The Need for a Fine-grained approach in Just-in-Time Defect Prediction

Giuseppe Ng  
De La Salle University  
Malate, Manila  
giuseppe_ng@dlsu.edu.ph

Charibeth Cheng  
De La Salle University  
Malate, Manila  
charibeth.cheng@dlsu.edu.ph

ABSTRACT
With software system complexity leading to the rise of software defects, research efforts have been done on techniques towards predicting software defects and Just-in-time (JIT) defect prediction which predicts whether a code change is defective. While using features to determine potentially defective code change, inspection effort is still significant. As code change can impact several files, we investigate an open source project to identify potential gaps with features in JIT perspective. In addition, with a lack of publicly available JIT dataset that link the features with actual commits, we also present a new dataset that can be utilized in JIT and semantic analysis.

CCS CONCEPTS
• Software and its engineering → Software defect analysis; • Computing methodologies → Machine learning;

KEYWORDS
empirical software engineering, software metrics, defect prediction, just-in-time prediction, software defect prediction

ACM Reference Format:
Giuseppe Ng and Charibeth Cheng. 2021. The Need for a Fine-grained approach in Just-in-Time Defect Prediction. In Proceedings of Unpublished. City of Manila, Philippines, 14 pages.

1 INTRODUCTION
Risks in software defect have increased over time [1–6]. For instance, the Heartbleed bug was due to an OpenSSL vulnerability [6]. This vulnerability affected servers that utilized the library for secure connections such as Apache and Nginx, allowing an attacker to remotely read vulnerable server protected memory and potentially retrieve cryptographic information. The vulnerability affected an estimated 24-55% of popular HTTPS websites [7]. Such risks had lead towards the development of Software Defect Prediction (SDP) that predicts whether specific files in a software project are buggy or clean [3, 4, 6, 8–11]. Finding software defects remains to be extremely challenging as tools could yield false positives predictions [3, 12]. In addition, this technique, however, is coarse granularity level of prediction and focuses on files or packages that are defective [13, 14].

Mockus and Weiss proposed changing the focus from files and packages to submitted code changes into VCS. This technique is referred to as Just-in-Time Defect Prediction (JIT) [15–17]. Several works were done in the field of JIT and the crafting of features towards better performance [6, 10, 18–20].

When developers submit code changes to the repository, the developer submits a message regarding the change. The repository then generates the code diff [6]. A typical submitted code change can be seen in figure 1.

![Sample code commit.](image)

The sample code commit on figure 1 shows two basic sections. The first section pertains to information about the commit, such as author, the timestamp, and a brief description of the submission. The other section shows the changes in the code compared to the previous version. These are denoted by the file name and path, the line numbers, the '+' symbol for line additions and '-' symbol for line deletions [6].

Determining defective classes or files has the downside of lack of explainability of its defectiveness, leading to the inspection of code change as an alternate approach [2, 8, 19]. Building datasets is a major aspect in machine learning approaches [12, 21, 22]. SDP datasets are based on analysis of Version Control Systems (VCS) such as GIT, SVN, or CVS, with defect information extracted from bug ticketing system [4, 10, 11, 23]. Various traditional SDP datasets were made available online to aid in empirical software engineering and software defects prediction. These datasets were created using open source and closed source projects [18, 24–26]. The ReLink dataset uses the following projects: (1) Apache, (2) Safe, and (3) ZXing to predict whether a source code file is defective...
The AEEEM dataset is composed of the following open source projects: (1) Equinox (EQ), (2) JDT Core, (3) Lucene (LC), (4) Mylyn (ML), and (5) PDE UI. This dataset is intended to predict whether a class is defective or not [28]. MORPH utilized several open source projects such as: ant-1.3, arc, camel-1.0, poi-1.5, redaktor, skarbońka, tomatc, velocity-1.4, xalan-2.4, and xerces-1.2 to predict whether a class is defective [29]. Shepperd et al. released the NASA dataset that utilized closed source projects inside NASA for traditional software defect prediction [24]. SOFTLAB also used closed source projects for their dataset [5, 30]. The NASA and SOFTLAB datasets predicts defects on a function level [5, 24, 30].

When creating a dataset, in general, once the corpus has been identified, the predictor metrics need to be calculated [31]. The selection of traditional software metrics depends largely on the context and purpose defined [32]. In terms of context of traditional SDP, metrics describing a software’s size, complexity, and quality are critical [33]. CK metrics [34] and Lines of Code (LOC) are possible metrics that give different insight on the source code data [4, 11]. CK metrics measures the complexity such as number of sub-classes, depth of inheritance, number of methods to determine level of complexity [4, 34]. Nagappan and Bell showed that code churn, the measurement of a file changes over time, could be used as a feature [35]. Scattered changes in the code could lead to defective software [36].

Process metrics could be used as well in determining defective code [37]. Process metrics pertains to the use of information from source tracking repositories such as version control and issue tracking systems to capture development changes over time. These can be classified as: (1) code change, (2) developer-related metrics such as number of engineers, (3) code dependency, (4) organization-metrics such as organization volatility, (5) other process metrics [4].

The work by [18] builds from the findings of early works and provided a dataset that included several projects for JIT defect prediction. This dataset has been used in several JIT works [6, 10, 18–20]. The projects utilized for this are Bugzilla (BUG), Columba (COL), Eclipse JDT (JDT), Eclipse Platform (PLA), Mozilla (MOZ), and PostgreSQL (POS). The Bugzilla and Mozilla datasets were created from data provided by the Mining Software Repository 2007 Challenge. The Eclipse JDT and Platform data were created from the MSR 2008 Challenge data. Columba and Postgres were from their respective CVS repositories [18].

