# CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution of Faults in Large Industrial Projects

Mathieu Nayrolles
La Forge Research Lab, Ubisoft
Montréal, QC, Canada
mathieu.nayrolles@ubisoft.com

Abdelwahab Hamou-Lhadj
ECE Department, Concordia University
Montréal, QC, Canada
wahab.hamou-lhadj@concordia.ca

## I. Introduction

Automatic prevention and resolution of faults is an important research topic in the field of software maintenance and evolution. A particular line of research focuses on the problem of preventing the introduction of faults by detecting risky commits (commits that may potentially introduce faults in the system) before reaching the central code repository. We refer to this as just-in-time fault detection/prevention.

There exist techniques that aim to detect risky commits (e.g., [REF]), among which the most recent approach is the one proposed Rosen et al. [1]. The authors developed an approach and a supporting tool, Commit-guru, that relies on building models from historical commits using code and process metrics (e.g., code complexity, the experience of the developers, etc.) as the main features. These models are used to classify new commits as risky or not. Commit-guru has been shown to outperform previous techniques (e.g., [2], [3])

Commit-guru and similar tools suffer from a number of limitations. First, they tend to generate high false positive rates by classifying healthy commits as risky. The second limitation is that they do not provide sufficient insights to developers on how to fix the detected risky commits. They simply return measurements that are often difficult to interpret by developers. In addition, they have been mainly validated using open source systems. Their effectiveness when applied to industrial systems has yet to be shown.

In this paper, we propose an approach, called CLEVER (Combining Levels of Bug Prevention and Resolution techniques), that relies on a two-phase process for intercepting risky commits before they reach the central repository. The first phase consists of building a metric-based model to assess the likelihood that an incoming commit is risky or not. This is similar to existing approaches. The next phase relies on clone detection to compare code blocks extracted from risky commits (detected in the first phase) with those of known historical fault-introducing commits. This additional phase provides CLEVER with two apparent advantages over Commit-guru. First, as we will show in the evaluation section, CLEVER is able to reduce the number of false positives by relying on code matching instead of mere metrics. The second advantage is that, with CLEVER, it is possible to use commits that were used to fix faults introduced by previous commits to guide the developers on how to improve the risky commits at hand. This way, CLEVER goes one step further than Commit-guru (and similar techniques) by providing developers with a potential fix for their risky commits.

Another important aspect of CLEVER is its ability to detect risky commits not only by comparing them to commits of a single project but also to those belonging to other projects that share common dependencies. This is important in the context of an industrial project where software systems tend to have many dependencies that make them vulnerable to the same faults.

CLEVER was developed in collaboration with software developers from Ubisoft La Forge. Ubisoft is one of the world's largest video game development companies specializing in the design and implementation of high-budget video games. Ubisoft software systems are highly coupled containing millions of files and commits, developed and maintained by more than 8,000 developers scattered across 29 locations in six continents.

We tested CLEVER on 12 major Ubisoft systems. The results show that CLEVER can detect risky commits with 79% precision and 65% recall, which outperforms the performance of Commit-guru, which is 66% precision and 63% recall when applied to the same dataset. In addition, more than XX% of the proposed fixes were judged highly relevant by Ubisoft software developers, making CLEVER an effective and practical approach for the detection and resolution of risky commits.

The remaining parts of this paper are organised as follows. In Section II, we present related work. Sections III, IV and V are dedicated to describing the CLEVER approach, the case study setup, and the case study results. Then, Sections VI and VII present the threats to validity and a conclusion accompanied with future work.

## II. Related Work

The work most related to ours come from two main areas, work that aims to predict future defects in files, modules and changes and work that aims to propose or generate patches for buggy software.

### A. File, Module and Risky Change Prediction

The majority of previous file/module-level prediction work used code or process metrics. Approaches using code metrics only use information from the code itself and do not use any historical data. Chidamber and Kemerer published the well-known CK metrics suite [4] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [5]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [6].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [7], El Emam *et al.* [8], Subramanyam *et al.* [9] and Gyimothy *et al.* [10] for object-oriented designs. Service oriented designs have been far less studied than object oriented design as they are relatively new, but, Nayrolles *et al.* [11], [12], Demange *et al.* [13] and Palma *et al.* [14] used Moha *et al.* metric suites to detect software defects. All these approaches, proved software metrics to be useful at detecting software fault for object oriented and service oriented designs, respectively. More recently, Nagappan *et al.* [15], [16] and Zimmerman *et al.* [17], [18] further refined metrics-based detection by using statical analysis and call-graph analysis.

Other approaches use historical development data, often referred to as process metrics. Naggapan and Ball [19] studied the feasibility of using relative churn metrics to prediction buggy modules in the Windows Server 2003. Other work by Hassan *et al* and Ostrand *et al* used past changes and defects to predict buggy locations (e.g., [20], [21]). Hassan and Holt proposed an approach that highlights the top ten most susceptible locations to have a bug using heuristics based on file-level metrics [20]. They find that locations that have been recently modified and fixed locations are the most defect-prone. Similarly, Ostrand *et al.* [21] predict future crash location by combining the data from changed and past defect locations. They validate their approach on industrial systems at AT&T. They showed that data from prior changes and defects can effectively defect-prone locations for open-source and industrial systems. Kim *et al.* [22] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [20]. Rahman and Devanbu found that, in general, process-based metrics perform as good as or better than code-based metrics [23].

Other work focused on the prediction of risky changes. Kim et al. proposed the change classification problem, which predicts whether a change is buggy or clean [24]. Hassan [25] used the entropy of changes to predict risky changes. They find that the more complex a change is, the more likely it is to introduce a defect. Kamei *et al.* performed a large-scale empirical study on change classification [26]. They aforementioned studies find that size of a change and the history of the files being changed (i.e., how buggy they were in the past) are the best indicators of risky changes.

