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

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

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

Emad Shihab
DAS Lab, CSE Dept, Concordia University
Montréal, QC, Canada
eshihab@cse.concordia.ca

I. Introduction

Research in software maintenance has evolved over the year to include areas like mining bug repositories, bug analytic, and bug prevention and reproduction. The ultimate goal is to develop better techniques and tools to./ help software developers detect, correct, and prevent bugs effectively and efficiently.

One particular (and growing) line of research focuses on the problem of preventing the introduction of bugs by detecting risky commits (preferably before the commits reach the central repository). Recent approaches (e.g., [1], [2]) rely on training models based on code and process metrics (e.g., code complexity, the experience of the developers, etc.) that are used to classify new commits as risky or not.

Despite the recent advances in the field, the literature shows that many existing software maintenance tools have yet to be adopted by industry [3]–[11]. After assessing existing commercially available tools and state of the art approaches proposed in the litterature with software architects, team leaders, and senior developers, we extracted factors that we believe are contributing to this situation:

- Integration with the developer's workflow: Most existing maintenance tools ([12]–[20] are some noticeable examples) are not integrated well into the work flow of software developers (i.e., coding, testing, debugging, committing). Using these tools, developers have to download, install and understand them to achieve a given task. They would constantly need to switch from one workspace to another for different tasks (i.e., feature location with a command line tool, development and testing code with an IDE, development and testing front end code with another IDE and a browser, etc.) [21]–[23].
- Corrective actions: The outcome of these tools does not always lead to corrective actions that the developers can implement. Most of these tools return several results that are often difficult to interpret by developers. Take, for example, FindBugs [10], a popular bug detection tool. This tool detects hundreds of bug signatures and reports them using an abbreviated code such as `CO_COMPARETO_INCORRECT_FLOATING`. Using this code, developers can browse the FindBug's dictionary and find the corresponding definition *This method compares double or float values using a pattern like this:* `val1 > val2 ? 1 : val1 < val2 ? -1 : 0`. While the detection of this bug pattern is accurate, the tool does not propose any corrective actions to the developers that can help them fix the problem. Moreover, it has been reported in the literature that the output of existing maintenance tools tends to be verbose at the point where developers decide to simply ignore them [24]–[28].
- Leverage of historical data: These tools do not leverage a large body of knowledge that already exists in open source or within the organisation. For defect prevention, for example, the state of the art approaches consists of adapting statistical models built for one project to another project [1], [2]. As argued by Lewis *et al.* [3] and Johnson *et al.* [8], approaches based solely on statistical models are perceived by developers as black box solutions. Developers are less likely to trust the output of these tools.

In this paper, we propose an approach that adresses these challenges whithin the framework of ultra-large repositories at Ubisoft.

Ubisoft, one of the world's largest video game development companies, specialises in the design and implementation of high-budget video games. Such high-budget video games involve thousands of highly qualified personels from various fields (developers, artists, management, marketing, ...). To give the reader a grasp over the actual size of such software projects; the final products can contain millions of files and hundreds of thousands of commits.
Furthermore, several high-budget games are under development

at the same time by 8,000 developers scatered accross the 29 locations on six continents.

Our approach named MISFIRE (MetrIcS and clone detection for Fault Interception and Removing in ultra-large rEpositories) leverages three decades of code history in a cross-projects manner with the aims to automatically detecte and propose faults correction before they reach the central software repository. More specifically, it uses a combination of well-known code metrics to build statistical models to prevent faults insertion. In addition to these metrics, we also use an in-house clone detector that extracts code blocks from incoming commits and compare them to those of known defect-introducing commits. Finally, using the fix used to fix past defects, we can provide a solution, in the form of a contextualised diff, to the developer that would remove the fault. A contextualised diff is a diff that is tempered with to match the current workspace of the developer regarding variable types and names.

This novel approach outperforms state-of-the-art approaches such as Commit Guru [29] when it comes to defect introduction detection. Indeed, we can detect defects introduction with a 79% precision and a 65% recall. The performances of Commit-Guru, on the same dataset, are 66% precision and 63% recall. Also we asked in-house experts to manually evaluate the quality of the fix we propose to remove the faults from the current code submission. More than XX% of the analysed proposed fixes were judged highly relevant, and another XX% were classified as relevant. Overall, XX% of the proposed fixes help developers toward the craft of an actual fix. Finally, MISFIRE addresses the three factors we identified as contributing to the slow adoption of automatic defect prevention tools in large companies (Integration with the developer's workflow, Corrective actions and Leverage of historical data).

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 misfire 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 [30] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [31]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [32].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [33], El Emam *et al.* [34], Subramanyam *et al.* [35] and Gyimothy *et al.* [36] 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.* [18], [19], Demange *et al.* [37] and Palma *et al.* [38] 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.* [39], [40] and Zimmerman *et al.* [41], [42] 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 [43] 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., [44], [45]). 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 [44]. They find that locations that have been recently modified and fixed locations are the most defect-prone. Similarly, Ostrand *et al.* [45] 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.* [46] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [44]. Rahman and Devanbu found that, in general, process-based metrics perform as good as or better than code-based metrics [47].