As a change-level defect prediction or JIT dataset, the granularity here is neither class, file, nor function. Instead, the features extracted were based on code change or commits in a code repository as explained in figure 1. The features extracted are grouped into several dimensions: (1) Diffusion, (2) Size of the change, (3) Fix metric, (4) Historical information regarding the change, and (5) Experience of developers [18, 20]. The features range from number of modified systems, files and directories, entropy, lines added and deleted, etc [10, 18, 19, 38]. Labeling a commit as bug inducing is done through the SZZ algorithm [18, 39]. In the case of Columba and PostgreSQL, since the defect identifiers were not included in the changelog or commit messages, an approximate SZZ approach was done where keywords such as “fixed” or “bug” would indicate a bug fix for a defect inducing commit [18].

A summary of the software metrics used for JIT are the following [18–20, 40]:

1. Number of modified subsystems;
2. Number of modified directories;
3. Number of modified files;
4. Entropy or the distribution of the modified code across each file;
5. Lines added;
6. Lines removed;
7. Lines of code in a file before the current change;
8. Whether the change is a bug fix;
9. Number of developers that changed the modified file;
10. The average time interval between the last and the current change;
11. The number of unique changes to the modified file;
12. Developer experience in terms of number of changes;
13. Recent developer experience; and
14. Developer experience on the subsystem [18].

The work by [6] also provided a dataset consisting of two open source projects: (1) Qt3, and (2) Openstack 4. These datasets were cleaned by McIntosh and Kamei [6, 41]. This dataset contained commit ids unlike other datasets that made it possible to identify defect inducing commits and look at the actual change log of such [6]. To our knowledge, there has been no other publicly available dataset that has this information. As such, it is clear that many JIT approaches rely on feature metrics such as [10, 18–20, 40, 42].

As conventional JIT approaches rely on feature metrics for defect prediction, we investigate whether code change feature metrics are adequate. In addition, we analyze the nature of code changes to show that inspection effort is still significant.

2 BACKGROUND

Software defects are as deficiencies in elements of a software because they do not meet requirements and specifications. Software faults are defined as the causes of the defect while software failure is defined as the end result of the defect [1, 43].

In terms of software quality, several factors are considered with regards to testing requirements beyond the functional requirements of software. Some of these are: domain specific, possibly based on external and internal components integrated into the system; and intended expertise of the end user [43].

Factors that lead to software defects are: human-related, lack of communication, insufficient design and testing, unrealistic timeframe, and defects from third party components, etc. [1]. A defect taxonomy can be used to effectively group different kinds of software defects. These are [44]:

1. requirements;
2. features and functionalities;
3. structural defects;
4. data;
5. implementation and coding;
6. integration;

1http://2007.msrconf.org/challenge/
2http://2008.msrconf.org/challenge/
3http://www.qt.io/
4http://www.openstack.org/
The Need for a Fine-grained approach in Just-in-Time Defect Prediction

(7) system software architecture;
(8) test definition and execution; and
(9) unclassified defects [44].

2.1 Software Defect Prediction

The goal of software defect prediction, then, is to identify the faults in the code and code modifications, estimate the number of defects and identify areas of optimization[11]. This technique falls under data-driven software engineering [45].

The learning models in SDP can generally be classified under supervised or non-supervised [1, 4, 11, 21, 22]. The model is expected to learn from a corpus of data that encompasses code archive (in classes or files form and source trees such as GIT, SVN, or CVS), and defect information (in a bug ticketing system such as Bugzilla or Jira) [4, 11, 12]. Linking source commit information and bug tickets may also be necessary. Granularity or instance of defect prediction can be by package, class, file, or function basis [10, 11].

Training data can be from the same project or referred to as 'Within Project Defect Prediction' (WPDP) or from other projects, 'Cross Project Defect Prediction' (CPDP) [11]. Traditional defect prediction models used WPDP project where older versions of the software are used to predict defects on newer versions [46].

The work by [31] outlines the basic workflow in designing a prediction model. This can be seen in figure 2.

![Figure 2: Generic Flow of Designing a Default Prediction Model](https://opendev.org/openstack/swift/commits/branch/master)

Figure 2: Generic Flow of Designing a Default Prediction Model

The traditional process in building the model involves extracting the features, and building the ML model that will be trained with this model. Given the instance whether it’s a package, a file, a class, or a method, the ML model should be able to classify the instance as defective [10, 12–14].

2.2 Just-in-Time Defect Prediction

Where traditional SDP have been focused on the predicting the coarse-grained instances as defective, others have expanded SDP to analyzing code changes instead of files to identify whether the change is defective. This is referred to as Just-in-Time (JIT) defect prediction [47, 48]. Figure 3 illustrates a typical code repository where all the commits are stored.

![Figure 3: Repository of Commits from Openstack.](https://opendev.org/openstack/swift/commits/branch/master)

Figure 3: Repository of Commits from Openstack.

For every code change in the code repository, features are extracted that are then used to train the model [18, 20]. Figure 4 shows the approach of a JIT defect prediction [10].

![Figure 4: Neural Network Architecture, from [10].](https://opendev.org/openstack/swift/commits/branch/master)

Figure 4: Neural Network Architecture, from [10].

Unlike traditional SDP, the granularity in JIT is actual code changes, achieving more fine-grained approach [49].

As discussed by Kamei [18], the diffusion aspect is represented by NS, ND, NF and Entropy. Highly distributed changes are more complex and are more likely to be defective. Number of modified subsystems, modified directories, and modified files using the root directory name, directory names, and file names respectively [18].

Entropy is calculated using a similar measure to Hassan [18, 36]. It is the measurement over time how distributed the changes are in the code base. Changes in one file is simpler than one that impacts many different files. Time period used for the dataset is 2 weeks [18]. Entropy is defined as [18]:

\[
H(P) = -\sum_{k=1}^{n} (p_k \times \log_2 p_k)
\] (1)
where probabilities $p_k \geq 0, \forall k \in 1, 2, ..., n$, $n$ is the number of files in the change, $P$ is a set of $p_k$, where $p_k$ is the proportion that $file_k$ is modified in a change and $\sum_{k=1}^{n} p_k = 1$ [18]. Figure 5 illustrates entropy in a sample scenario [50].

