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

Double Blind Review

ABSTRACT

Automatic prevention and resolution of faults is an important research topic in the field of software maintenance and evolution. Existing approaches leverage code and process metrics to build metric-based models that can effectively prevent defect insertion in a software project. Metrics, however, may vary from one project to another, hindering the reuse of these models. Moreover, they tend to generate high false positive rates by classifying healthy commits as risky. Finally, they do not provide sufficient insights to developers on how to fix the detected risky commits. In this paper, we propose an approach, called CLEVER (Combining Levels of Bug Prevention and Resolution techniques), which relies on a two-phases process for intercepting risky commits before they reach the central repository. CLEVER was developed in collaboration with Ubisoft developers. When applied to 12 Ubisoft systems, the results show that CLEVER can detect risky commits with 79% precision and 65% recall, which outperforms the performance of Commit-guru, a recent approach that was proposed in the literature. In addition, CLEVER is able to recommend qualitative fixes to developers on how to fix risky commits in 66.7% of the cases.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Maintaining software*; • **Information systems** → Expert systems;

KEYWORDS

Defect Predictions, Fault Fixing, Software Maintenance, Software Evolution

ACM Reference Format:
Double Blind Review. 018. CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution in Large Industrial Projects. In *Proceedings of ACM ICSE Conference (ICSE'18).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automatic prevention and resolution of faults is an important research topic in the field of software maintenance and evolution. Effective approaches can help reduce significantly the cost of maintenance of software systems, while improving their quality. 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 they reach 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., [2, 5, 42]), among which the most recent approach is the one proposed by Rosen et al. [38]. 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 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., [18, 25]).

However, 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 recommendations 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-phases 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 suspicious 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 suggest recommendations to 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

ICSE'18, May 2018, Gothemburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

setting 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 (66% precision and 63% recall) when applied to the same dataset. In addition, 66.7% of the proposed fixes were accepted by a least one Ubisoft software developer, 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 2, we present related work. Sections 3, 4 and 5 are dedicated to describing the CLEVER approach, the case study setup, and the case study results. Then, Sections 6 and 7 present the threats to validity and a conclusion accompanied with future work.

2 RELATED WORK

Our approach, CLEVER, is related to two research areas: defect prediction and patch generation.

2.1 File, Module and Risky Change Prediction

Existing studies for predicting risky changes within a repository rely mainly on code and process metrics. As discussed in the introduction section, Rosen et al. [38] developed Commit-guru a tool 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. There exist other studies that leverage several code metric suites such as the CK metrics suite [5] or the Briand's coupling metrics [2]. These metrics have been used, with success, to predict defects as shown by Subramanyam *et al.* [42] and Gyimothy *et al.* [11].

Further improvements to these metrics have been proposed by Nagappan *et al.* [31, 33] and Zimmerman *et al.* [46, 47] who used call graphs as the main artifact for computing code metrics with a static analyzer.

Nagappan *et al.* et proposed a technique that uses data mined from source code repository such as churns to assess the quality of a change [32]. Hassan *et al* and Ostrand *et al* used past changes and defects to predict buggy locations [12], [35]. Their methods rely on various heuristics to identify the locations that are most likely to introduce a defect. Kim *et al* [24] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [12]. Rahman and Devanbu found that, in general, process-based metrics perform as good as code-based metrics [37].

Other studies that aim to predict risky changes use the entropy of a given change [13] and the size of the change combined with files being changed [19].

These techniques operate at different levels of the systems and may require the presence of the entire source code. In addition, the reliance of metrics may result in high false positives rates. We need a way to to validate whether a suspicious change is indeed risky. In this paper, we address this issue using a two-phase process that combines the use of metrics to detect suspicious risky changes, and code matching to increase the detect accuracy. As we will show in the evaluation section, CLEVER reduces the number of false positives while keeping good recall. In addition, CLEVER operates at commit-time for preventing the introduction of faults before they reach the code repository. Through interactions with Ubisoft developers, we found that this integrates well with the workflow of developers.

