CLEVERT: 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
Montréal, QC, Canada
mathieu.nayrolles@ubisoft.com

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

I. Introduction

Automatic prevention and resolution of faults is an important research topic in the field of software maintenance and evolution. A growing line of research focuses on the problem of preventing the introduction of faults by detecting risky commits (commits that may potentially introduce a fault in the system) before reaching the central code repository. We refer to this as just-in-time fault detection/prevention. Recent techniques such as the work of XXX et al. [1] rely on building models from historical commits using code and process metrics (e.g., code complexity, the experience of the developers, etc.) as the main features. These models are later used to classify new commits as risky or not. The problem with these techniques is that they generate high false positive rates. In addition, tools that implement these techniques such as Commit-guru [REF] do not provide any insights to developers on how to fix the risky commits. They simply return measurements that are often difficult to interpret by developers. In addition, they have been applied to single projects only, despite the fact that many projects share dependencies due to the reuse of libraries, which make them vulnerable to similar faults. In addition, existing techniques are 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 CLEVERT (Combining Levels of Bug Prevention and Resolution Techniques), that relies on a two-step process for intercepting risky commits before they reach the central repository. The first step 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 except that our model is built using commits from multiple inter-related projects. In the next step, CLEVERT uses clone detect to compare code blocks extracted from risky commits with those of known fault-introducing commits. CLEVERT does not 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. As we will show in the case study, this second step reduces the number of false positives. Another advantage of CLEVERT is that it uses commits that are used to fix previous fault-introducing commits to guide the developers on how to improve the risky commits. This way, CLEVERT goes one step further than Commit-guru (and similar techniques) by providing developers with a potential fix for their risky commits.

CLEVERT was developed in collaboration with developers 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 with millions of files and commits, developed and maintained by more than 8,000 developers scattered across 29 locations in six continents.

When applied to 12 Ubisoft systems, CLEVERT detects risky commits with 79% precision and 65% recall, which outperforms the performances of Commit-guru, which is 66% precision and 63% recall when applied to the same dataset. In addition, more than XX% of the proposed fixes were judged highly relevant by software developers at Ubisoft, making CLEVERT an effective and practical appraoch for the detectiong and resolution of risky commits.

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

II. Related Work

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

A. File, Module and Risky Change Prediction

The majority of previous file/module-level prediction work used code or process metrics. Approaches using code metrics only use information from the code itself and do not use

any historical data. Chidamber and Kemerer published the well-known CK metrics suite [2] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [3]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [4].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [5], El Emam *et al.* [6], Subramanyam *et al.* [7] and Gyimothy *et al.* [8] for object-oriented designs. Service oriented designs have been far less studied than object oriented design as they are relatively new, but, Nayrolles *et al.* [9], [10], Demange *et al.* [11] and Palma *et al.* [12] used Moha *et al.* metric suites to detect software defects. All these approaches, proved software metrics to be useful at detecting software fault for object oriented and service oriented designs, respectively. More recently, Nagappan *et al.* [13], [14] and Zimmerman *et al.* [15], [16] further refined metrics-based detection by using statical analysis and call-graph analysis.

Other approaches use historical development data, often referred to as process metrics. Naggapan and Ball [17] studied the feasibility of using relative churn metrics to prediction buggy modules in the Windows Server 2003. Other work by Hassan *et al* and Ostrand *et al* used past changes and defects to predict buggy locations (e.g., [18], [19]). Hassan and Holt proposed an approach that highlights the top ten most susceptible locations to have a bug using heuristics based on file-level metrics [18]. They find that locations that have been recently modified and fixed locations are the most defect-prone. Similarly, Ostrand *et al.* [19] predict future crash location by combining the data from changed and past defect locations. They validate their approach on industrial systems at AT&T. They showed that data from prior changes and defects can effectively defect-prone locations for open-source and industrial systems. Kim *et al.* [20] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [18]. Rahman and Devanbu found that, in general, process-based metrics perform as good as or better than code-based metrics [21].