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 [48]. Hassan [49] 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 [50]. 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, misfire takes a different approach in that it leverages dependencies of a project to determine risky changes.

B. Transfert Defect Learning

Mathieu: To do after we complete V.C

C. Automatic Patch Generation

Since misfire 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.*

[51] 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.* [52] generated patches from human-written patches and showed that their tool, PAR, successfully generated patches for 27 of 119 bugs. Tao *et al.* [53] 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. [54], [55], and determine what bugs are best fit for automatic patch generation [56].

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 MISFIRE Approach

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

In the first process (Figures 1 and 2), misfire manages events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes. For simplicity, 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. In the second phase, misfire analyses the developer's new commits before they reach the central repository to detect potential risky commits (commits that may introduce bugs). Also, MISFIRE proposes potential fixes mined from the all the repositories within the organisation. Moreover, the fixes are transformed to match the actual workspace of the developer using adpating variable names, indentation and overall code structure.

The project tracking component of misfire listens to bug (or issue) closing events of major open-source projects (currently, misfire is tested with 12 large projects within Ubisoft). These projects share many dependencies that are both internal and external. We perform project dependency analysis to identify groups of highly-coupled projects with the aim to be able to transfert defect learning from one project to another similar project within the organisation. Transfering defects learning from projects to projects is a challenging task [2], and, as shown in our experiments (Section IV) this adhoc metric gives us statisfactory results.

In the second process (Figure 3), MISFIRE intercepts incoming commit before they leave developers' workstation using a pre-commit hook. A pre-commit hook is a script that executes itself at commit-time. Pre-commit hooks are custom scripts set to fire off when certain important actions of the versioning process occur. There are two groups 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 all sorts of reasons such as checking compliance with coding rules or automatic runs of unit test suites. The pre-commit hook runs before the developer specifies a commit message. At Ubisoft, several actions are undertook at commit-time. For example, one has to refer which task or issue is being addressed by the commit at hand, specify which reviewers reviewed the commit at hand if it was done during a pair-programming session and so on. We seamlessly intergrate MISFIRE with this already existing process. First, we extract the block of code modifed by the developer and compute the metrics representing the commit at hand. The metrics are exhaustively described in section III-B. Using the resulting vectore (step 4), we classify the commit as *risky* or *non-risky*. A *risky* commit is a commit that is likely to introduce a defect while a *non-risky* commit is classified as sane. Note that false positives (sane commit classified as *risky*) and false negatives (commit introducing a defect classified as *non-risky*) are present in this step. In section IV, we report these results and explain how we favor precision (i.e. reduce the amount of false-positives) over recall (i.e. reduce the amount of false-negative) to create a sense of trust between developers and MISFIRE. We report on this particular aspect in section VI. If in step 4 (classification) the commit is classified as *non-risky*, then the process stops, and the commit is allowed to be transfered from the developer's workstation to the central repository. In the case of a *risky* classification, the process continues with further analyses. We first start by normalising and formatting blocks of modified and compares them to the known defect-introducing commits present in the currated cluster by using text-based type 2 and type 3 clone detection. The clusters are said to be currated because we collect usage statics when MISFIRE proposes a contextualised fix to the developer. If, the developer does not use the fix the pertinence score ($p-score$) of the fix is reduced and we only propose fixes that have a $p - score > \alpha$. α is a live metric that management teams can adjust at any time without the need to rebuild statistical models. In addition, α is automatically adjusted in a nightly fashion based on the usage statistics of the day before. Finally, α can be different across projects, teams and even individual developers. In step 7, we adapt the matchin fixes to the actual context of the developer by modifying indentation depth and variable name in an effort to reduce context switching [57]–[59]. Finally, depending on the similarity several processes can be engaged. For example, if the similarity between the commit at hand and known defect-introducing commit in the same project's cluster is $S > 80\%$ then, we perform a hard reject of the commit while suggesting how to improve it. If the similarity is over 50%, then we perform a soft-reject where the developers would have to seek an aditional review before being able to submit the commit to the central repository. Finally, if the similarity is over 30%, then we accept the commit and suggest potential improvements throught our contextualised fixes. As for α, these thresholds for S can be different by projects, teams and individuals and are manually and automatically adjustable.

In the upcoming subsections, we describe each step in detail.