2.2 Automatic Patch Generation

One feature of CLEVER is the ability to propose fixes that can help developers correct the detected risky commit. This is similar in principle to the work on automatic patch generation. Pan *et al.* and Kim *et al.* proposed two approaches that extract and apply fix patterns [22, 36]. Pan *et al.* identified 27 patterns and were able to fix 45.7% - 63.6% of bugs using one of the proposed patterns. The patterns found by Kim *et al.* are mined from human-written patches and were able to successfully generate patches for 27 out of 119 bugs. The tool by Kim *et al.*, named PAR, is similar to the second part of CLEVER where we propose fixes. Our approach also mines potential fixes from human-written patches found in the historical data. In our work, we do not generate patches, but instead propose known patches to developers for further assessment. It has also been shown that patch generation is useful in understanding and debugging the causes of faults [44].

Despite the advances in the field of automatic patch generation, this task remains overly complex. Developers expect from tools high quality patches that can be safely deployed. Many studies proposed a classification of what is considered an acceptable quality patch for an automatically generated patch to be adopted in industry [7, 26, 27].

3 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 and their 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 1 large project within Ubisoft and have been evaluated with 11 other large projects. 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.

In the second process (Figure 3), CLEVER intercepts incoming commits before they leave a developer's 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

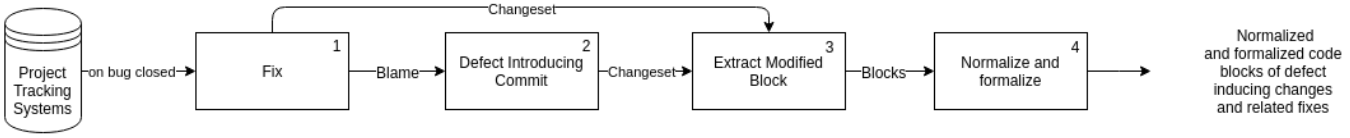

Figure 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

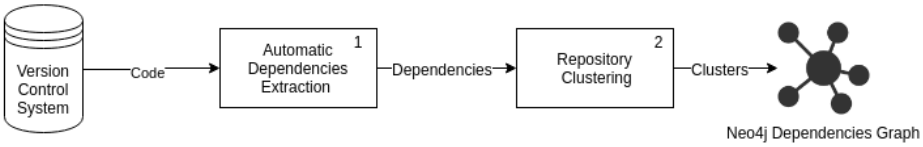

Figure 2: Clustering by dependency

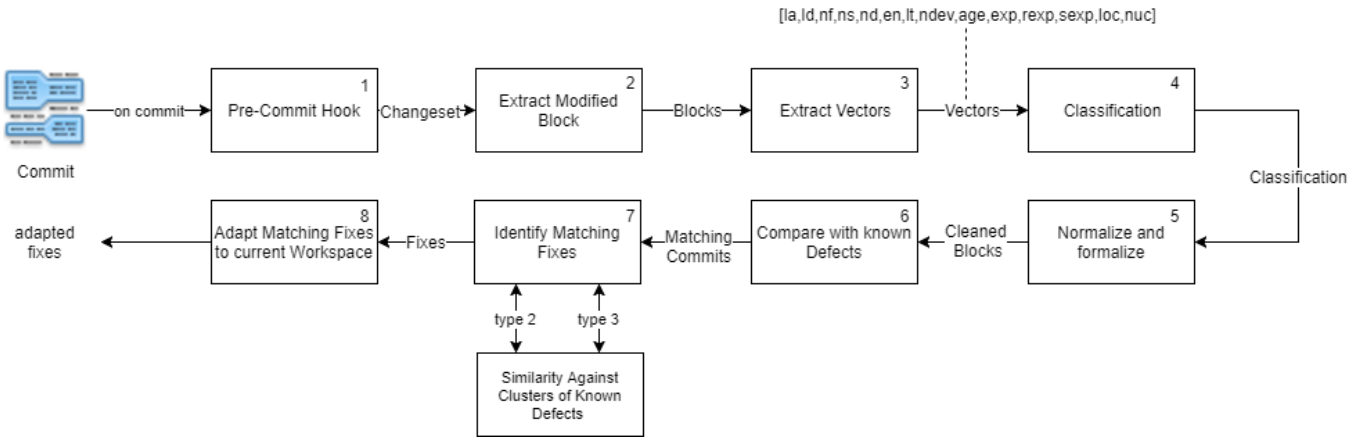

Figure 3: Classifying incoming commits and proposing fixes

versioning systems such as *Git*. 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 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 will 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 3.2. 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 blocks that are modified by the developer and then compare them to code blocks of known fault-introducing commits.

3.1 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 in Figure 4. In total, we discovered 405 distinct dependencies that internal and external both. The resulting partitioning is shown in Figure 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