Other work focused on the prediction of risky changes. Kim et al. proposed the change classification problem, which predicts whether a change is buggy or clean [22]. Hassan [23] used the entropy of changes to predict risky changes. They find that the more complex a change is, the more likely it is to introduce a defect. Kamei *et al.* performed a large-scale empirical study on change classification [24]. They aforementioned studies find that size of a change and the history of the files being changed (i.e., how buggy they were in the past) are the best indicators of risky changes.

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

B. Transfer Defect Learning

Mathieu: To do after we complete V.C

C. Automatic Patch Generation

Since CLEVERT not only flags risky changes but also provides developers with fixes that have been applied in the past, automatic patch generation work is also related. Pan *et al.* [25] identified 27 bug fixing patterns that can be applied to fix software bugs in Java programs. They showed that between 45.7 - 63.6% of the bugs can be fixed with their patterns. Later, Kim *et al.* [26] generated patches from human-written patches and showed that their tool, PAR, successfully generated patches for 27 of 119 bugs. Tao *et al.* [27] also showed that automatically generated patches can assist developers in debugging tasks. Other work also focused on determining how to best generate acceptable and high quality patches, e.g. [28], [29], and determine what bugs are best fit for automatic patch generation [30].

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

III. The CLEVERT Approach

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

In the first process (Figures 1 and 2), CLEVERT manages events happening on project tracking systems to extract defect-introducing commits and commits that provided the 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 defect.

