrective, and the remaining two are perfective. This suggests that extrinsic defects are not a large concern for Override ghost commits.

Algorithm 4 Logging Mitigation

```
1: loggingVariable = variable being logged
2: range = loggingLineNumber scanSize
3: linesToTrace = fg
4: for line in range do
5:     if line contains loggingVariable then
6:         Append line to linesToTrace
7:     end if
8: end for
9: return szz(linesToTrace)
```

Logging: We propose applying data flow analysis to determine where the value being logged, or the method containing the exception being logged, was last updated. Algorithm 4 outlines our mitigation strategy for Logging changes, with a computational cost proportional to the number of logging variables of interest and the breadth of the scanned area.

For example, commit 56bed30 adds a logging statement to log a start failure exception LOG.trace("Error

on start: ", e); . Applying our strategy to the catch statement where e is caught implicates commit 082fdc5 , where the catch block was added without logging. Similar to Algorithm 1, Algorithm 4 increases the complexity of SZZ by increasing the amount of static analysis required.

Ten of the twelve logging changes are contained within a Checkchange. In such cases, we implicate commits using Algorithm 1, and consider them to be corrective (intrinsic).

Algorithm 5 Expanding Class Mitigation

```
1: linesToTrace = fg
2: for line within expandedClass do
3:     if line last updated expandedClass then
4:         Append line to linesToTrace
5:     end if
6: end for
7: return szz(linesToTrace )
```

Expanding Class: An initial attempt may implicate the commit that last updated the expanded class as potentially x inducing. For example, commit 24f73a5 adds the method testReceipts to the StompTest class. The intuition behind our approach is that a limitation in the initial implementation or last update to the class may be implicated in this future x. Algorithm 5 outlines our mitigation strategy for Expanding Class changes, with a computational cost proportional to the size of the expanded class.

This approach is naïve, since the last change to a class may not be responsible for the expansion. Yet this same limitation is at the core of SZZ, i.e., the last edit to a line may not be truly responsible for introducing the defect [28].

We nd that all studied Expanding Class changes are corrective. These changes x defects by adding functionality that should have been added when the surrounding block was last updated. For example, commit 5f7a81f creates a copy of datasequence to x a race condition in the decompress method. We implicate commit 44bb9fb , which adds this method without accounting for the race condition. Commit c391321 xes a null pointer exception by adding return statements, while commit 4d0e572 xes a defect that is caused by the doRecoverNextMessages method not breaking out loops by adding break statements. What is striking about these examples is how distinct they are. An even ner grained analysis may be needed to propose mitigation strategies for each of these changes.

6.3 Promising Directions for Future Work on MG 2

To implicate MG 2 commits in future xes, we propose to track of program elements that were removed in a lookup table. This lookup table can be checked during the SZZ mapping phase. If program elements that were removed are re-added later, the lookup table can map defect- xing commits to the commits where the elements were removed.

A key limitation of this approach is the cost of creating and traversing the lookup table; however, we envision that a simple hash-like data structure could be ef cient. Perhaps of greater concern is the risk of false positives, when commits that reintroduce a program element have done so as a coincidence rather than an intentional resurrection of the previous code. To mitigate this risk, more heavyweight matching techniques (e.g., clone detection [2]) could be applied. This would increase the analysis cost (since entire program elements would need to be tracked and not just the identi er), but would likely reduce the false positive rate.

6.4 Promising Directions for Future Work on FG

While in theory, the null pointer exception discussed in Section 5.3 and its x should have been tracked under an independent issue ID, in our experience, this reuse of issue IDs is common developer behaviour. Indeed, Miura et al. [17] found that 5%–62% (median 29%) of work items across 14 studied systems are composed of two or more commits. Moreover, Park et al. [23] found that 22%–33% of resolved defects across three studied systems required more than one x attempt. Future SZZ extensions should take such behaviour into account to mitigate ltering ghosts.

Such ltering ghosts happen due to the inherent limitations of SZZ. A potential strategy to address multiple x attempts being linked to a single issue ID would be to relax the date-cutoff in the ltering stage of SZZ by specifying a date range, within which commits may be implicated. This way, commits made after the bug report creation date, but discussed in bug report comments may be considered. This approach would increase the total number of commits to be analyzed, and thus further increases the complexity of applying SZZ. A trade-off between the recall of SZZ and the resources needed to analyze the extra commits could be explored by varying the threshold of the date range.

7 Threats to Validity

Construct Validity: Construct threats to validity are associated with how closely our measurements re ect what we set out to measure. When linking VCS commits to ITS reports, we rely on developers recording the issueID within the commit message. However, developers may mistype or omit the issueID, which would introduce linkage bias [3] into our datasets. To mitigate the risk of linkage bias, we select a sample of projects where the linkage rate exceeds 50%.

We characterize ghost commits using an open coding approach. Since we are not developers of the studied projects, our understanding of the studied projects is limited. This surface understanding of the projects could introduce misclassi cation in our results. To mitigate this risk, two authors independently coded the samples.

Internal Validity: Internal threats to validity emerge when alternative hypotheses may also explain our observations. We argue that addition-only xes (MG 1) and removal-only commits (MG 2) present a risk for current SZZ implementations. However, it may be that these commits do not account for enough data to be of practical consequence. On the other hand, we observe that ghost commits account for a considerable proportion of the xes and the commits in the studied projects. Since their mitigation will improve the recall of SZZ approaches, the relative importance of addressing the ghost commit problem may depend on the importance of false negatives for the project(s) under analysis.