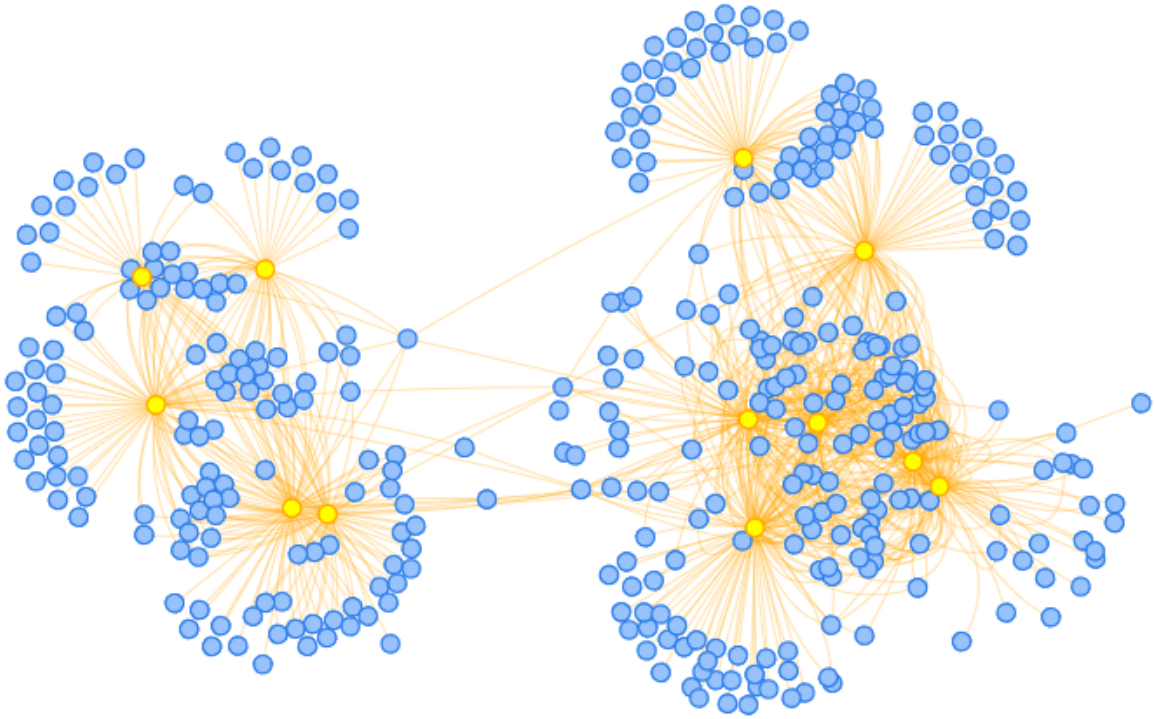

Figure 4: Dependency Graph

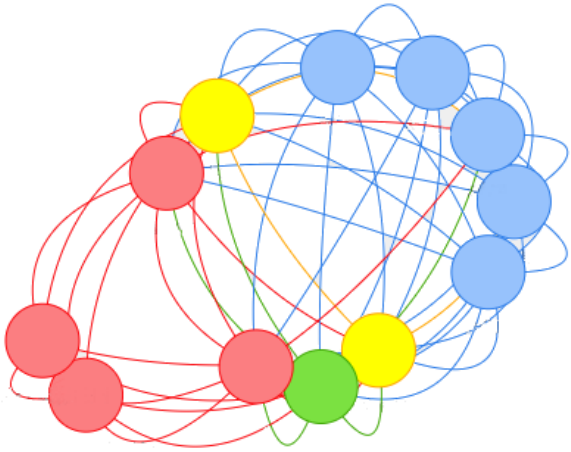

Figure 5: Clusters

Girvan–Newman algorithm [10, 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, 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.

3.2 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 well-known SZZ algorithm presented by Kim et al. [23].

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 [3].

3.3 Building a Metric-Based Model

We adapted Commit-guru [38] for building the metric-based model. Commit-guru uses a list of keywords proposed by Hindle *et al.* [14] 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 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 5. Finally, we had to rewrite Commit-guru in GoLang for performance and internal reasons.

3.4 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 compare 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 α 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 [8, 16, 17, 29, 30, 45], we chose the NICAD clone detector because it is freely available and has shown to perform well [6]. 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 [21]. 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 [39]. This functionality allowed us to detect Segments 1 and 2 as a clone pair, as shown by Table 1, 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 [8].

The extracted, pretty-printed, normalized filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [15]. Then, a percentage of unique statements can be computed and, given the threshold α, the blocks are marked as clones.

3.5 Classifying Incoming Commits

As discusses in Section 3.3, a new commit goes through the metric-based model first (Steps 1 to 4). If the commit is 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. We estimate that, in an average workday, if all commits had to be compared against all signatures on our cluster it would take around 25 minutes to process a commit. In comparison, it takes, in average, 3.75 seconds with the current approach.

3.6 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 top 1 matching defect-commit and present it to the developer. We believe that this makes CLEVER a practical approach. Developers can understand why a given modification has been reported as risky by looking at code instead of simple metrics as in the case of the studies reported in [18, 38].

Finally, using the fixes of past defects, we can provide a solution, in the form of a contextualised diff, to 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 it applies in their situation.