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

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

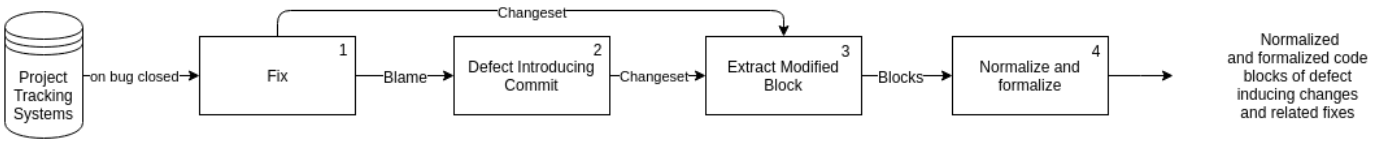

Fig. 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

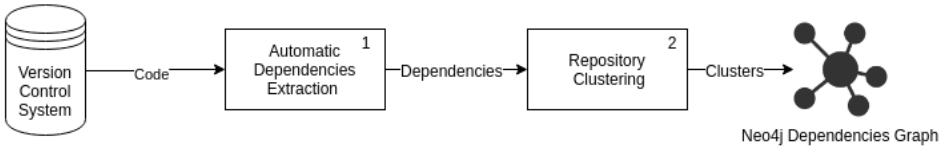

Fig. 2: Clustering by dependency

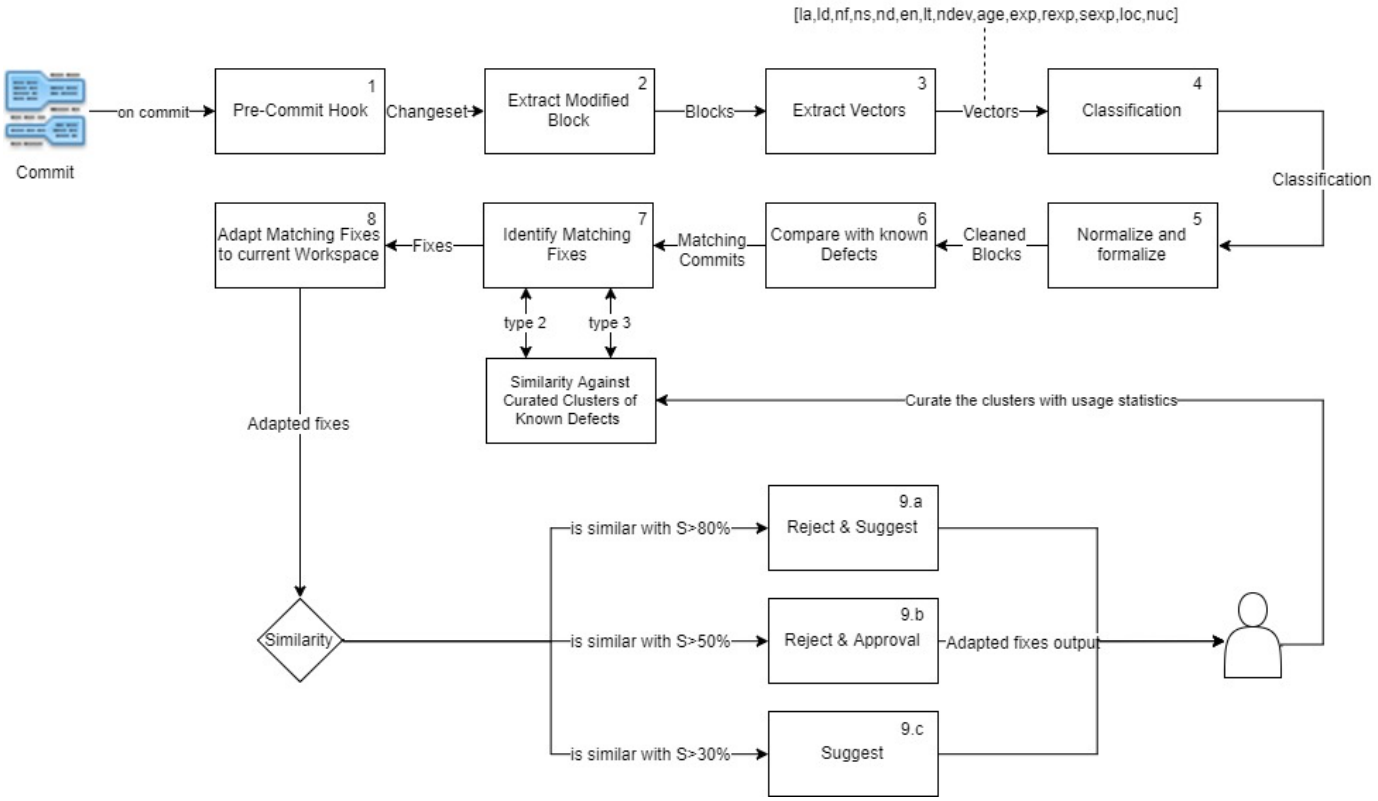

Fig. 3: Classifying incoming commits and proposing fixes

the commit at hand, specifying the reviewers who reviewed the commit at hand, and son. Implementing this part of CLEVERT as a pre-commit hook is an important step towards the integration of CLEVERT with the workflow of developers. Developers do not have to download, install and understand additional tools in order to use CLEVERT.

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

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

A. Clustering Project Repositories

We cluster projects according to their dependencies. The rationale is that projects that share dependencies are most likely to contain defects caused by misuse of these dependencies. In this step, the project dependencies are analysed and saved into a single NoSQL graph database as shown in Figure 2. A node corresponds to a project that is connected to other

projects on which it depends. Dependencies can be *external* or *internal* depending on whether the products are created in-house or supplied by a third-party. For confidentiality reasons, we cannot reveal the name of the projects involved in the project dependency graph. We show the 12 projects in yellow color with their dependencies in blue color 4. The resulting partitioning is shown in 5.

Internal dependencies are managed within the framework of a single repository, which makes their automatic extraction possible. The dependencies could also be automatically retrieved if 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 [31], [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.

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

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

Extracting Commits: CLEVERT listens to issue closing events happening on the project tracking system used at Ubisoft. Every time an issue is closed, CLEVERT retrieves the commit that was used to fix the issue (the fix-commit) as well as the one that introduced the defect (the defect-commit). To link fix-commits and their related issues we implemented the SZZ algorithm presented by Kim et al. [33].

Extracting Code Blocks: Algorithm 1 presents an overview of how 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.