![Figure 5: Example of Entropy [50.]](image)

For time interval $t_1$, there are four changes and the probabilities are $P_A = \frac{2}{5}, P_B = \frac{1}{5}, P_C = \frac{1}{5}$. The entropy in $t_1$ is calculated as $H = - (0.4 \times \log_2 0.4 + 0.25 \times \log_2 0.25 + 0.25 \times \log_2 0.25) = 1$. For time interval $t_2$, entropy is 1.378 [50].

Lines added (LA), lines deleted (LD), and lines of code before the current change (LT) were calculated directly from the source code. LA and LD were normalized by dividing by LT while LT is normalized by dividing by NF since these two metrics have high correlation [18]. NUC is calculated by counting the number of commits that caused changes to specific files. This metric was also normalized by dividing by NF due to their correlation with the NF feature [18]. This technique follows findings made by Nagappan and Ball [35].

Age is calculated as the average of time interval between the last and current change of files [18, 20]. Should Files A, B and C were modified 3 days ago, 5 days ago, and 4 days ago respectively, Age is calculated as $4 \times \frac{3+5+4}{3}$. More detailed explanation of the features of the dataset can be found in [18].

Experience is the number of commits done by the author prior to the current commit while REXP is the total experience of the developer while factoring the recency of the change. Using $\frac{1}{n}$ where $n$ represents the year, a developer who made 5 changes 3 years ago would be represented as $\frac{5}{3}$. REXP is then computed as the summation of all such factored EXP values [18, 51].

### 2.3 SZZ Algorithm

Labeling defective changes is done through the SZZ algorithm [39]. Two key concepts in labeling is defining a bug inducing change and a bug fixing change. Bug inducing change is defined as a change A that caused for a change B to be submitted to rectify an issue introduced by change A. Change B is referred to as the bug fixing change [52]. An open source implementation was created by [51].

With regards to the Columbia and Postgres dataset, an approximated SZZ approach was taken due to the defect ids not referenced in the commit messages [18].

The overall workflow of the algorithm is as follows [51]:

1. Retrieve all closed bug reports from a ticket tracker of the system
2. Link the bug report to the bug fixing commits of the code repository
3. Identify the bug inducing commits through the bug fix commit information.

Phase 1 involves the linking of the bug reports to bug fixing commits while Phase 2 is the process of identifying the bug inducing commit. Phase 1 largely involves matching the commit to the closed ticket IDs [51]. In the even that there is no ticket to match, words such as fix in the commit message are assumed to be bug fixes [18, 51]. Phase 2 is illustrated in Figure 6 [51].

![Figure 6: SZZ Phases [51.]](image)

The tool will identify all commits that happened for specific lines using `git blame`. Each candidate is then ruled out by checking the date of the commit. If the change is newer but it was only a partial fix, then it is included as a bug inducing commit [51].

The end result of the algorithm is the list of submitted commits fixing bugs and potential list of changes as bug inducing. Using the SZZ tool provided by [51], pairs of bug inducing commits and bug fix commits can be identified from a dump of the bug tracking and report system and the git repository. The commits that are not labelled as defective from SZZ can be used to build the non-defective labelled dataset [51].

Subtle semantic differences would be difficult to identify from calculated metrics [3]. In JIT, there are limitations in feature metrics from code changes as two different code changes can lead to the same code metrics, thereby highlighting the need to extract features from semantic code [6].

### 2.4 Semantic Representation of Code Change

Although extraction of features from source code has been done for various purposes including SDP, the study of semantic features in JIT context is under-represented. [6] explored extracting features from semantic information of a code change. In their work, the code change, represented by the set of lines added and deleted, and the commit message were used in feature extraction. Figure 7 illustrates their approach [6].
The Need for a Fine-grained approach in Just-in-Time Defect Prediction

Their work shows that utilizing the semantic information from code changes can lead towards JIT [6].

2.5 Comparing Coarse-grained and Fine-grained JIT

A commit or code change can either be clean, completely bug inducing, or partially bug inducing. Traditional SDP where the model used for prediction is based on previous release data. In this scenario, the predictor can incorrectly label a file as defective. It can also completely miss defect inducing files [42]. Such an approach lacks feedback that developers need when doing code review [18, 38, 42]. Figure 8 illustrates how existing approaches are not adequate in identifying defective files within commits [42].

Regarding JIT defect prediction, coarse-grained JIT can predict whether an incoming code change is defect inducing. This approach, unlike the traditional SDP, can catch defect inducing changes as soon as it is made, thereby giving faster feedback to developers. However, the code inspection effort is still significant [38, 42].

A submitted code change can modify more than one file and can lead to significant inspection cost by developers to determine what particular area of the code change was defective. In addition, not all the files in the code change are necessarily defect inducing, and yet, such approaches could cause defect-free files in the commit to be defect inducing [42].

Hence, a finer grained approach towards JIT that is able to analyze changes within a code change could reduce the inspection cost. Ideally, this would entail identifying actual files that are defective and even the area within that file in the code change to be inspected. Figure 9 illustrates the difference of perspective of coarse-grained and fine-grained JIT.

3 METHODOLOGY

To determine whether feature metrics can determine the nature of defective code change, our first step is to create a dataset using an open source project as code repository and code change data are available. To build the dataset, we downloaded the source code repository and exported the defect data from an open source project. This is consistent with the work done by [18, 51]. The inclusion
of the commit id in the dataset is consistent with the work done by [6]. We then investigated the marked bug inducing commits of the project. Specifically, we examine the nature of code changes and report our observations in relation to the calculated feature metrics.

To see the predictive performance of the dataset, we use a Within-Project Defect prediction approach, a practice that Google used with known existing models [55–57].