Our work shares a similar goal to works on the prediction of risky changes. However, CLEVERT takes a different approach in that it leverages dependencies of a project to determine risky changes.

### B. Automatic Patch Generation

Since CLEVER not only flags risky changes but also provides developers with fixes that have been applied in the past, automatic patch generation work is also related. Pan *et al.* [27] identified 27 bug fixing patterns that can be applied to fix software bugs in Java programs. They showed that between 45.7 - 63.6% of the bugs can be fixed with their patterns. Later, Kim *et al.* [28] generated patches from human-written patches and showed that their tool, PAR, successfully generated patches for 27 of 119 bugs. Tao *et al.* [29] also showed that automatically generated patches can assist developers in debugging tasks. Other work also focused on determining how to best generate acceptable and high quality patches, e.g. [30], [31], and determine what bugs are best fit for automatic patch generation [32].

Our work differs from the work on automated patch generation in that we do not generate patches, rather we use clone detection to determine the similarity of a change to a previous risky change and suggest to the developer the fixes of the prior risky changes.

## III. The CLEVER Approach

Figures 1, 2 and 3 show an overview of the CLEVER approach, which consists of two parallel processes.

In the first process (Figures 1 and 2), CLEVER manages events happening on project tracking systems to extract fault-introducing commits and commits used to provide the corresponding fixes. For simplicity reasons, in the rest of this paper, we refer to commits that are used to fix defects as *fix-commits*. We use the term *defect-commit* to mean a commit that introduces a fault.

The project tracking component of CLEVER listens to bug (or issue) closing events of Ubisoft projects. Currently,CLEVER is tested with 12 large projects within Ubisoft. These projects share many dependencies. We clustered them based on their dependencies with the aim to improve the accuracy of CLEVER. This clustering step is important in order to identify faults that may exist due to dependencies, while enhancing the quality of the proposed fixes. Applying CLEVER to projects that are not related to each other may turn to be ineffective.

In the second process (Figure 3), CLEVER intercepts incoming commits before leaving the developers' workstation using the concept of pre-commit hooks. A pre-commit hook is a script that is executed at commit-time and it is supported by most major code versioning systems such as Github. There are two types of hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, whereas server-side hooks run on network operations such as receiving pushed commits. These hooks can be used

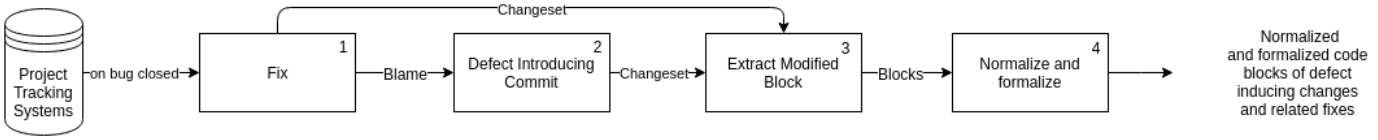


Fig. 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

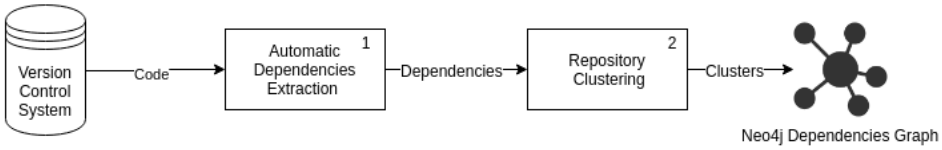


Fig. 2: Clustering by dependency

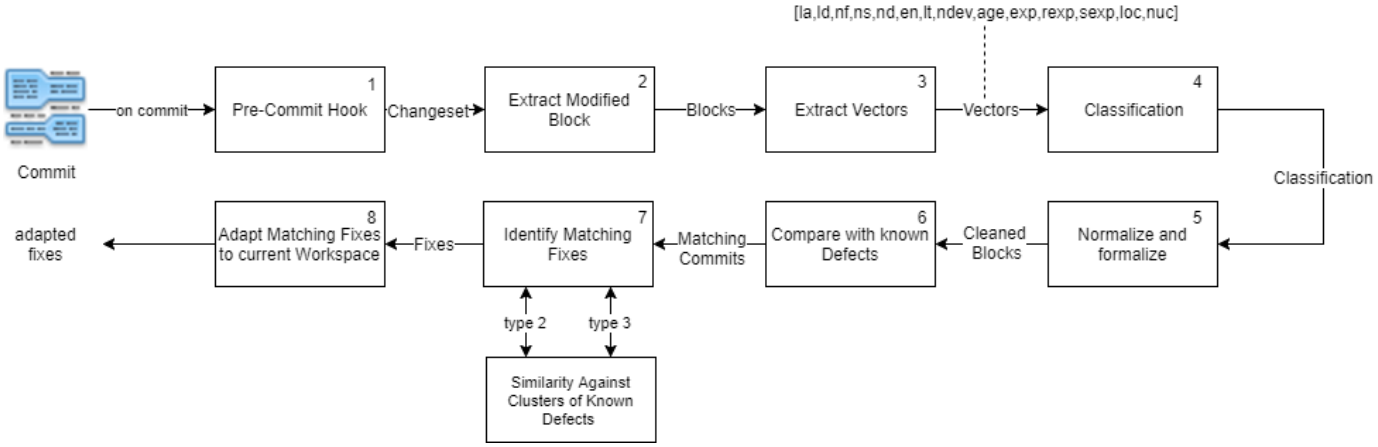


Fig. 3: Classifying incoming commits and proposing fixes

for different purposes such as checking compliance with coding rules, or the automatic execution of unit tests. A pre-commit hook runs before a developer specifies a commit message.