4 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.

4.1 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.

4.2 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 project groups of the company.

4.3 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 3.2. 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 [23]. 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 [1, 9, 19, 25, 28].

4.4 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 4.3. If at t_4, c_3 is pushed to p_2 and classify by the metric-based classifier as *risky*, we extract c_3 blocks and compares them with the ones of c_1. If $c3$ and $c1$ are a match 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.

Table 1: Pretty-Printing Example

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

4.5 Evaluation Measures

Similar to prior work (e.g., [19, 43]), 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 [4, 20, 38, 41], if a defect is not reported within six months then it is not considered.

5 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 each. 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 [38] 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., [18, 25]). Commit-guru is also open source and easy to use.

5.1 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.* [40] 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 with the second phase of CLEVER (clone detection) there is considerable reduction in 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).

5.2 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.

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?"* for each fix.

The review session took around 60 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:

- 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 2 shows answers 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 see 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.

Table 2: Workshop results

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
P1	Accepted	Rejected	Accepted	Accepted	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P2	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P3	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P4	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Accepted	Rejected	Accepted	Accepted	Unsure
P5	Accepted	Rejected	Accepted	Accepted	Unsure	Accepted	Unsure	Rejected	Rejected	Accepted	Accepted	Unsure
P6	Accepted	Rejected	Accepted	Unsure	Unsure	Accepted	Unsure	Accepted	Rejected	Accepted	Accepted	Unsure

*F*2 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?"* they rejected it. In this occurrence, the proposed fix was not applicable.