To validate our findings, we referred to the dataset provided by [6] to see if the bug inducing commits in the dataset are consistent.

4 CREATING AND LABELLING A DATASET

We downloaded the git source repository of the Trac project6 and also obtained an export of their list of bugs7. The export covers tickets from August 24, 2003 to January 13, 2020.

Using the SZZ tool provided by [51], pairs of bug inducing commits and bug fix commits can be identified from a dump of the bug tracking and report system and the git repository. The software metrics from the project was calculated using the scripts provided.

There are 12,840 commits processed in the repository. The SZZ tool labelled 1681 commits as bug inducing. 1552 bug fixes were found by the tool. The information is shown in table 1.

Table 1: Summary of the information on the Trac dataset

| Project | Period       | #change | %Defect induced change |
|---------|--------------|---------|------------------------|
| Trac    | 08/2003-01/2020| 12840   | 13.09%                 |

We used cloc software8 to calculate the lines of code (LOC) of the project. This is used to determine the size of the project being processed. Table 2 illustrates the statistics of the code.

Table 2: CLOC Summary of the Trac Project

| Language   | files | blank | comment | code   |
|------------|-------|-------|---------|--------|
| Python     | 323   | 16624 | 23054   | 74407  |
| HTML       | 73    | 1126  | 32      | 9984   |
| CSS        | 17    | 294   | 185     | 4206   |
| Javascript | 25    | 261   | 327     | 2346   |
| DTD        | 1     | 200   | 214     | 564    |
| make       | 1     | 201   | 93      | 557    |
| PowerShell | 1     | 98    | 114     | 209    |
| YAML       | 2     | 7     | 5       | 207    |
| CoffeeScript| 2       | 22    | 50      | 93     |
| DOS Batch  | 1     | 16    | 0       | 79     |
| Bourne Shell| 1   | 13    | 179     | 59     |
| Bourne Again Shell| 2 | 4     | 16      | 37     |
| XML        | 1     | 0     | 0       | 12     |
| Total      | 450   | 18866 | 24269   | 92760  |

With this information we had managed to extract the features shown on table 3.

Table 3: Features of the Trac dataset

| Metric                        | Description                                    |
|-------------------------------|------------------------------------------------|
| Commit                        | The commit hash in the repository              |
| NS                            | Number of modified subsystems                  |
| ND                            | Number of modified directories                 |
| Entropy                       | Distribution of the modified code across each file |
| Lines_of_Code_Added           | Lines added                                    |
| Lines_of_Code_Deleted         | Lines deleted                                  |
| Files_Churned                 | Number of modified files                       |
| Lines_of_Code_Old             | Lines of code in a file before the current change |
| Purpose (Fix)                 | Whether the change is a bug fix                |
| Number_of_Authors (NDEV)      | Number of developers that changed the modified file |
| AGE                           | The average time interval between the last and the current change |
| Number_of_Uncertainchanges    | The number of unique changes to the modified file |
| EXP                           | Developer experience in terms of number of changes |
| REXP                          | Recent developer experience                    |
| Defective                     | Whether this is bug inducing or not           |

There are some deviations from the feature extraction methods described in [18]. We highlight the notable differences in the calculations in the tool provided in [51].

Entropy, Experience, and REXP are calculated similarly with Kamei [18]. Lines of code added and deleted are calculated by extracting the number of lines inserted and deleted respectively and dividing them by the total lines of code of the new file [51]. Lines of code of the old file is not normalized unlike LT feature in Kamei’s dataset [18, 51].

Age is calculated as an average of the time difference of every file modified by the current commit. Number of unique changes is calculated as the number of commits for every file modified by the commit [51].

For the normalization on the dataset, we employ the min-max standardization from [10]:

$$Norm(x) = \frac{x - \min(x)}{\max(x) - \min(x)}$$

This has been a commonly used technique to treat datasets in other works [10, 58].

4.1 The Trac Dataset

In this section, we report on interesting observations that we had seen from the dataset.

Commit ID. Our key inclusion in this dataset is the Commit ID for every submitted code change of the dataset. With this, it is
possible to look into each and every change that was marked as bug inducing for further study. Many JIT datasets do not include this, making the change level semantic learning on change-level defect prediction much more challenging.

**Lines of Code Added and Deleted.** We compare the lines of code added and deleted and found an interesting difference between the maximum lines of code added and deleted. The maximum lines of code added is 1175 while the lines of code removed is 468343. In analyzing the increase of lines of code removed, we noted that there is a commit that removed almost all the files in the repository as a repository clean up. The presence of this submission is an outlier in the repository.

**Files Churned.** We observed that since the tool already normalizes the number of modified files by the total number of files, the values are already in the range of 0 to 1. There are 39 instances when the values are not normalized due to potential divide by 0 issues that could happen when the submitted code change cleans up the repository. The minimum and maximum values of the feature are 0 and 739 respectively with an average of 0.1175 and a standard deviation of 6.5236.

**Lines of Code before the Change (LT).** The feature is not normalized and has an average of 141.7006 with a standard deviation of 326.9681. We found that over time, there is a general increasing trend in this value as shown in figure 10.

**Number of Modified Subsystems.** Upon applying the normalization step, we observed that over time, there is an increasing trend in the values. Figure 11 illustrates this.

This would make sense that over time, the system complexity would grow as more features are added in. The minimum and maximum values of the feature are 0 and 198 respectively with an average of 0.1175 and a standard deviation of 33.9173.

**Number of Modified Directories.** The minimum and maximum values of the feature are 0 and 8 respectively with an average of 1.5326 and a standard deviation of 1.5488. These values are raw and not standardized.

**Purpose.** In identifying whether a code change is a bug fix, the feature is represented as a boolean value. In the entire 12,840 commits, the tool marked a total of 2318 bug fixes representing 18.05% of all code commits to the repository.