Ubisoft's developers use pre-commit hooks for all sorts of reasons such as identifying the tasks that are addressed by the commit at hand, specifying the reviewers who review the commit, and so on. Implementing this part of CLEVER as a pre-commit hook is an important step towards the integration of CLEVER with the workflow of developers at Ubisoft. The developers do not have to download, install, and understand additional tools in order to use CLEVER.

Once the commit is intercepted, we compute code and process metrics associated with this commit. The selected metrics are discussed further in Section III-B. The result is a feature vector (Step 4) that is used for classifying the commit as *risky* or *non-risky*.

If the commit is classified as *non-risky*, then the process stops, and the commit can be transferred from the developer's workstation to the central repository. *Risky* commits, on the other hand, are further analysed in order to reduce the number of false positives (healthy commits that are detected as risky). We achieve this by first extracting the code block that is modified by the developer in order to apply the commit and then comparing it to code blocks of known fault-introducing commits.

### A. *Clustering Projects*

We cluster projects according to their dependencies. The rationale is that projects that share dependencies are most likely to contain defects caused by misuse of these dependencies. In this step, the project dependencies are analysed and saved into a single NoSQL graph database as shown in Figure 2. A node corresponds to a project that is connected to other projects on which it depends. Dependencies can be *external* or *internal* depending on whether the products are created in-house or supplied by a third-party. For confidentiality reasons, we cannot reveal the name of the projects involved in the project dependency graph. We show the 12 projects in yellow color with their dependencies in blue color 4. The resulting partitioning is shown in 5.

Internal dependencies are managed within the framework of a single repository, which makes their automatic extraction possible. The dependencies could also be automatically retrieved if the projects use a dependency manager such as Maven.

Once the project dependency graph is extracted, we use a clustering algorithm to partition the graph. To this end, we choose the Girvan–Newman algorithm [33], [34], used to detect communities by progressively removing edges from the original network. Instead of trying to construct a measure that identifies the edges that are the most central to communities,

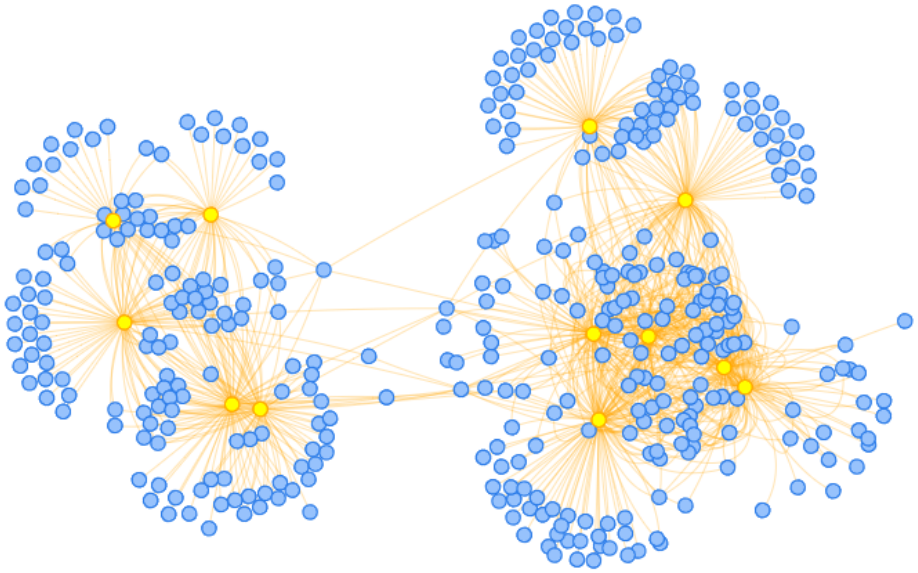

Fig. 4: Dependency Graph

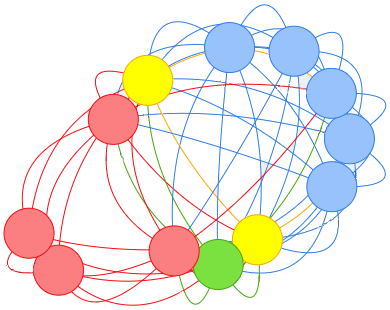

Fig. 5: Clusters

the Girvan–Newman algorithm focuses on edges that are most likely "between" communities. This algorithm is very effective at discovering community structure in both computer-generated and real-world network data [34]. Other clustering algorithms can also be used.

### *B. Building a Database of Code Blocks of Defect-Commits and Fix-Commits*

To build our database of code blocks that are related to defect-commits and fix-commits, we first need to identify the respective commits. Then, we extract the relevant blocks of code from the commits.

**Extracting Commits:** CLEVER listens to issue closing events happening on the project tracking system used at Ubisoft. Every time an issue is closed, CLEVER retrieves the commit that was used to fix the issue (the fix-commit) as well as the one that introduced the defect (the defect-commit). To link fix-commits and their related issues we implemented the SZZ algorithm presented by Kim et al. [35].

**Extracting Code Blocks:** Algorithm 1 presents an overview of how to extract blocks. This algorithm receives as arguments, the changesets and the blocks that have been previously extracted. Then, Lines 1 to 5 show the $for$ loop that iterates over the changesets. For each changeset (Line 2), we extract the blocks by calling the $extract\_blocks(Changeset\ cs)$ function. In this function, we expand our changeset to the left and to the right in order to have a complete block.

```
   Data: Changeset[] changesets;
         Block[] prior_blocks;
   Result: Up to date blocks of the systems
1  for i ← 0 to size_of changesets do
2      Block[] blocks ← extract_blocks(changesets);
3      for j ← 0 to size_of blocks do
4          write blocks[j];
5      end
6  end
7  Function extract_blocks(Changeset cs)
8      if cs is unbalanced right then
9          cs ← expand_left(cs);
10     else if cs is unbalanced left then
11         cs ← expand_right(cs);
12     end
14     return txl_extract_blocks(cs);
```

**Algorithm 1:** Overview of the Extract Blocks Operation

As depicted by the diff below (not from Ubisoft), changesets contain only the modified chunk of code and not necessarily complete blocks.

```
@@ -315,36 +315,6 @@
int initprocesstree_sysdep
(ProcessTree_T **reference) {
    mach_port_deallocate(mytask,
```

```
      task);
}
}
- if (task_for_pid(mytask, pt[i].pid,
-  &task) == KERN_SUCCESS) {
-   mach_msg_type_number_t   count;
-   task_basic_info_data_t   taskinfo;
```

Therefore, we need to expand the changeset to the left (or right) to have syntactically correct blocks. We do so by checking the block's beginning and ending with parentheses algorithms [36].