*F*4 was accepted by two reviewers and marked as unsure by the other participants. 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 *F*4 was applicable or not. *F*5 and *F*7 were classified as unsure by all our participants for the same reasons.

*F*8 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.

*F*12 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 was out of scope of this session focusing on their system.

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 generated code rather than the code generator. These aspects of CLEVER needs to be improved.

6 DISCUSSION

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

6.1 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-phases 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.

6.2 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).

7 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 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. Moreover, we will investigate how to improve the fixes proposed by CLEVER to add contextual information to help developers better assess the applicability of the fixes.

8 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.

ACKNOWLEDGMENTS

We are thankful to the software development team at Ubisoft for their participations to the study and their assessment of the effectiveness of CLEVER.

REFERENCES

[1] Bhattacharya, P. and Neamtiu, I. 2011. Bug-fix time prediction models: can we do better? *Proceeding of the international conference on mining software repositories* (New York, New York, USA, May 2011), 207–210.

[2] Briand, L. et al. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*. 25, 1 (1999), 91–121. DOI:https://doi.org/10.1109/32.748920.

[3] Bultena, B. and Ruskey, F. 1998. An Eades-McKay algorithm for well-formed parentheses strings. *Information Processing Letters*. 68, 5 (1998), 255–259.

[4] Chen, T.-h. et al. 2014. An Empirical Study of Dormant Bugs Categories and Subject Descriptors. *Proceedings of the international conference on mining software repository* (2014), 82–91.

[5] Chidamber, S. and Kemerer, C. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 20, 6 (Jun. 1994), 476–493. DOI:https://doi.org/10.1109/32.295895.

[6] Cordy, J.R. and Roy, C.K. 2011. The NiCad Clone Detector. *Proceedings of the international conference on program comprehension* (Jun. 2011), 219–220.

[7] Dallmeier, V. et al. 2009. Generating Fixes from Object Behavior Anomalies. *Proceedings of the international conference on automated software engineering* (2009), 550–554.

[8] Ducasse, S. et al. 1999. A Language Independent Approach for Detecting Duplicated Code. *Proceedings of the international conference on software maintenance* (1999), 109–118.

[9] El Emam, K. et al. 2001. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*. 56, 1 (Feb. 2001), 63–75. DOI:https://doi.org/10.1016/S0164-1212(00)00086-8.

[10] Girvan, M. and Newman, M.E.J. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*. 99, 12 (Jun. 2002), 7821–7826. DOI:https://doi.org/10.1073/pnas.122653799.

[11] Gyimothy, T. et al. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*. 31, 10 (Oct. 2005), 897–910. DOI:https://doi.org/10.1109/TSE.2005.112.

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

[13] Hassan, A.E. 2009. Predicting faults using the complexity of code changes. *Proceedings of the international conference on software engineering* (May 2009), 78–88.

[14] Hindle, A. et al. 2008. What do large commits tell us?: a taxonomical study of large commits. *Proceedings of the international workshop on mining software repositories* (New York, New York, USA, 2008), 99–108.

[15] Hunt, J.W. and Szymanski, T.G. 1977. A fast algorithm for computing longest common subsequences. *Communications of the*

ACM. 20, 5 (May 1977), 350–353. DOI:https://doi.org/10.1145/359581.359603.

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

[17] Johnson, J.H. 1994. Visualizing textual redundancy in legacy source. *Proceedings of the conference of the centre for advanced studies on collaborative research* (Oct. 1994), 32.

[18] Kamei, Y. et al. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*. 39, 6 (Jun. 2013), 757–773. DOI:https://doi.org/10.1109/TSE.2012.70.

[19] Kamei, Y. et al. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*. 39, 6 (Jun. 2013), 757–773. DOI:https://doi.org/10.1109/TSE.2012.70.

[20] Kamei, Y. et al. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*. 39, 6 (Jun. 2013), 757–773. DOI:https://doi.org/10.1109/TSE.2012.70.