**Entropy.** In terms of entropy, we found that the average entropy is 1.2246 with a standard deviation of 2.0082.

**Experience and Recent Experience.** We found that the average is 1145.5416 with a standard deviation of 1014.5913. We found that the huge code clean up influenced the values of the feature by plotting the chart that is shown in figure 12.

Analyzing the chart, shows an increasing trend on experience over time. However, because the code clean up removes many files, this also eliminates the experience that is tracked by looking at those specific files.

With the recent experience chart, the trend of the feature shows that the huge dip from the clean up does not influence the feature. 

**Number of Unique Changes (NUC).** The feature has an average of 2.0177 and a standard deviation of 3.9729. Interestingly, the maximum value is 180. This shows that this is an extreme outlier and suggests that standardization techniques may not be advisable.

**Defective.** This is the label of the dataset and marks a particular code change as defective.

Figures 14 and 15 show the histogram and the scatter matrix visualization of the dataset.

The statistics of the features of this dataset are summarized in table 4.
In training an actual model to see its potential, we applied a 70-30 split of seen and unseen data into a 9 layer architecture of ReLU units and a sigmoid output. Our data preprocessing techniques were normalization and undersampling. We utilized Smooth L1 Loss as our loss function with 3500 epoch training. An Adam optimizer with a learning rate of 0.001 was used.

By repetitively removing features from the training and checking the loss and recall, we identified that lines of code added, lines of code deleted, modified subsystems, modified subdirectories, experience, exp, age, number unique changes may be a potent starting set of features for training.

The selection of modified subsystems, age, and number unique changes as good features is in agreement with the results from [18]. The use of experience metrics such as experience and exp is consistent with the findings from [15, 42, 59] where experience-based metrics are good features.

Interestingly, our tests show that lines of code added, lines of code deleted, and modified subdirectories may be included as well in the list of good features. Our tests managed to attain a recall of 64.15% with a loss of 0.038 in a WPDP approach.

### 4.2 Observations of the Trac repository code changes

We would like to make note that in this study, we do not claim to be experts of the Trac source project. The analysis below is purely based on the information we had extracted from the repository.

In observing bug inducing changes and bug fixes, we make note that bug fixes tend to modify lines that were added by bug inducing changes. In doing code comparisons of defective code, figure 16 shows a defect inducing code change from the Trac repository.

In contrast, a sample bug fix code change is shown in figure 17.

To identify defect inducing lines, we compare the bug inducing change and the bug repair change for lines that were modified. Lines that were modified from the bug inducing change to the bug fix are defect inducing, while lines that were untouched are not defect inducing. This is consistent with the works by [60, 61].

The defect inducing change here only changes 1 file with 14 line additions and 13 line deletions and is a bug fix to another reported bug. This code change was submitted October 8, 2005. The bug fix for this was submitted May 12, 2010, 10 years after the original code change was accepted in the system. Clearly, this defect inducing change had already been released into actual stable versions of Trac.

Notice also that figure 17 also deleted lines 183 and 184, replacing them with 186, and 187. The lines were also in figure 16, however, 

---

**Table 4: Statistics of the Trac Dataset**

| Feature      | Min - Max | Ave   | Std - Dev |
|--------------|-----------|-------|-----------|
| LA           | 0 - 1175  | 6.8046| 20.4584   |
| LD           | 0 - 468343| 40.6886| 4133.1486 |
| Files Churned| 0 - 739   | 0.1175| 6.5236    |
| LT           | 0 - 1429  | 141.7008| 326.9681 |
| NS           | 0 - 198   | 0.1175| 33.9173   |
| ND           | 0 - 8     | 1.5326| 1.5488    |
| Entropy      | 0 - 8.4429| 1.2246| 2.0082    |
| Exp          | 0 - 3491  | 1145.5415| 1014.5912|
| NUC          | 0 - 180   | 2.0177| 3.9729    |

---

9Commit hash f1ab6d93ddcbdb043f9bfa4fca5b5e0bdebbde87313b
10Commit hash 936593f0a75421c43f2a0b0824801f90c8978c8
the change itself did not introduce these lines. Instead, it is introduced by another bug fix\textsuperscript{11} submitted to the repository. This implies that these lines were also defect-inducing when added to a different code change in the repository. The defect itself may have been caused by shifting context over time, possibly by code or by the dependent library environment or extrinsic bugs as defined by \cite{62}. We make note that a single bug fix change can be linked to several defect inducing code changes. It is possible also that defect was introduced by the perceived understanding of a developer of the code as pointed out by \cite{63}.

Refactor code changes\textsuperscript{12} are among the defect inducing changes. This is shown in figure 18.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure16.png}
\caption{Relevant Snippet of Defect Inducing Change.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure17.png}
\caption{Relevant Snippet of the Bug Fix.}
\end{figure}

In such works, large code changes and new files are included. A total of 205 lines from the new file was added in the shown code change. Should this file be the defect inducing file in the code change, there is still a significant amount of inspection effort to be done by the developer.

In analyzing the code change, we compare the feature metrics to the calculated average from our previous table 4. Table 5 shows the comparison.

\begin{table}[h]
\centering
\caption{Comparison of the Code Change to Average}
\begin{tabular}{|l|c|c|}
\hline
Feature & Code Change & Ave \\
\hline
LA & 4.67 & 6.8046 \\
LD & 4.33 & 40.6886 \\
Files Churned & 0.003 & 0.1175 \\
LT & 3 & 141.7008 \\
NS & 1 & 0.1175 \\
ND & 1 & 1.5326 \\
Entropy & 0 & 1.2246 \\
Exp & 812 & 1145.5415 \\
NUC & 1 & 2.0177 \\
\hline
\end{tabular}
\end{table}

This shows that the calculated feature metric for LA and LD being below the average. Making note that Kamei et al. \cite{18} made

\textsuperscript{11}Commit e955cc988b14d4e8585186dd50d2db31ce8327444

\textsuperscript{12}Commit 1fb1d30d38d2f8fa50d8fd704693f8904d10d8bd
mention that a large code changes (e.g. lines added (LA), deleted (LD), and entropy) could imply defective code change, the defect inducing code change is actually below the average for the Trac dataset.