### C. Building a Metric-Based Model

We adapted Commit-guru [1] for building the metric-based model. Commit-guru uses a list of keywords proposed by Hindle *et al.* [37] to classify commit in terms of *maintenance*, *feature* or *fix*. Then, it uses the SZZ algorithm to find the defect-commit linked to the fix-commit. For each defect-commit, Commit-guru computes the following code metrics: *la* (lines added), *ld* (lines deleted), *nf* (number of modified files), *ns* (number of modified subsystems), *nd* (number of modified directories), *en* (distriubtion of modified code across each file), *lt* (lines of code in each file (sum) before the commit), *ndev* (the number of developers that modifed the files in a commit), *age* (the average time interval between the last and current change), *exp* (number of changes previously made by the author ), *rexp* (experience weighted by age of files (1 / (n + 1))), *sexp* (previous changes made by the author in the same subsystem), *loc* (total number of modified LOC across all files), *nuc* (number of unique changes to the files). Then, a statistical model is built using the metric values of the defect-commits. Using linear regression, Commit-guru is able to predict whether incoming commits are *risky* or not.

We had to modify Commit-guru to fit the context of this study. First, we used information found in Ubisoft's internal project tracking system used at Ubisoft to classify the purpose of a commit (i.e., *maintenance*, *feature* or *fix*). In other words, CLEVER only classifies a commit as a defect-commit if it is the root cause of a fix linked to a crash in the internal project tracking system. Using internal pre-commit hooks, Ubisoft developers must link every commit to a given task #ID. If the task #ID entered by the developer matches a bug or crash report within the project tracking system, then we perform the SCM blame/annotate function on all the modified lines of code for their corresponding files on the fix-commit's parents. This returns the commits that previously modified these lines of code, and are flagged as defect-commits. Another modification consists of the actual classification algorithm. We did not use linear regression but instead the random forest algorithm. The random forest algorithm turned out to be more effective as described in Section V. Finally, we had to rewrite Commit-guru in GoLang for performance and internal reasons.

### D. Comparing Code Blocks

Each time a developer makes a commit, CLEVER intercepts it using a pre-commit hook and classifies it as *risky* or not. If the commit is classified as *risky* by the metric-based classifier, then, we extract the corresponding code block (in a similar way as in the previous phase), and compars it to the code blocks of historical defect-commits. If there is a match, then the new commit is deemed to be risky. A threshold $S$ is used to assess the extent beyond which two commits are considered similar.

To compare the extracted blocks to the ones in the database, we resort to clone detection techniques, more specifically, text-based clone detection techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based comparisons) would require a complete program to work, i.e., a program that compiles.

In the relatively wide-range of tools and techniques that exist to detect clones by considering code as text [38]–[43], we chose the NICAD clone detector because it is freely available and has shown to perform well [44]. We improved NICAD to process blocks that comes from commit-diffs. This is because the current version of NICADcan only process syntactically correct code and commit-diffs are, by definition, snippets that represent modified regions of a given set of files.

By reusing NICAD, CLEVER can detect Types 3 software clones [45]. Type 3 clones can contain added or deleted code statements, which make them suitable for comparing commit code blocks. In addition, NICAD uses a pretty-printing strategy from where statements are broken down into several lines [46]. This functionality allowed us to detect Segments 1 and 2 as a clone pair, as shown by Table I, because only the initialization of $i$ changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc [42].

The extracted, pretty-printed, normalized filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [47]. Then, a percentage of unique statements can be computed and, given the threshold $\alpha$, the blocks are marked as clones.

### E. Classifying Incoming Commits

As discusses in Section XXXX, a new commit goes through the metric-based model first (Steps 1 to 4). If classified as *non-risky*, we simply let it through, and we stop the process. If the commit is classified as *risky*, however, we continue the process with Steps 5 to 9 our approach.

One may wonder why we needed to have a metric-based model in the first place. we could have resorted to clone detection as the main mechanism. The main reason for having the metric-based model is efficiency. If each commit had to be analysed against all known signatures using code clone similarity, then, it would have made CLEVER time consuming.

### F. Proposing Fixes

An important aspect in the design of CLEVER is the ability to provide guidance to developers on how to improve risky commits. We achieve this by extracting from the database the fix-commit corresponding to the matching defect-commit and present it to the developer. We believe that this makes CLEVER a practical approach. Developers can understand why a given

TABLE I: Pretty-Printing Example

| Segment 1 | Segment 2 | Segment 3 | S1 & S2 | S1 & S3 | S2 & S3 |
|---|---|---|---|---|---|
| for (<br>i = 0;<br>i >10;<br>i++) | for (<br>i = 1;<br>i >10;<br>i++) | for (<br>j = 2;<br>j >100;<br>j++) | 1<br>0<br>1<br>1 | 1<br>0<br>0<br>0 | 1<br>0<br>0<br>0 |
| Total Matches | | | 3 | 1 | 1 |
| Total Mismatches | | | 1 | 3 | 3 |

modification has been reported as risky by looking at code instead of simple metrics as in the case of the studies reported in [1], [2]).