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

[22] Kim, D. et al. 2013. Automatic patch generation learned from human-written patches. *Proceedings of the international conference on software engineering* (May 2013), 802–811.

[23] Kim, S. et al. 2006. Automatic Identification of Bug-Introducing Changes. *Proceedings of the international conference on automated software engineering* (2006), 81–90.

[24] Kim, S. et al. 2007. Predicting Faults from Cached History. *Proceedings of the international conference on software engineering* (May 2007), 489–498.

[25] Kpodjedo, S. et al. 2010. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*. 16, 1 (Dec. 2010), 141–175. DOI:https://doi.org/10.1007/s10664-010-9151-7.

[26] Le Goues, C. et al. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. *Proceedings of the international conference on software engineering* (2012), 3–13.

[27] Le, X.-B.D. et al. 2015. Should fixing these failures be delegated to automated program repair? *Proceedings of the international symposium on software reliability engineering* (2015), 427–437.

[28] Lee, T. et al. 2011. Micro interaction metrics for defect prediction. *Proceedings of the european conference on foundations of software engineering* (New York, New York, USA, 2011), 311–231.

[29] Manber, U. 1994. Finding similar files in a large file system. *Proceedings of the usenix winter* (Jan. 1994), 1–10.

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

[31] Nagappan, N. and Ball, T. 2005. Static analysis tools as early indicators of pre-release defect density. *Proceedings of the international conference on software engineering* (New York, New York, USA, May 2005), 580–586.

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

[33] Nagappan, N. et al. 2006. Mining metrics to predict component failures. *Proceeding of the international conference on software engineering* (New York, New York, USA, May 2006), 452–461.

[34] Newman, M.E.J. and Girvan, M. 2004. Finding and evaluating community structure in networks. *Physical Review E*. 69, 2 (Feb. 2004), 026113. DOI:https://doi.org/10.1103/PhysRevE.69.026113.

[35] Ostrand, T. et al. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*. 31, 4 (Apr. 2005), 340–355. DOI:https://doi.org/10.1109/TSE.2005.49.

[36] Pan, K. et al. 2008. Toward an understanding of bug fix patterns. *Empirical Software Engineering*. 14, 3 (Aug. 2008), 286–315. DOI:https://doi.org/10.1007/s10664-008-9077-5.

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

[38] Rosen, C. et al. 2015. Commit guru: analytics and risk prediction of software commits. *Proceedings of the joint meeting on foundations of software engineering* (New York, New York, USA, Aug. 2015), 966–969.

[39] Roy, C.K. 2009. *Detection and Analysis of Near-Miss Software Clones*. Queen's University.

[40] Roy, C.K. and Cordy, J.R. 2008. An Empirical Study of Function Clones in Open Source Software. *Proceedings of the working conference on reverse engineering* (Oct. 2008), 81–90.

[41] Shivaji, S. et al. 2013. Reducing Features to Improve Code Change-Based Bug Prediction. *IEEE Transactions on Software Engineering*. 39, 4 (2013), 552–569.

[42] Subramanyam, R. and Krishnan, M. 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*. 29, 4 (Apr. 2003), 297–310. DOI:https://doi.org/10.1109/TSE.2003.1191795.

[43] Sunghun Kim, S. et al. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*. 34, 2 (Mar. 2008), 181–196. DOI:https://doi.org/10.1109/TSE.2007.70773.

[44] Tao, Y. et al. 2014. Automatically generated patches as debugging aids: a human study. *Proceedings of the international symposium on foundations of software engineering* (2014), 64–74.

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

[46] Zimmermann, T. and Nagappan, N. 2008. Predicting defects using network analysis on dependency graphs. *Proceedings of the international conference on software engineering* (New York, New York, USA, May 2008), 531.

[47] Zimmermann, T. et al. 2007. Predicting Defects for Eclipse. *Proceedings of the international workshop on predictor models in software engineering* (May 2007), 9.