With the large code base of this project at 92760 LOC, locating defective code requires significant inspection effort [10]. Identifying specific lines and repairing them would need bug reports. The presence of such error means that the defect had already manifested to actual end users [60, 64].

As code changes can modify many files, we examined the number of files modified per commit and found that on average, bug inducing commits modify 7.97 files. On average, the number of different file types modified by such commits is 1.72 file types. From the analysis of the modified files, there are 39 known files that have been modified. Some files do not have file extensions and have been marked as “no extension” instead. We have listed down the top file extensions modified on average on table 6.

Table 6: File extensions modified in Bug inducing changes in the Trac repository

| File Extension | Average #changed |
|----------------|------------------|
| py             | 5.08             |
| html           | 0.97             |
| no extension   | 0.43             |
| cs             | 0.39             |
| css            | 0.55             |
| po             | 0.19             |
| js             | 0.12             |
| pot            | 0.10             |
| txt            | 0.09             |
| png            | 0.09             |

Noting that there are potentially multiple files impacted by a specific change, identifying a commit as bug inducing does not identify which files are defective. The developer who utilizes JIT will still need to study the commit to determine what about the code change is causing software defects. Noting that in figure 19 a particular code change can impact multiple file types such as .cs and .py, diagnosing the bug can be complex and time-consuming. As noted by [18], inspecting possible defective files would take a considerable amount of time. Even if we utilize Effort-Aware works as suggested by [10, 18], if there is a need for the developer to study a large change, he would have to devote time and effort to understand the context of the code to determine how to fix the bug. Coarse grained approaches hinder their use from a practical standpoint [42].

Our results show that not only does changes in commits modify multiple files, but also changes several different file types. With our dataset including the commit ID, we are able to look into the commits marked bug inducing by the SZZ tool unlike many other available JIT datasets. One such example found in our dataset is shown in figure 19. The diff below illustrates two of the 46 files changed in the commit.

Figure 19: Sample code commit showing multiple different file types modified.

In analyzing other code changes, we found code changes that had large number of lines inserted and deleted. However, the code change was apparently simply adding copyright and modifying the copyright year. Figure 20 shows one of these code changes.

Figure 20: Sample code commit showing copyright changes.

The code change itself modified 83 files and had 646 lines added and 34 line deletions. Although our experiments showed that lines of code added, and lines of code deleted were good features to use in predicting defective code change, further look into the metrics showed that this metric can be misleading. According to Kamei et al., large changes are more likely to be defective (e.g. lines of code added, and entropy) [18], as the code change itself is only adding comments and updating copyright and comments, this might lead
to false positive results. Also, in other works, comments have been
discarded as noted in [6, 65, 66]. Comments are not typically part of
the executing part of the code [66]. The amount of inspection
effort would largely go to waste for these kind of commits.

At the same time, we found large code changes that make coarse-
grained JIT largely unhelpful in reducing the inspection cost. Figure
shows a large code change 15 that served to significantly rewrite
areas of the code.

---

Figure 21: Sample code commit linked with many bug fixes.

As code changes can involve many files, a code change can be
marked as 0061c0ce4443395a258a99c70e1a7d0435e8a460a
and 00dd97c7e9a852f4a8dfe8a460a
that add new features. The code change 16, the code change is a
combination of python files and XML. Figure 23 shows code change
adding new files into the repository.

The changes are modifications of existing files, and so, there are
lines removed and new lines added. This code change 17 modifies 6
files with 163 lines added and 110 lines deleted. In marking the file
within the code change as defective, the file itself is 822 lines and is
shown in figure 24.

With no explainable reason for what is defective, labelling code
changes and even a specific file as defect inducing is quite complex
for the developer to inspect. This is consistent with the insights
from [5] and [8].

With the datasets provided by Hoang et al. [6] showing that
bug inducing commits can span across many files and multiple file
types, we believe that our findings from the trac dataset have been
confirmed. Labelling a code change as defective alone may still have
significant inspection effort from the developer to gauge which file
or files caused the bug given a specific code change.

5 ANALYSIS

Relying solely on the feature metrics alone to classify code change as
defective. There have been instances when the code change is very
small and yet is defect inducing. The use of semantics to capture
such nuances could provide better level of detail. For example,
using the code itself as input can allow the classifier to ignore code
comments. This mirrors sentiments from [3].

In addition, classifying a code change as defective may not pro-
vide enough context as to what could be defective in that commit.
There is an associated inspection cost towards code changes marked
as defective [18]. Although, there has been development towards
Effort-Aware researches in JIT to address this, the approach used is
by sorting the changes accordingly by inspection cost [10].

As code changes can involve many files, a code change can be
partially defective or completely defective. Using coarse-grained JIT
has inherent inaccuracies in classifying an entire code change as
defective. A finer-grained approach such as files within a code
change can provide better accuracy level. This is consistent with
findings from [42].

---

\[16\] Commit hash 788944cebe5f599a396480653801360897b27d20

\[17\] Commit hash 788944cebe5f599a396480653801360897b27d20

---
Considering, however, the code change in figure 21 that is a **dormant bug** and is linked to many bug fixes, identifying the file itself as defective may not be enough. From the code changes we have shown in our analysis, it is clear that defect inducing changes are complicated. Labelling a code change as defect inducing is too simplistic a perspective. In addition, as a bug fix may end up modifying lines that were introduced by multiple code changes, the origin of a software defect could be from the combination of various code changes itself and not from a single one.
We have presented a new dataset that can be used for JIT research which specific file in the change as defective. Second, publicly available JIT datasets do not include the commit information that can be used to link the actual code change diffs to extract semantic features.