Finally, using the fixes of past defects, we can provide a solution, in the form of a contextualised diff, to the developers. A contextualised diff is a diff that is modified to match the current workspace of the developer regarding variable types and names. In Step 8 of Figure 3, we adapt the matching fixes to the actual context of the developer by modifying indentation depth and variable name in an effort to reduce context switching. We believe that this would make it easier for developers to understand the proposed fixes and see if they apply in their situation.

[Wahab: you can remove the bubble sort example; I think you explained well the concept. This will also save us 1 page].

## IV. Case Study Setup

In this section, we present the setup of our case study in terms of repository selection, dependency analysis, comparison process and evaluation measures.

### A. Project Repository Selection

In collaboration with Ubisoft developers, we selected 12 major software projects (i.e., systems) developed at Ubisoft to evaluate the effectiveness of CLEVER. These systems continue to be actively maintained by thousands of developers. Ubisoft projects are organized by game engines. A game engine can be used in the development of many high-budget games. The projects selected for this case study are related to the same game engine. For confidentiality and security reasons, neither the names nor the characteristics of these projects are provided. We can however disclose that the size of these systems altogether consists of millions of lines of code.

### B. Project Dependency Analysis

Figure 4 shows the project dependency graph. As shown in Figure 4, these projects are highly interconnected. A review of each cluster shows that this partitioning divides projects in terms of their high-level functionalities. For example, one cluster is related to a particular given family of video games, whereas the other cluster refers to another family. We showed this partitioning to 11 experienced software developers and ask them to validate it. They all agreed that the results of this automatic clustering is accurate and reflects well the various projects group of the company.

### C. Building a Database of Defect-Commits and Fix-Commits

To build the database that we can use to assess the performance of CLEVER, we use the same process as discussed in Section III-B. We retrieve the full history of each project and label commits as defect-commits if they appear to be linked to a closed issue using the SZZ algorithm [35]. This baseline is used to compute the precision and recall of CLEVER. Each time CLEVER classifies a commit as *risky*; we can check if the *risky* commit is in the database of defect-introducing commits. The same evaluation process is used by related studies [3], [8], [26], [48], [49].

### D. Process of Comparing New Commits

Because our approach relies on commit pre-hooks to detect risky commits, we had to find a way to *replay* past commits. To do so, we *cloned* our test subjects, and then created a new branch called *CLEVER*. When created, this branch is reinitialized at the initial state of the project (the first commit), and each commit can be replayed as they have originally been. For each commit, we store the time taken for *CLEVER* to run, the number of detected clone pairs, and the commits that match the current commit. As an example, suppose that we have three commits from two projects. At time $t_1$, commit $c_1$ in project $p_1$ introduces a defect. The defect is experienced by a user that reports it via an issue $i_1$ at $t_2$. A developer fixes the defect introduced by $c_1$ in commit $c_2$ and closes $i_1$ at $t_3$. From $t_3$ we known that $c_1$ introduced a defect using the process described in Section IV-C. If at $t_4$, $c_3$ is pushed to $p_2$ and $c_3$ matches $c_1$ after preprocessing, pretty-printing and formatting, then $c_3$ is classified as *risky* by CLEVER and $c_2$ is proposed to the developer as a potential solution for the defect introduced in $c_3$.

### E. Evaluation Measures

Similar to prior work (e.g., [24], [26]), we used precision, recall, and $F_1$-measure to evaluate our approach. They are computed using TP (true positives), FP (false positives), FN (false negatives), which are defined as follows:

- TP is the number of defect-commits that were properly classified by CLEVER
- FP is the number of healthy commits that were classified by CLEVER as risky
- FN is the number of defect introducing-commits that were not detected by CLEVER
- Precision: TP / (TP + FP)
- Recall: TP / (TP + FN)
- $F_1$-measure: 2.(precision.recall)/(precision+recall)

It is worth mentioning that, in the case of defect prevention, false positives can be hard to identify as the defects could be in the code but not yet reported through a bug report (or issue). To address this, we did not include the last six months of history. Following similar studies [1], [50]–[52], if a defect is not reported within six months then it is not considered.

## V. Case Study Results

In this section, we show the effectiveness of CLEVER in detecting risky commits using a combination of metric-based models and clone detection. The main research question addressed by this case study is: *Can we detect risky commits by combining metrics and code comparison within and across related Ubisoft projects, and if so, what would be the accuracy?*

The experiments took nearly two months using a cluster of six 12 3.6 Ghz cores with 32GB of RAM. The most time consuming part of the experiment consists of building the baseline as each commit must be analysed with the SZZ algorithm. Once the baseline was established, the model built, it took, on average, 3.75 seconds to analyse an incoming commit on our cluster.

In the following subsections, we provide insights on the performance of CLEVER by comparing it to Commit-guru [1] alone, i.e., an approach that relies only on metric-based models. We chose Commit-guru because it has been shown to outperform other techniques (e.g., [2], [3]). Commit-guru is also open source and easy to use.

[Wahab: I see that the case study does not answer this question. It deals with cold start. The problem is that the cold start evaluation is really unclear. See my comment below. I think it is better to remove the cold start and replace it with a small study that addresses the text below. Another solution is to remove both points all together from the paper] In addition, we compare CLEVER to itself but without applying the clustering step. This is important in order to validate the usefulness of the clustering step.

### A. Performance of CLEVER

when applied to 12 Ubisoft projects, CLEVER detects risky commits with an average precision, recall, and F1-measure of 79.10%, a 65.61%, and 71.72% respectively. For clone detection, we used a threshold of 30% . This is because Roy et al.[REF] showed through empirical studies that using NICAD with a threshold of around 30%, the default setting, provides good results for the detection of Type 3 clones. When applied to the same projects, Commit-guru achieves an average precision, recall, and F1-measure of 66.71%, 63.01% and 64.80%, respectively.