Developers may not create a new issue report for every defect. As we observed in our analysis of the ltering ghosts, follow-up work (e.g., minor defects in an initial patch) may be tacked onto the same issue ID as the initial commit. Future work should investigate how the SZZ approach can account for these patterns of use of issue trackers.

External Validity: External threats to validity are concerned with the generalizability of our results. We studied 14 open source projects from the Apache Software Foundation. Since these projects are primarily written in Java, our results may not generalize to other organizations or programming languages. However, the studied projects are of varying sizes (0.087 MLOC–4.3 MLOC) and span multiple domains (e.g., database management systems, content repositories).

8 Conclusions

Defects are introduced during software development. Identifying commits that are at risk of inducing future fixes can help teams to allocate quality assurance effort more effectively. To aid in identifying risky commits, the popular SZZ approach for identifying fix-inducing commits is used; however, the SZZ approach is not without limitations. In this paper, we focus on three types of ghost commits, i.e., commits that cannot connect to or from other commits. We conduct an empirical study of 14 Apache open source projects to quantify and characterize these ghost commits, observing that they occur regularly and share several common properties. Based on that characterization, we propose context-aware directions for the community to improve upon the SZZ approach to mitigate ghost commits.

Acknowledgements

Kamei was supported by JSPS KAKENHI Japan (Grant Numbers: JP21H04877, JP18H03222) and JSPS International Joint Research Program with SNSF (Project “SENSOR”).

References

[1] G. Bavota, B. D. Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, “When Does a Refactoring Induce Bugs? An Empirical Study,” in Proc. of the Int'l Working Conf. on Source Code Analysis and Manipulation, 2012, pp. 104–113.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, “Clone Detection Using Abstract Syntax Trees,” in Proc. of the Int'l Conf. on Software Maintenance, 1998, pp. 368–377.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, “Fair and Balanced?: Bias in Bug-Fix Datasets,” in Proc. of the Joint Meeting of the European Software Engineering Conf. and the Symposium on The Foundations of Software Engineering, 2009, pp. 121–130.

[4] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, “How Long Does a Bug Survive? An Empirical Study,” in Proc. of the Working Conf. on Reverse Engineering IEEE, 2011, pp. 191–200.

[5] K. Charmaz, Constructing Grounded Theory Sage, 2014.

[6] J. Cohen, “A Coefficient of Agreement for Nominal Scales,” Educational and Psychological Measurement, vol. 20, no. 1, pp. 37–46, 1960.

[7] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes,” IEEE Transactions on Software Engineering, vol. 43, no. 7, pp. 641–657, 2017.

[8] J. Eyolfson, L. Tan, and P. Lam, “Do Time of Day and Developer Experience Affect Commit Bugginess?” in Proc. of the Working Conf. on Mining Software Repositories, 2011, pp. 153–162.

[9] D. C. Howell, “Median Absolute Deviation,” Encyclopedia of Statistics in Behavioral Science, pp. 1193–1193, 2005.

[10] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, “Studying Just-In-Time Defect Prediction using Cross-Project Models,” Empirical Software Engineering, vol. 21, no. 5, pp. 2072–2106, 2016.

[11] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A Large-Scale Empirical Study of Just-in-Time Quality Assurance,” IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 757–773, 2013.

[12] S. Kim and E. James Whitehead, “How Long Did it Take to Fix Bugs?” in Proc. of the Int'l Workshop on Mining Software Repositories, 2006, pp. 173–174.

[13] S. Kim, E. James Whitehead, and Y. Zhang, “Classifying software changes: Clean or buggy?” IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 181–196, 2008.

[14] S. Kim, T. Zimmermann, K. Pan, and E. James Whitehead, “Automatic Identification of Bug-introducing Changes,” in Proc. of the Int'l Conf. on Automated Software Engineering, 2006, pp. 81–90.

[15] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, “Investigating Code Review Quality: Do People and Participation Matter?” in Proc. of the Int'l Conf. on Software Maintenance and Evolution, 2015, pp. 111–120.

[16] S. McIntosh and Y. Kamei, “Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-in-Time Defect Prediction,” IEEE Transactions on Software Engineering, vol. 44, no. 5, pp. 412–428, 2017.

[17] K. Miura, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, “The Impact of Task Granularity on Co-evolution Analyses,” in Proc. of the Int'l Symposium on Empirical Software Engineering and Measurement, 2016, pp. 47–57.

[18] A. Mockus and D. M. Weiss, “Predicting Risk of Software Changes,” Bell Labs Technical Journal, vol. 5, no. 2, pp. 169–180, 2000.

[19] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating Github for Engineered Software Projects,” Empirical Software Engineering, vol. 22, no. 6, pp. 3219–3253, 2017.

[20] E. C. Neto, D. A. da Costa, and U. Kulesza, “The Impact of Refactoring Changes on the SZZ Algorithm: An Empirical Study,” in Proc. of the Int'l Conf. on Software Analysis, Evolution and Reengineering, 2018, pp. 380–390.

[21] T. H. Nguyen, B. Adams, and A. E. Hassan, “A Case Study of Bias in Bug-Fix Datasets,” in Proc. of the Working Conf. on Reverse Engineering, 2010, pp. 259–268.

[22] K. Pan, S. Kim, and E. James Whitehead, “Toward an Understanding of Bug Fix Patterns,” Empirical Software Engineering, vol. 14, no. 3, pp. 286–315, 2009.

[23] J. Park, M. Kim, B. Ray, and D.-H. Bae, “An Empirical Study of Supplementary Bug Fixes,” in Proc. of the Working Conf. on Mining Software Repositories, 2012, pp. 40–49.