For future works, assembling more JIT datasets with the commit identifiers intact will allow for more diverse perspectives in progressing both JIT and semantic learning in JIT. By also looking deeper into semantic-based approaches in traditional SDP such as identifying which specific file in the change as defective. Second, publicly available JIT datasets do not include the commit information that can be used to link the actual code change diffs to extract semantic features.

REFERENCES
[1] J. Ge, J. Liu, and W. Liu, "Comparative study on defect prediction algorithms of supervised learning software based on imbalanced classification data sets," Proceedings - 2018 IEEE/ACIS 19th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2018, pp. 399–406, 2018.
[2] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," Empirical Software Engineering, 2018.
[3] A. Ghose, S. W. Ng, T. Tran, T. T. M. Pham, J. Grundy, and H. K. Dam, "Automatic feature learning for predicting vulnerable software components," IEEE Transactions on Software Engineering, vol. 14, no. November, pp. 1–12, 2018.
[4] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," IET Software, vol. 12, no. 3, pp. 161–173, 2018.
[5] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous Defect Prediction," IEEE Transactions on Software Engineering, vol. 44, no. 9, pp. 874–896, 2018.
[6] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepBT: An end-to-end deep learning framework for just-in-time defect prediction," IEEE International Working Conference on Mining Software Repositories, vol. 2019-May, pp. 34–45, 2019.
[7] Z. Durumeric, M. Payer, V. Paxson, J. Kenten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amanan, and J. Beekman, "The Matter of Heartbleed," in Proceedings of the 2014 Conference on Internet Measurement Conference - IMC '14, (New York, New York, USA), pp. 475–488, ACM Press, 2014.
[8] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C. J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," IEEE International Working Conference on Mining Software Repositories, vol. 2019-May, pp. 46–57, 2019.
[9] L. H. Son, N. Pritam, M. Khari, R. Kumar, P. T. M. Phuong, and P. H. Thong, "Empirical study of software defect prediction: A systematic mapping," Symmetry, vol. 11, no. 2, 2019.
[10] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," PLoS ONE, vol. 14, no. 2, pp. 1–19, 2019.
[11] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," IEEE Transactions on Software Engineering, vol. 45, no. 2, pp. 111–147, 2019.
[12] M. Rahman, Y. Watanobe, and K. Nakamura, "Source Code Assessment and Classification Based on Estimated Error Probability Using Attention LSTM Language Model and Its Application in Programming Education," Applied Sciences, vol. 10, no. 0, 2020.
[13] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, no. 1, pp. 17–26, 2015.
[14] AlAbbadi, "A Deep Ensemble Learning Method for Effort-Aware Just-In-Time Defect Prediction," Future Internet, vol. 11, p. 246, nov 2019.
[15] A. Mockus and D. M. Weiss, "Predicting risk of software changes," Bell Labs Technical Journal, vol. 5, no. 2, pp. 169–180, 2000.
[16] Q. Huang, X. Xia, and D. Lo, "Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-In-Time Defect Prediction," in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSM), pp. 159–170, IEEE, sep 2017.
[17] L. Cai, Y. R. Fan, M. Yan, and X. Xia, "Just-in-time software defect prediction: literature review," Ruan Jian Xue Bao Journal of Software, vol. 30, no. 5, pp. 1288–1307, 2019.
[18] Y. Kamei, E. Shuhab, 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.
[19] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-Aware just-in-Time defect prediction: Simple unsupervised models could be better than supervised models," Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, vol. 13-18-Nove, pp. 157–168, 2016.
[20] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," pp. 72–83, 2017.
[21] R. Ozakinci and A. Tarhan, "Early software defect prediction: A systematic map and review," Journal of Systems and Software, vol. 144, pp. 216–239, 2018.
[22] C. Catal and B. Dir, "A systematic review of software fault prediction studies," Expert Systems with Applications, vol. 36, no. 4, pp. 7346–7354, 2009.
[23] Q. Liu, Z. Xue, L. Liu, H. Fan, B. Du, and Y. Qian, "Generating committee messages from diffs using pointer-generator network," IEEE International Working Conference on Mining Software Repositories, vol. 2019-May, pp. 299–309, 2019.
[24] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208–1215, 2013.
[25] A. Lamanjii, J. Pérez, and S. Demeyer, "The eclipse and mozilla defect tracking dataset: A genuine dataset for mining bug information," IEEE International Working Conference on Mining Software Repositories, pp. 203–206, 2013.
[26] A. Pietri, D. Spinellis, and S. Zacchiroli, "The Software Heritage Graph Dataset: Public software development under one roof," in ASR 2019: The 10th International Conference on Mining Software Repositories, pp. 138–142. IEEE, 2019.
[27] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering Links between Bugs and Changes," Proceedings of the 19th ACM SIGSOFT symposium and the 15th European conference on Foundations of software engineering - SIGSOFT/FSE '14, p. 15, 2011.
[28] M. D’Ambros, M. Lanza, and R. Robbes, Evaluating defect prediction approaches: A benchmark and an extensive comparison, vol. 17. 2012.
[29] F. Peters and T. Menzies, "Privacy and utility for defect prediction: Experiments with MORPH,” Proceedings - International Conference on Software Engineering, pp. 189–199, 2012.
[30] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," Empirical Software Engineering, vol. 14, no. 5, pp. 540–578, 2009.
[31] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S.Panichella, "Defect prediction as a multijobective optimization problem," Software Testing, Verification and Reliability, vol. 21, pp. 426–459, jun 2015.
[32] J. Voas and R. Kuhn, "What Happened to Software Metrics?,” Computers, vol. 50, no. 5, pp. 88–98, 2017.
[33] S. H. Kan, Metrics and Models in Software Quality Engineering. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.
[34] S. R. Chidamber and C. F. Kemerer, "A Metrics for Object Oriented Design,” 1994.
[35] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," Proceedings - 27th International Conference on Software Engineering, ICSE'05, pp. 284–292, 2005.
[36] A. E. Hassan, "Predicting faults using the complexity of code changes," Proceedings - International Conference on Software Engineering, pp. 78–88, 2009.
[37] R. Moser, W. Pedrycz, and G. Succi, "A Comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," Proceedings - International Conference on Software Engineering, ICSE05, pp. 284–292, 2005.
[38] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug report generation support human developers? Findings from a Google case study," Proceedings - International Conference on Software Engineering, ICSE 2008, pp. 181–190, 2008.
[39] J. Slawski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?,” Proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–12, 2005.
[40] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "MULTI: Multi-objective effort-aware just-in-time software defect prediction," Information and Software Technology, vol. 93, no. August 2017, pp. 1–13, 2018.
References