We can see that the second phase of CLEVER (clone detection) considerably reduces the number of false positives (precision of 79.10% for CLEVER compared to 66.71% for Commit-guru), while achieving similar recall (65.61% for CELEVER compared to 63.01% for Commit-guru).

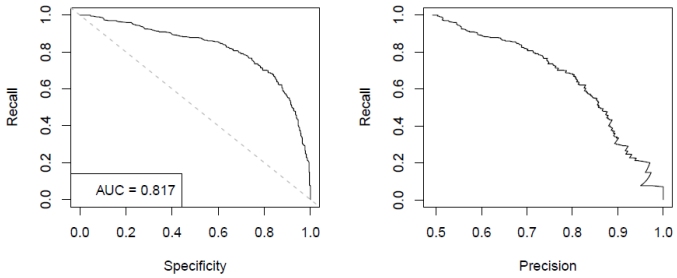


Fig. 6: Performances of Misfire while cold-starting the last project in the blue cluster

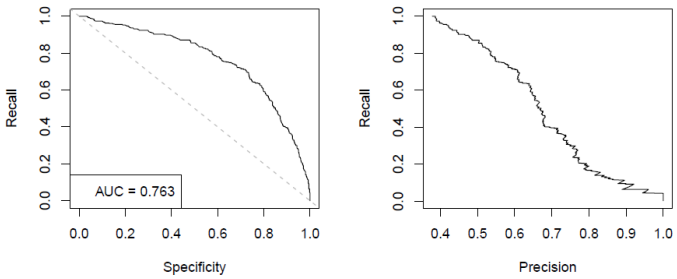


Fig. 7: Performances of Misfire while cold-starting the last project in the yellow cluster

### B. Cluster Classifier Performance

The clusters computed with the dependencies of each project help to solve an important problem in defect prediction, known as cold start [53]. [Wahab: you need to explain this in one or two sentences]

There exist approaches have been proposed to solve this problems, however, they all require to classify *similar* projects by hand and then, manipulate the feature space in order to adapt the model learnt from one project to another one [54].

[WahabL you need to develop this section. The text is not clear. The graphs are not either. I think defect learning transfer is another topic you can tackle in another paper. One possibility to fix this is to talk about the result of CLEVER without the clustering step]

With our approach, the *similarity* between projects is computed automatically and the results show that we do not actually need to manipulate the feature space. Figures 6, 7 and 8 show the performance of CLEVER classification in terms of ROC-curve for the first thousand commits of the last project (chronologically) for the blue, yellow and red clusters presented in Figure 5. The left sides or the graph are low cutoff (aggressive) while the right sides are high cutoff (conservative). The area under the ROC curves are 0.817, 0.763 and 0.806 for the blue, yellow and red clusters, respectively.

[Wahab: please use P1, P2.... P6 and F1, F2, ... F12. Also, to use the terms Accepted, Rejected, Unsure instead of checks, x, and -)]

These results show that not only the clusters are effective in identifying similar projects for defect learning transfer but also provide excellent performance while starting a new project. As new projects mature, their defects would be integrated into

TABLE II: Workshop results

| | BF1 | BF2 | BF3 | BF4 | BF5 | BF6 | BF7 | BF8 | BF9 | BF10 | BF11 | BF12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R1 | ✓ | x | ✓ | ✓ | - | ✓ | - | x | x | ✓ | ✓ | - |
| R2 | ✓ | x | ✓ | - | - | ✓ | - | x | x | ✓ | ✓ | - |
| R3 | ✓ | x | ✓ | - | - | ✓ | - | x | x | ✓ | ✓ | - |
| R4 | ✓ | x | ✓ | - | - | ✓ | - | ✓ | x | ✓ | ✓ | - |
| R5 | ✓ | x | ✓ | ✓ | - | ✓ | - | x | x | ✓ | ✓ | - |
| R6 | ✓ | x | ✓ | - | - | ✓ | - | ✓ | x | ✓ | ✓ | - |

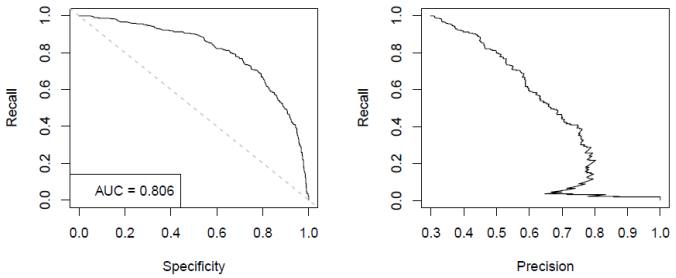


Fig. 8: Performances of Misfire while cold-starting the last project in the red cluster

the model in order to improve it. To confirm this, we ran an experiment with the blue cluster where we first apply the model learnt from other members of the cluster for the first thousand commits. The performance of this model is 75.1% precision, 57.6% recall for the first thousand commits.

After the first thousand commits, we take the commits of the day (for each day until the end of the project) and rebuild the model by adding the commits of the day to the ones already known from the clusters. Overall, combining the commits from the project at hand in a nightly fashion with the commits known from the cluster allowed to correctly classify an additional 2.05% of commits in the last 30% of the project life compared to only using the commits from the project. This shows that, in addition to providing a viable alternative to a cold-start, our clusters also allow us to enhance the performance of the model at all time and no only at the start.

### C. Analysis of the Quality of the Fixes Proposed by CLEVER

In order to validate the quality of the fixes proposed by CLEVER, we conducted an internal workshop where we invited a number of Ubisoft development team. The workshop was attended by six participants: two software architects, two developers, one technical lead, and one IT project manager. The participants have many years of experience at Ubisoft.

In this workshop, the participants were asked to assess the quality of proposed fixed. We presented them with the original buggy commits, the original fixes for these commits, and the fixes that are automatically extracted by CLEVER.

