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

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

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

1 INTRODUCTION

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-phase process for intercepting risky commits before they reach the central repository. 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:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR ’18, May 28–29, 2018, Gothenburg, Sweden © 2018 Association for Computing Machinery. ACM ISBN 978-1-4503-5716-6/18/05...$15.00 https://doi.org/10.1145/3196398.3196438
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 setting where software systems tend to have many dependencies that make them vulnerable to the same faults.

CLEVER was developed in a context of a research project between Concordia University and Ubisoft. 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 at 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. [36] 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. [40] and Gyimothy et al. [11].

Further improvements to these metrics have been proposed by Nagappan et al. [29, 31] 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. proposed a technique that uses data mined from source code repository such as churns to assess the quality of a change [30]. Hassan et al and Ostrand et al used past changes and defects to predict buggy locations [12], [33]. Their methods rely on various heuristics to identify the locations that are most likely to introduce a defect. Kim et al [22] 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 [35].

Other studies that aim to predict risky changes use the entropy of a given change [13, 41] and the size of the change combined with files being changed [18].

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 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 [20, 34]. 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 [42].

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, 24, 25].

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 on 12 large Ubisoft 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 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. Dependencies can be internal (i.e. library developed at Ubisoft) or external (i.e. library provided by third parties). The resulting partitioning is shown in Figure 5.

At Ubisoft, 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 [10, 32], 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 [32]. 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.

#### 3.2.1 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. [21].

#### 3.2.2 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

```plaintext
for i ← 0 to size_of changesets do
    Block[] blocks ← extract_blocks(changesets);
    for j ← 0 to size_of blocks do
        write blocks[j];
end
```