[1] S. McIntosh and Y. Kamei, "Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-in-Time Defect Prediction," in Proceedings of the 4th International Conference on Software Engineering ICSE, 18, (New York, NY, USA), p. 560, Association for Computing Machinery, 2018.

[2] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," Journal of Systems and Software, vol. 150, pp. 22–36, 2019.

[3] P. Bourque and R. E. Fairley, "SWEBOK v3 - Guide to the Software Engineering Body of Knowledge," 2014.

[4] M. Felderer and A. Beer, "Using defect taxonomies for testing requirements," IEEE Software, vol. 32, no. 3, pp. 94–101, 2015.

[5] G. Upadhyaya and H. Rajan, "On Accelerating Source Code Analysis at Massive Scale," IEEE Transactions on Software Engineering, vol. 44, no. 7, pp. 669–688, 2018.

[6] Z. Li, X. Y. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "On the Multiple Sources and Privacy Preservation Issues for Heterogeneous Defect Prediction," IEEE Transactions on Software Engineering, vol. 45, no. 4, pp. 391–411, 2019.

[7] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," Proceedings - International Conference on Software Engineering, no. October 2014, pp. 200–210, 2012.

[8] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Uhayashi, "An empirical study of just-in-time defect prediction using cross-project models," Empirical Software Engineering, no. May 2014, pp. 172–181, 2014.

[9] Y. Cho, J. H. Kwon, and I. Y. Ko, "Cross-sub-project just-in-time defect prediction on multi-repo projects," CEUR Workshop Proceedings, vol. 2273, no. QuaSoQ, pp. 6–13, 2018.

[10] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," Proceedings - International Conference on Software Engineering, no. May 2010, pp. 31–41, 2010.

[11] M. Borg, O. Svensson, K. Berg, and D. Hansson, "SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project," in Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaTeSQE 2019, (New York, New York, USA), pp. 7–12, ACM Press, 2019.

[12] M. Kiehn, X. Pan, and F. Camei, "Empirical study in using version histories for change risk classification," IEEE International Working Conference on Mining Software Repositories, vol. 2019-May, pp. 58–62, 2019.

[13] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction," 25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings, vol. 2018-March, pp. 592–601, 2018.

[14] H. Liang, Y. Ya, L. Jiang, and Z. Xie, "Seml: A Semantic LSTM Model for Software Defect Prediction," IEEE Access, vol. 7, pp. 83812–83824, 2019.

[15] S. Kim, T. Zimmermann, E. J. Whitehead, and A. Zeller, "Predicting faults from cached history," Proceedings - International Conference on Software Engineering, pp. 489–498, 2007.

[16] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "BugCache for inspections: Hit or miss?" SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering, no. May 2014, pp. 322–331, 2011.

[17] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," no. September 2012, p. 1, 2012.

[18] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in Proceedings of the 38th International Conference on Software Engineering - ICSE ’16, vol. 14-22-May-, (New York, New York, USA), pp. 297–308, ACM Press, 2016.

[19] S. Matsumoto, Y. Kamei, A. Monden, K. I. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," ACM International Conference Proceeding Series, no. January, 2010.

[20] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-In-Time Defect Identification and Localization: A Two-Phase Framework," IEEE Transactions on Software Engineering, pp. 1–20, 2020.

[21] S. Wattanakriengkrai, P. Thongtanunam, C. Thanitthamthavorn, H. Hata, and K. Matsumoto, "Predicting Defective Lines Using a Model-Agnostic Technique," IEEE Transactions on Software Engineering, no. September, pp. 1–23, 2020.

[22] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. German, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," Empirical Software Engineering, vol. 25, no. 2, pp. 1294–1340, 2020.

[23] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope, "Measuring the impact of lexical and structural inconsistencies on developers’ cognitive load during bug localization," Empirical Software Engineering, vol. 25, no. 3, pp. 2140–2178, 2020.

[24] F. Qin, M. Yan, X. Xia, X. Wang, Y. Fan, A. E. Hassan, and D. Lo, "JITo: a tool for just-in-time defect identification and localization," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, no. September, (New York, NY, USA), pp. 156–159, ACM, nov 2020.

[25] C. Williams and J. Spacco, "SZZ revisited: verifying when changes induce fixes," in Proceedings of the 2008 workshop on Defects in large software systems - DEFECTS '08, no. January 2008, (New York, New York, USA), p. 32, ACM Press, 2008.

[26] A. Eghbali and M. Pradel, "No Strings Attached: An Empirical Study of String-related Software Bugs," Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, pp. 956–967, 2020.

[27] T. H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," 11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings, pp. 82–91, 2014.

[28] S. Davies, M. Roper, and M. Wood, "Comparing text-based and dependence-based approaches for determining the origins of bugs," Journal of Software: Evolution and Process, vol. 26, pp. 107–139, jan 2014.

[29] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm," Information and Software Technology, vol. 99, no. March, pp. 164–176, 2018.