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;
```

Data: $Changeset[]$ changesets;
$Block[]$ prior_blocks;
Result: Up to date blocks of the systems
1 **for** $i \leftarrow 0$ **to** $size\_of\ changesets$ **do**
2 Block[] blocks $\leftarrow extract\_blocks(changesets)$;
3 **for** $j \leftarrow 0$ **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 $cs \leftarrow expand\_left(cs)$;
10 **else if** $cs\ is\ unbalanced\ left$ **then**
11 $cs \leftarrow expand\_right(cs)$;
12 **end**
14 **return** $txl\_extract\_blocks(cs)$;

Algorithm 1: Overview of the Extract Blocks Operation

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 [34].

C. Building a Metric-Based Model

We build a metric-based model using 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 modified LOC across all files
- nuc: number of unique changes to the files

To this end, we adapted Commit-guru developed by Rosen _et al. [1]. We had to rewrite Commit-guru in GoLang for performance and internal reasons. Commit-guru's back-end has has three major components: ingestion, analysis, and prediction. The ingestion component is responsible for ingesting (i.e., downloading) a given repository. Once the repository is downloaded on a local server, each commit history is analysed.

Commit-guru classifies commits using a list of keywords proposed by Hindle *et al.* [35]. CLEVERT, on the other hand, uses internal project tracking system. In other words, it only

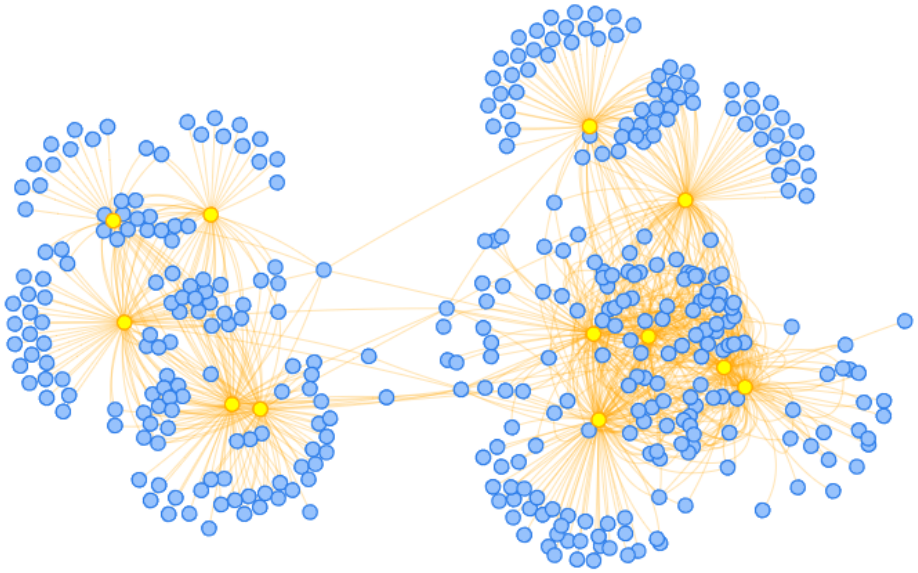

Fig. 4: Dependency Graph

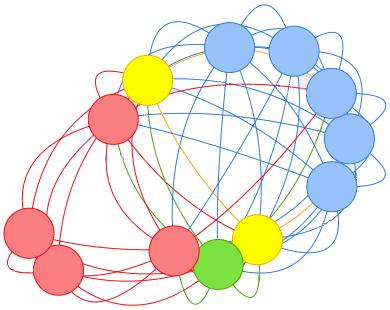

Fig. 5: Clusters

classifies commits as introducing a defect if they are the root cause of a fix linked to a crash in the internal project tracking system (JIRA). Project tracking system allows one to report unexpected system behaviour and managers can assign them to developers.

Using the internal pre-commit hook, developers must link every commit to a give 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.

The SZZ algorithm, used by Commit-guru and CLEVERT has been shown to be effective in detecting risky commits [1], [24].

D. Comparing Code Blocks

Each time a developer makes a commit, CLEVERT intercepts it using a pre-commit hook, extracts the corresponding code block (in a similar way as in the previous phase), and compares it to the code blocks of historical defect-commits. If there is a match, then the new commit is deemed to be risky. A threshold S is used to assess the extent beyond which two commits are considered similar.

To compare the extracted blocks to the ones in the database, we resort to clone detection techniques, more specifically, text-based clone detection techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based comparisons) would require a complete program to work, i.e., a program that compiles. In the relatively wide-range of tools and techniques that exist to detect clones by considering code as text [36]–[42] we had to build our own text-based clone detector for several reasons. First, the clone detector should have the ability, once a clone has been identified, to transform the matching clones for them to match the workspace of the developer regarding variables names and data structure.

While classical clone detector aims to detect clone pairs for removal and/or managing them, we have another goal. We want not only to match clone pairs by abstracting them, but also to transform one block into another.

The contextualised fix that we proposed to the developers can be syntactically incorrect as they are based on imcomplete blocks of code. Hence, they cannot be used directly by developers. We simply believe that the contextualised fixes are easier to understand and apply as the *look* familiar to the code the developer is currently attempting to submit.

Our clone detector can detect Types 1, 2 and 3 software clones [43]. Type 1 clones are copy-pasted blocks of code that only differ from each other in terms of non-code artefacts 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. CLEVERT detects Type 3 clones since they can contain added or deleted code statements, which make them suitable for comparing commit code blocks.