The participants were asked to review 12 randomly selected fixes that were proposed by CLEVER. These fixes are related to one system in which the participants have excellent knowledge. We presented them with the original buggy commits, the original fixes for these commits, and the fixes that were automatically extracted by CLEVER. We asked them the following questions:

*"Is the proposed fix applicable in the given situation?"* [Wahab: please other questions]

The review session took around 70 minutes. This does not include the time it took to explain the objective of the session, the setup, the collection of their feedback, etc.

We asked the participants to rank each fix that is proposed by CLEVER using this scheme: [Wahab: please use bullet points]

Fix Accepted: The participant found the fix proposed by CLEVER applicable to the risky commit Unsure: In this situation, the participant is unsure about the relevance of the fix. There might be a need for more information to arrive to a verdict. Fix Rejected: The participant found the fix is not applicable to the risky commit

Table II shows answer of the participants. The columns refer to the fixes proposed by CLEVER, whereas the rows refer to the participants that we denote using P1, P2, ..., P6. As we can from the table, 41.6% of the proposed fixes (F1, F3, F6, F10 and F12) have been accepted by all participants, while 25% have been accepted by at least one member (F4, F8, F11). We analysed the fixes that were rejected by some or all participants to understand the reasons.

$F2$ was rejected by our participants because the region of the commit that triggered a match is a generated code. Although this generated code was pushed into the repositories as part of bug fixing commit, the root cause of the bug lies in the code generator itself. Our proposed fix suggests to update the generated code. Because the proposed fix did not apply directly to the the question we ask our reviewers was *"Is the proposed fix applicable in the given situation?"*. In this occurrence, the proposed fix was not applicable.

$F4$ was accepted by two reviewers and marked as unsure $(-)$ by the other participant. We believe that this was due the lack of context surrounding the proposed fix. The participants were unable to determine if the fix was applicable or not without knowing what the original intent of the buggy commit was. In our review session, we only provided the reviewers with the regions of the commits that matched existing commits and not the full commit. Full commits can be quite lengthy as they can contain asset descriptions and generated code, in addition to the actual code. In this occurrence, the full context of the commit might have helped our reviewers to decide if the $F4$ was applicable or not. $F5$ and $F7$ were classified as unsure by all our participants for the same reasons.

$F8$ was rejected by four of participants and accepted by two. The participant argued that the proposed fix was more a refactoring opportunity than an actual fix.

$F12$ was marked as unsure by all the reviewers because the

code had to do with a subsystem that is maintained by another team and the participants felt that it is out of scope of this session.

After the session, we asked the participants two additional questions: *Will you use CLEVER in the future?* and *What aspects of CLEVER need to be improved?*.

The participants answered the first question positively. They all agreed that CLEVER could be a good tool for intercepting risky commits, and hence improving the quality assurance process. For the second question, the participants expressed concerns about the context surrounding the buggy commits and the fixes. While displaying the entire commits is not a solution, according to the participants, some context might be inferred from the commit messages and the issues associated with the fixes in the bug tracking systems. The second limitation of CLEVER is its inability to deal with generated code. At this point, CLEVER points towards the code generator rather than the generated code. This aspect of CLEVER needs to be improved.

## VI. Discussion

In this section, we propose a discussion on limitations and threats to validity.

### A. Limitations

We identified two main limitations of our approach, CLEVER, which require further studies.

CLEVER is designed to work on multiple related systems. Applying CLEVER on a single system will most likely be less effective. The the two-phase classification process of CLEVER would be hindered by the fact that it is unlikely to have a large number of similar bugs within the same system. For single systems, we recommend the use of metric-based models. A metric-based solution, however, may turn to be ineffective when applied across systems because of the difficulty associated with identifying common thresholds that are applicable to a wide range of systems.

The second limitation we identified has to do with the fact that CLEVER is designed to work with Ubisoft systems. Ubisoft uses C#, C, C++, Java and other internally developed languages. It is however common to have other languages used in an environment with many inter-related systems. We intend to extend CLEVER to process commits from other languages as well.

### B. Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. Because of the industrial nature of this study, we had to work with the systems developed by the company.

The programs we used in this study are all based on the C#, C, C++ and Java programming languages. This can limit the generalization of the results to projects written in other languages, especially that the main component of CLEVER is based on code clone matching.

Finally, part of the analysis of the CLEVER proposed fixes that we did was based on manual comparisons of the CLEVER fixes with those proposed by developers with a focus group composed of experienced engineers and software architects. Although, we exercised great care in analysing all the fixes, we may have misunderstood some aspects of the commits.

In conclusion, internal and external validity have both been minimized by choosing a set of 12 different systems, using input data that can be found in any programming languages and version systems (commits and changesets).

## VII. Conclusion

In this paper, we presented CLEVER (Combining Levels of Bug Prevention and Resolution Techniques), an approach that detects risky commits (i.e., a commit that is likely to introduce a bug) with an average of 79.10% precision and a 65.61% recall. CLEVER combines code metrics, clone detection techniques and project dependency analysis to detect risky commits within and across projects. CLEVER operates at commit-time, i.e., before the commits reach the central code repository. Also, because it relies on code comparison, CLEVER does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes CLEVER a practical approach for preventing bugs and proposing corrective measures that integrate well with the developer's workflow through the commit mechanism.

As a future work, we want to build a feedback loop between the users and the clusters of known buggy commits and their fixes. If a fix is never used by the end-users, then we could remove it from the clusters and improve our accuracy. We also intend to improve CLEVER to deal with generated code. We also want to improve the fixes proposed by CLEVER to add contextual information to help developers better assess the applicability of the fixes.

## VIII. Reproduction Package

For security and confidentiality reasons we cannot provide a reproduction package that will inevitably involve Ubisoft's copyrighted source code. However, the CLEVER source code is in the process of being open-sourced and will be soon available at https://github.com/ubisoftinc.