**Function** extract_blocks(Changeset cs)

```plaintext
if cs is unbalanced right then
    cs ← expand_left(cs);
else if cs is unbalanced left then
    cs ← expand_right(cs);
end
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.
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

@@ -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 changset 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 [36] 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), \( en \) (distribution of modified code across each file), \( lt \) (lines of code in each file (sum) before the commit), \( ndev \) (the number of developers that modified 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 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 or a bug in the internal project tracking system. Using internal pre-commit hooks, Ubisoft developers must link every commit to a given task.
Figure 4: Dependency Graph

Figure 5: Clusters
If the task identified 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 [43, 44]. 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 \( \alpha \) 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, 27, 28, 45], we chose the NICAD clone detector because it is freely available and has shown to perform well [6].

NICAD can detect Type 1, 2 and 3 software clones [19]. Type 1 clones are copy-pasted blocks of code that only differ from each other in terms of non-code artifacts such as indentation, whitespaces, comments and so on. Type 2 clones are blocks of code that are syntactically identical except literals, identifiers, and types that can be modified. Also, Type 2 clones share the particularities of Type 1 about indentation, whitespaces, and comments. Type 3 clones are similar to Type 2 clones in terms of modification of literals, identifiers, types, indentation, whitespaces, and comments but also contain added or deleted code statements.

The problem with the current implementation of NICAD is that it only considers complete Java, C#, and C files. We improved NICAD to process blocks that come from commit-diffs. This is because the current version of NICAD can 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 [19]. 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 [58]. 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 \( \alpha \), the blocks are marked as clones.

### 3.5 Classifying Incoming Commits

As discussed 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 8 in 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 more time consuming. We estimate that, in an average workday (i.e. thousands of commits), if all commits had to be compared against all signatures on the same cluster we used for our experiments it would take around 25 minutes to process a commit with the current processing power dedicated to CLEVER. In comparison, it takes, in average, 3.75 seconds with the current two-step 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-commits 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, 36].

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.

All the proposed fixes will come from projects in the same cluster as the project where the risky commit is. Thus, developers have access to fixes that should be easier to understand as they come from projects similar to theirs inside the company.

### 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, hundreds of millions of lines of code and hundreds of thousands of commits. All 12 systems are AAA video games.

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 are accurate and reflects well the various project groups of the company. The clusters are used for decreasing the rate of false positive. In addition, fixes mined across projects but within the cluster are qualitative as shown in our experiments.

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 [21]. 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, 18, 23, 26].

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 as presented in Figure 6. 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 \( t_4 \) we know that \( c_3 \) introduced a defect using the process described in Section 4.3. If at \( t_4 \), \( c_3 \) is pushed to \( p_2 \) and classified by the metric-based classifier as risky, we extract \( c_3 \) blocks and compare them with the ones of \( c_1 \). If \( c_3 \) and \( c_1 \) 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 \( c_3 \).

While this example explains the processes of CLEVER it does not encompass all the cases. Indeed, the user that experiences the defect can be internal (i.e. another developer, a tester, ...) or external (i.e. a player). In addition, many other projects receive commits in parallel and they are all to be compared with all the known signatures.

4.5 Evaluation Measures

Similar to prior work (e.g., [18, 41]), 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
In order to validate the quality of the fixes proposed by CLEVER, we conducted an internal workshop where we invited a number of people from 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 question “Is the proposed fix applicable in the given situation?” for each fix.

The review session took around 50 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 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.

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 bug and 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.

F4 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 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 participants 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 was 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?
All the participants answered the first question favourably. They also proposed to embed CLEAR with Ubisoft’s quality assurance tool suite. The participants reported that the most useful aspects of CLEAR are:

- Ability to leverage many years of historical data of inter-related projects, hence allowing development teams to share their experiences in fixing bugs.
- Easy integration of CLEAR into developers’ work flow based on the tool’s ability to operate at commit-time.
- Precision and recall of the tool (79% and 65% respectively) demonstrating CLEAR’s capabilities to catch many defects that would otherwise end up in the code repository. For the second question, the participants proposed to add a feedback loop to CLEAR where the input of software developers is taken into account during classification. The objective is to reduce the number of false negatives (risky commits that are flagged as non-risky) and false positives (non-risky commits that are flagged as risky). The feedback loop mechanism would work as follows: When a commit is misclassified by the tool, the software developer can choose to ignore CLEAR’s recommendation and report the misclassified commit. If the fix proposition is not used, then, we would give that particular pattern less strength over other patterns automatically.

We do not need manual input from the user because CLEAR knows the state of the commit before the recommendation and after. If both versions are identical then we can mark the recommendation as not helpful. This way, we can also compensate for human error (i.e., a developer rejecting CLEAR recommendation when the commit was indeed introducing a defect. We would know this by using the same processes that allowed us to build our database of defect-commits as described in Section 3.2. This feature is currently under development.

It is worth noting that Ubisoft developers who participated to this study did not think that CLEAR fixes that were deemed irrelevant were a barrier to the deployment of CLEAR. In their point of view, the performance of CLEAR in terms of classification should make a significant impact as suspicious commits will receive extended reviews and/or further investigations.

We are also investigating the use of adaptive learning techniques to improve the classification mechanism of CLEAR. In addition to this, the participants discussed the limitation of CLEAR as to its inability to deal with automatically generated code. We are currently working with Ubisoft’s developers to address this limitation.

5.3 Deployment of CLEAR at Ubisoft

CLEAR will soon be rolled out at Ubisoft. It will be made available to thousands of developers across various divisions. Our research team provided on-site training of this new tool. In addition, Ubisoft developed an instructional video to support the launch of CLEAR and raise awareness about the tool. Our research team is currently monitoring the use of CLEAR at Ubisoft to evaluate its adoption (usage, barriers, etc.).

6 DISCUSSION

In this section, we share the lessons learned, discuss the limitations of CLEAR, and present threats to validity of our study.

6.1 Lessons Learned

6.1.1 Understanding the industrial context: Throughout the design of CLEAR, we made many design decisions that were triggered by the discussions we had with Ubisoft developers. Some of the key decisions that we made included having CLEAR operate on clusters of inter-related systems and combining metric-based and code matching techniques into a two-phase approach. These decisions were not only critical in obtaining an improved accuracy, but also in proposing effective fixes that guide developers. From our interactions with Ubisoft developers, it was also important for us to come up with a solution that integrates well with the workflow of Ubisoft developers. This motivated the use of commit-time and the integration of CLEAR with Ubisoft version control systems. The key lesson here is the importance of understanding the industrial context by working with the company’s development teams. This collaboration is also an enabler for the adoption of tools, developed in the context of research projects.

6.1.2 Leveraging an iterative process: Throughout this research project, we followed an iterative and incremental process. The results of each iteration were presented to Ubisoft developers for feedback. Adjustments were made as needed, before the subsequent iteration started. This process was not only helpful in keeping the project on track, but also in producing “quick wins” as a way of showing practical results from each iteration. Examples of such “quick wins” include the creation of the defect introduction landscape at Ubisoft in terms of number defects, time to fix defects and time to discover defects organization wide. In addition, the computed clusters of similar projects turned out to be useful for upper management in order to organize collaborations between teams belonging to different projects.
6.1.3 Communicating effectively: During the development of CLEVER, we needed to constantly communicate the steps of our research to developers and project owners. It was important to adopt a communication strategy suitable to each stakeholder. For example, in our meetings with management, we focused more on the ability of CLEVER to improve code quality and reduce maintenance costs instead of the technical details of the proposed approach. Developers, on the other hand, were interested in the potential of CLEVER and its integration with their work environment.

6.1.4 Underestimating the time needed for full deployment of CLEVER. Part of our mandate was to develop a working tool. It took a tremendous amount of time and effort to develop a production tool version of CLEVER, which required a lot of engineering work. We recognize that we underestimated the complexity of this task. Examples of deliverables we had to produce include automating the acquisition of new commits, presenting the recommendations to the developers, building grammars for various programming languages, creating APIs that interact with any types of client systems, authentication, and authorization of end-users, etc. Overall, the machine learning code represents less than 5% of our code base. The lesson here is to manage expectations and to better estimate the project time and effort from an end to end perspective, and not only the research part.

6.2 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 to a single system will most likely be less effective. 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.

6.3 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. CLEVER is still in its infancy and we expect it to be available this year to thousands of developers.

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.

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. We are also thankful to NSERC (Natural Sciences and Engineering Research Council of Canada) which financed part of this research.

REFERENCES