For our clone detector, we reuse the pretty-printing strategy from Roy *et al.* where statements are broken down into several lines [44]. Furthermore, in th process, statements can be shows how this can improve the accuracy of clone detection with three `for` statements `for (i=0; i<10; i++)`, `for (i=1; i<10; i++)` and `for (j=2; j<100; j++)`.

The pretty-printing allows us to detect Segments 1 and 2 as a clone pair, as shown by Table I, because only the initialisation of i changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc [40].

The extracted, pretty-printed, normal used and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [45]. Then, a percentage of unique statements can be computed and, given the threshold α, the blocks are marked as clones.

Another important aspect of the design of CLEVERT is the ability to provide guidance to developers on how to improve risky commits. We achieve this by extracting from the database the fix-commit corresponding to the matching defect-commit and present it to the developer. We believe that this makes CLEVERT a practical approach for the developers as they will know why a given modification has been reported as risky in terms of code; this is something that is not supported by techniques based on statistical models (e.g., [1], [46]).

E. Classifying Incoming Commits

The classification of incoming commits within CLEVERT is a two-step process. First, the commit goes through the statical part of the approach (Steps 1 to 4). If the statical part of the approach classifies the commit 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 the Steps 5 to 9. It is important to note that we do not output a *risky* classification if we are unable to find a match in known defect-introducing signatures in our historical data. Each alarm is accompanied with a contextualised changeset that serves as a potential solution. We believe that this would encourage developers to adopt CLEVERT.

Finally, this two-step classification allows us to significantly reduce the time required, in average, to classify a commit. If each commit has to be analysed against all known signatures in terms of code clone similarity, then, it would take more to process. From this perspective, the metric-based model can be seen as a first-level filter that finds suspicious commits.

IV. Case Study Setup

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

A. Project Repository Selection

In collaboration with Ubisoft developers, we selected 12 major software systems developed at Ubisoft to evaluate the effectiveness of CLEVERT. 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 name nor the characteristics of these projects are provided. We can however disclose that the size of these systems altogether consists of millions of lines of code.

B. Project Dependency Analysis

Figure 4 shows the project dependency graph. As shown in Figure 4, these projects are 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 game,s whereas the other can be another family. This project partitioning has been successfully validated by 11 software developers at Ubisoft.

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

To build the database that we can use to assess the performance of CLEVERT, we use the same process as discussed in Section III-B. We retrieve the full history of each project and label commits as defect-commits if they appear to be linked to a closed issue using the SZZ algorithm [33]. This baseline is used to compute the precision and recall of CLEVERT. Each time CLEVERT 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 [6], [24], [47]–[49].

D. Process of Comparing New Commits

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

TABLE I: Pretty-Printing Example

Segment 1	Segment 2	Segment 3	S1 & S2	S1 & S3	S2 & S3

for (
i = 0;
i >10;
i++)
	for (
	i = 1;
	i >10;
	i++)
		for (
		j = 2;
		j >100;
		j++)
			1
			0
			1
			1
				1
				0
				0
				0
					1
					0
					0
					0

Total Matches			3	1	1
Total Mismatches			1	3	3

TABLE II: Bubble Sort Example