## IX. Acknowledgments

We are thankful to Olivier Pomarez, Yves Jacquier, Nicolas Fleury, Alain Bedel, Mark Besner, David Punset, Paul Vlasie, Cyrille Gauclin, Luc Bouchard, Chadi Lebbos and Florent Jousset, Anthony Brien, Thierry Jouin and Jean-Pierre Nolin from Ubisoft for their participations in validating CLEVERT hypothesis, efficiency and the fixes proposed by our approach.

## REFERENCES


[1] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the joint meeting on foundations of software engineering*, 2015, pp. 966–969.

[2] Y. Kamei, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[3] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," *Empirical Software Engineering*, vol. 16, no. 1, pp. 141–175, Dec. 2010.

[4] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[5] N. Moha, F. Palma, M. Nayrolles, and B. J. Conseil, "Specification and Detection of SOA Antipatterns," in *International conference on service oriented computing*, 2012, pp. 1–16.

[6] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[7] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[8] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, Feb. 2001.

[9] R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, Apr. 2003.

[10] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[11] M. Nayrolles, A. Maiga, A. Hamou-lhadj, and A. Larsson, "A Taxonomy of Bugs : An Empircial Study," pp. 1–10.

[12] M. Nayrolles, N. Moha, and P. Valtchev, "Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces," in *Working conference on reverse engineering*, 2013, pp. 321–330.

[13] A. Demange, N. Moha, and G. Tremblay, "Detection of SOA Patterns," in *Proceedings of the international conference on service-oriented computing*, 2013, pp. 114–130.

[14] F. Palma, "Detection of SOA Antipatterns," PhD thesis, Ecole Polytechnique de Montreal, 2013.

[15] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the international conference on software engineering*, 2005, pp. 580–586.

[16] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceeding of the international conference on software engineering*, 2006, pp. 452–461.

[17] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *Proceedings of the international workshop on predictor models in software engineering*, 2007, p. 9.

[18] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 13th international conference on software engineering - iCSE '08*, 2008, p. 531.

[19] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the international conference on software engineering*, 2005, pp. 284–292.

[20] A. Hassan and R. Holt, "The top ten list: dynamic fault prediction," in *Proceedings of the international conference on software maintenance*, 2005, pp. 263–272.

[21] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[22] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *Proceedings of the international conference on software engineering*, 2007, pp. 489–498.

[23] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the international conference on software engineering*, 2013, pp. 432–441.

[24] S. Sunghun Kim, E. Whitehead, and Y. Yi Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[25] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the international conference on software engineering*, 2009, pp. 78–88.

[26] Y. Kamei, "Studying re-opened bugs in open source software," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[27] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Aug. 2008.

[28] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the international conference on software engineering*, 2013, pp. 802–811.

[29] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the international symposium on foundations of software engineering*, 2014, pp. 64–74.

[30] V. Dallmeier, A. Zeller, and B. Meyer, "Generating Fixes from Object Behavior Anomalies," in *Proceedings of the international conference on automated software engineering*, 2009, pp. 550–554.

[31] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the international conference on software engineering*, 2012, pp. 3–13.

[32] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?" in *Proceedings of the international symposium on software reliability engineering*, 2015, pp. 427–437.

[33] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.

[34] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, Feb. 2004.

[35] S. Kim, T. Zimmermann, K. Pan, and E. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *Proceedings*

*of the international conference on automated software engineering*, 2006, pp. 81–90.

[36] B. Bultena and F. Ruskey, "An Eades-McKay algorithm for well-formed parentheses strings," *Information Processing Letters*, vol. 68, no. 5, pp. 255–259, 1998.

[37] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits." in *Proceedings of the international workshop on mining software repositories*, 2008, pp. 99–108.

[38] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the conference of the centre for advanced studies on collaborative research*, 1993, pp. 171–183.

[39] J. H. Johnson, "Visualizing textual redundancy in legacy source," in *Proceedings of the conference of the centre for advanced studies on collaborative research*, 1994, p. 32.

[40] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *Proceedings international conference on automated software engineering*, 2001, pp. 107–114.

[41] U. Manber, "Finding similar files in a large file system," in *Proceedings of the usenix winter*, 1994, pp. 1–10.

[42] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proceedings of the international conference on software maintenance*, 1999, pp. 109–118.

[43] R. Wettel and R. Marinescu, "Archeology of code duplication: recovering duplication chains from small duplication fragments," in *Proceedings of the seventh international symposium on symbolic and numeric algorithms for scientific computing*, 2005, pp. 63–71.

[44] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *Proceedings of the international conference on program comprehension*, 2011, pp. 219–220.

[45] C. Kapser and M. W. Godfrey, "Toward a Taxonomy of Clones in Source Code: A Case Study," in *International workshop on evolution of large scale industrial software architectures*, 2003, pp. 67–78.

[46] C. K. Roy, "Detection and Analysis of Near-Miss Software Clones," PhD thesis, Queen's University, 2009.

[47] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, May 1977.

[48] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the european conference on foundations of software engineering*, 2011, pp. 311–231.

[49] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceeding of the iIternational conference on mining software repositories*, 2011, pp. 207–210.

[50] T.-h. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An Empirical Study of Dormant Bugs Categories and Subject Descriptors," in *Proceedings of the international conference on mining software repository*, 2014, pp. 82–91.

[51] S. Shivaji, S. Member, S. Member, R. Akella, and S. Kim, "Reducing Features to Improve Code Change-Based Bug Prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2013.

[52] Y. Kamei, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[53] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, "Methods and metrics for cold-start recommendations," in *Proceedings of the 25th annual international aCM sIGIR conference on research and development in information retrieval - sIGIR '02*, 2002, p. 253.

[54] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the international conference on software engineering*, 2013, pp. 382–391.