```
boolean t = true;
while(t){
    t = false;
    for(int i = 0; i < mas.length - 1; i++){
        if (mas[i] > mas[i+1]){
            String temp = mas[i];
            mas[i] = mas[i+1];
            mas[i+1] = temp;
            t = true;
        }
    }
}
```

```
for(int j = tab.length - 1; j >= 0; j--){
    for(int i = 0; i < j; i++){
        if (tab[i] > tab[i+1]){
            int tmp = tab[i];
            tab[i] = tab[i+1];
            tab[i+1] = tmp;
        }
    }
}
```

```
boolean t = true;
##=#;
while(
#)
{
#=#;
for(
##=#;
#<#.#-#;
#++)
{
if(
#[#]>#[#+#])
{
##=#[#];
#[#]=#[#+#];
#[#+#]=#;
#=#;
}
}
}
```

```
for(
##=#.#-#;
#>=#;
#--)
{
for(
##=#;
#<#;
#++)
{
if(
#[#]>#[#+#])
{
##=#[#];
#[#]=#[#+#];
#[#+#]=#;
}
}
}
```

TABLE III: Bubble Sort Example Fix

```
for(int j = tab.length - 1; j >= 0; j--){
    for(int i = 0; i < j; i++){
        if (tab[i] > tab[i+1]){
            int tmp = tab[i];
            tab[i] = tab[i+1];
            tab[i+1] = tmp;
        }
    }
}
```

```
for(int j = mas.length - 1; j >= 0; j--){
    for(int i = 0; i < j; i++){
        if (mas[i] > mas[i+1]){
            int temp = mas[i];
            mas[i] = mas[i+1];
            mas[i+1] = temp;
        }
    }
}
```

E. Evaluation Measures

Similar to prior work focusing on risky commits (e.g., [22], [24]), 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 CLEVERT
- FP: is the number of healthy commits that were classified by CLEVERT as risky
- FN: is the number of defect introducing-commits that were not detected by CLEVERT
- Precision: TP / (TP + FP)
- Recall: TP / (TP + FN)
- F_1-measure: 2.(precision.recall)/(precision+recall)

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

V. Case Study Results

In this section, we show the effectiveness of CLEVERT 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 projects, and if so, what would be the accuracy?*

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

In the following subsections, we provide insights on the performance of CLEVERT by comparing it to Commit-guru [1] alone, i.e., an approach that relies only on metric-based models. We chose Commit-guru because it has been shown to outperform other techniques (e.g., [REF]). Commit-guru is also open source and easy to use. In addition, we compare CLEVERT to itself when applied to all the projects, i.e., without applying the clustering step. This is important in order to validate the usefulness of the clustering step.

A. Performance of CLEVERT

CLEVERT detects risky commits with a 79.10% precision and a 65.61% recall in average on the 12 projects we analysed at Ubisoft so far while using the threshold displayed on 3. Any commit satisfying the two-steps classification but have a less than 30% will be classified as *non-risky*. When applied to the same projects, Commit-guru achieves an average precision and recall 66.71% and 63.01%, respectively.

It is important to note that we did not evaluate the effect of the feedback loop, where developers indicate if they found the proposed fix applicable, on the precision and the recall. While CLEVERT is currently beta-tested by the teams of one product, we need to gather at least a year of utilisation to measure the impact. We intent to report of this as a future work. We could expect the precision to go up and the recall to go down as bugs signatures are discarded from the cluster if no new bugs signatures were to be added. Indeed, it is unclear what the effect of adding signature will discarding other over time will be.

B. Cluster Classifier Performances

Our clusters, computed with the dependencies of each project allow to solve one major problem in defect learning and prediction: cold start [53]. Several approaches have already been proposed to solve these problems, however, they all require to classify *similar* projects by hand and then, manipulate the feature space in order to adapt the model learnt from one project to the other [54]. With our approach, the *similarity* between projects is computed automatically and, results show that we do not actually need to manipulate the feature space. Figures 6, 7 and 8 show the performance of the CLEVERT classification, in terms of ROC-curve and recall over precision curves, for

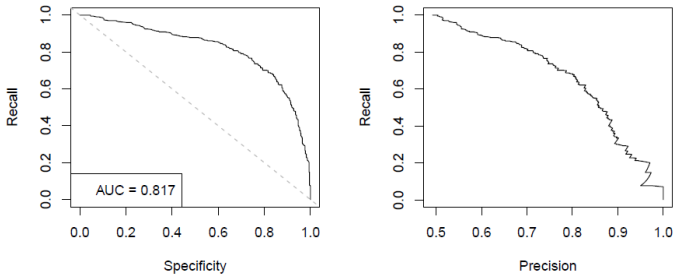

Fig. 6: Performances of Misfire while cold-starting the last project in the blue cluster

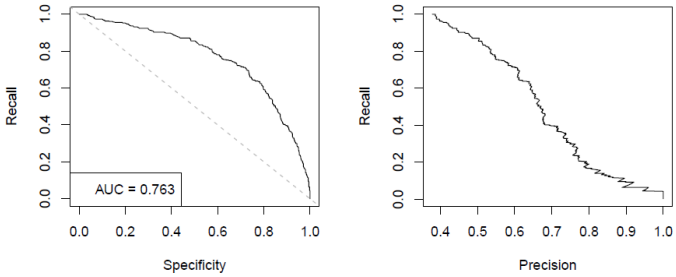

Fig. 7: Performances of Misfire while cold-starting the last project in the yellow cluster

the first thousand commit of the last project (chronologically) for the blue, yellow and red clusters presented in Figure 5. The left sides or the graph are low cutoff (aggressive) while the right sides are high cutoff (conservative). The area under the ROC curves are 0.817, 0.763 and 0.806 for the blue, yellow and red clusters, respectively.

These results show that not only the clusters are effective in identifying similar projects for defect learning transfer but also provide excellent performance while starting a new project. As new projects mature, their defects would be integrated into the model in order to improve it. To confirm this, we ran an experiment with the blue cluster where we first apply the model learnt from other members of the cluster for the first thousand commits. The performance of this model is 75.1% precision, 57.6% recall for the first thousand commits. After the first thousand commits, we take the commits of the day (for each day until the end of the project) and rebuild the model by adding the commits of the day to the ones already known

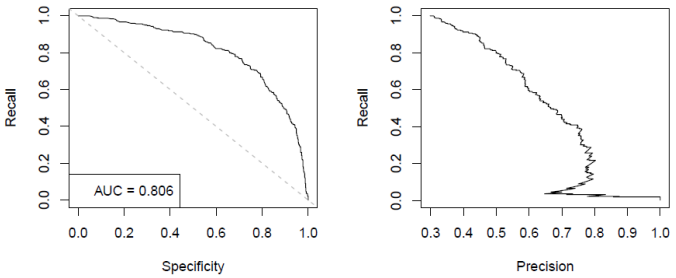

Fig. 8: Performances of Misfire while cold-starting the last project in the red cluster

TABLE IV: Workshop results

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

from the clusters. Overall, combining the commits from the project at-hand in a nightly fashion with the commits known from the cluster allowed us to correctly classified an additional 2.05% of commits in the last 30% of the project life compared to only using the commits from the project. This shows that, in addition to providing a viable alternative to a cold-start, our clusters also allow us to enhance the performances of the model at all time and no only at the start.

C. Analysis of the Quality of the Fixes Proposed by CLEVERT

In order to validate the quality of the fixes proposed by CLEVERT, we conducted an internal workshop. In this workshop, participants were asked to assess the quality of proposed fixed when presented with the original buggy commit, the origin fix for this commit and the proposed one. The attendance was composed of two software architects, two programmers, one technical lead and one IT project manager.

Our pannel was asked to collectively review 12 randomly selected fixes coming from one given system. All the participants are familiar with the system. The participants reviewed the fixes over a 70 minutes period, and we report their opinions individually. In addition to assessing the quality of the proposed fixes and their likeliness of the proposed fixes with actual fixes; we asked two additional questions to our pannel of experts: *Will you use MISFIRE in the future?* and *What aspects need to be improved?*.

Table IV reports the results of our pannel regarding accepted fix (✓), fix refused (x) and non-applicable or unsure (-). 41.6% of the proposed bug fixes have been unanimously accepted (1, 3, 6, 10 and 12) while 25% have been accepted by at least one member (4, 8, 11). We detail the refusals (x) and unsure (-) in the following.

$BF2$ was rejected by our pannel as the region of the commit that triggered a match is, in fact, generated code. While this generated code is pushed into the repositories and can be part of bug fixing commit, the root cause of the bug lies in the code generator itself. Our proposed fix was proposing to update the generated code. While a match in the generated code could lead the developer to the root cause (i.e. the code generator) the question we ask our reviewers was *"Is the proposed fix applicable in the given situation?"*. In this occurence, the proposed fix is not applicable.

$BF4$ was accepted by two reviewers and marked as unsure ($-$) by the rest of our pannel. Here, the reservation for the pannelists that were unsure came from the lack of context souroinding the proposed fix. Indeed, they were unable to determine if the fix was applicable or not without knowing what the original intent of the buggy commit was. In our reviewing session, we only provided the reviewers with the regions of the commits that match rather than the full commit. Full commits can be lengthy as they can containe assets descriptions and generated code in addition to the actual code. In this occurence, the full context of the commit might have helped our reviewers to decide if the $BF4$ was applicable or not. $BF5$ and $BF7$ were classified as unsure by the totallity of our pannel for the same reasons as $BF4$. $BF8$ was refused by four of our reviewers and rejected by two is at felt to a reviewer that the code clone match was more a refactoring opportunity than a fix. $BF12$ was marked as unsure by our pannel because the code was corresponding to a subsystem that is maintained by another team and our pannel judged itself unqualified.

For the two remaining questions we ask our reviewers *Will you use MISFIRE in the future?* and *What aspects need to be improved?*, they answered positively for the first question if some aspects are improved. First, the reviewers expressed concerns about the context sourounding the buggy commit and the fixes. While displaying the entire commits is not the solution, according to the pannel, some context mught be infered from the commit message (which were not provided) and from the JIRA issue associated with the fix. The second concern of our pannel, which generated many discussions, is the incapcity of CLEVERT to match generated code. In the occurence of a match in generated code, CLEVERT should be able to identify it and point towards the code generator rather than the generated code.

One interesting remark made by the pannel is that were expected buggy commit, and their associated proposed fix to be more complex (i.e. network, threading, physics) than the one we presented. Ultimately, they agreed that the bug showcased were rather simple and could have been catched during a code review by an experienced developer.

VI. Discussion

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

A. Limitations

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

CLEVERT is designed to work on multiple related systems. Applying CLEVERT on a single system will most likely be less effective as the the two steps classification 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 statistical models based on process and code metrics for the detection of risky commits such as the

ones developed by Kamei et al. and Rosen et al. [1], [24]. 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 CLEVERT is designed to work with Ubisoft systems. Ubisoft uses C#, C, C++, Java and other internally developped languages. It is however common to have other languages used in an environment with many inter-related systems. We intend to extend CLEVERT to process commits from other languages as well.

B. Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. Howerver, the systems vary in terms of purpose, size, and history.

In addition, we see a threat to validity that stems from the fact that we only used closed-source systems. The results may not be generalizable to open-source systems.

The programs we used in this study are all based on the C#, C, C++ and Java programming language. This can limit the generalisation of the results to projects written in other languages.

Finally, part of the analysis of the CLEVERT proposed fixes that we did was based on manual comparisons of the CLEVERT fixes with those proposed by developers with a focus group composed of experienced engineers and software achitects. 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 (commit and changesets).

VII. Conclusion

In this paper, we presented CLEVERT (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. CLEVERT combines code metrics, clone detection techniques and project dependency analysis to detect risky commits within and across projects. CLEVERT operates at commit-time, i.e., before the commits reach the central code repository. Also, because it relies on code comparison, CLEVERT does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes CLEVERT a practical approach for preventing bugs and proposing corrective measures that integrate well with the developer's workflow through the commit mechanism.

VIII. Reproduction Package

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

IX. Acknowledgements

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

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the international*

conference on software engineering, 2005, pp. 580–586.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] S. Kim, T. Zimmermann, K. Pan, and E. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *Proceedings of the international conference on automated software engineering*, 2006, pp. 81–90.

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

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

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

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

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

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

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

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

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

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

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

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

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

[47] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the*

european conference on foundations of software engineering, 2011, pp. 311–231.

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

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

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

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

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

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

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