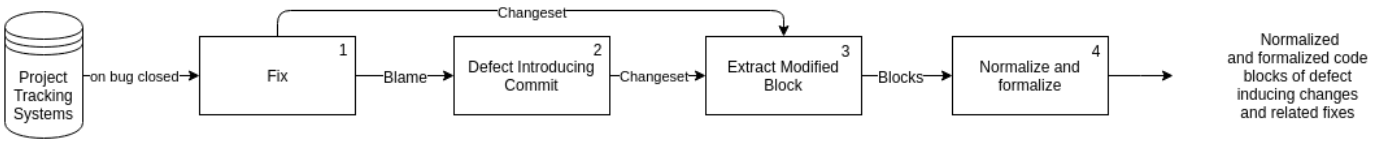

Fig. 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

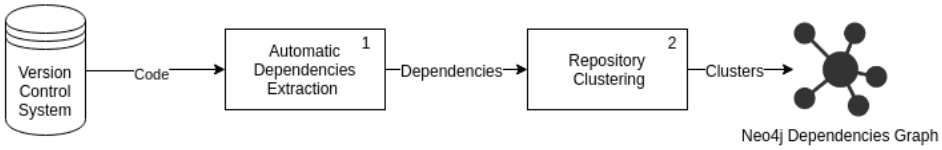

Fig. 2: Clustering by dependency

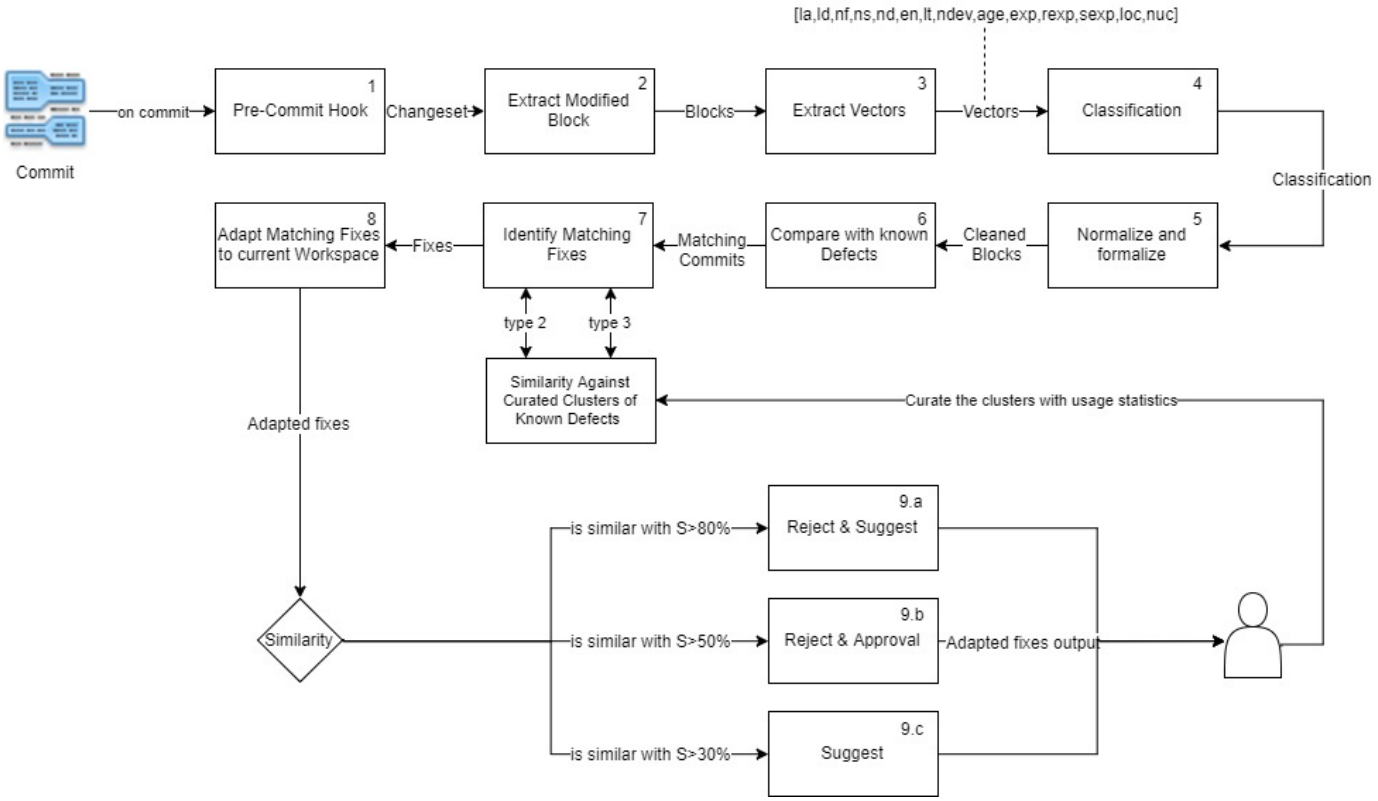

Fig. 3: Classifying incoming commits and proposing fixes

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. Graph databases use graph structures as a way to store and query information. In our case, a node corresponds to a project that is connected to other projects on which it depends. Dependencies can be *external* or *internal*. *external* dependencies refer to products that are not creater nor maintained within the organisation. These products can be open and closed source both. *Internal* dependencies refer to projects that are maintained in-house. While we cannot describe in details the *external* and *internal* dependencies used at Ubisoft for confidentiality and security reasons, we can offer the reader a graphical representation of the 12 analysed projects (yellow) with their dependencies (blue) 4 and the resulting clusters 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 [60], [61], used to detect communities by progressively removing edges from the

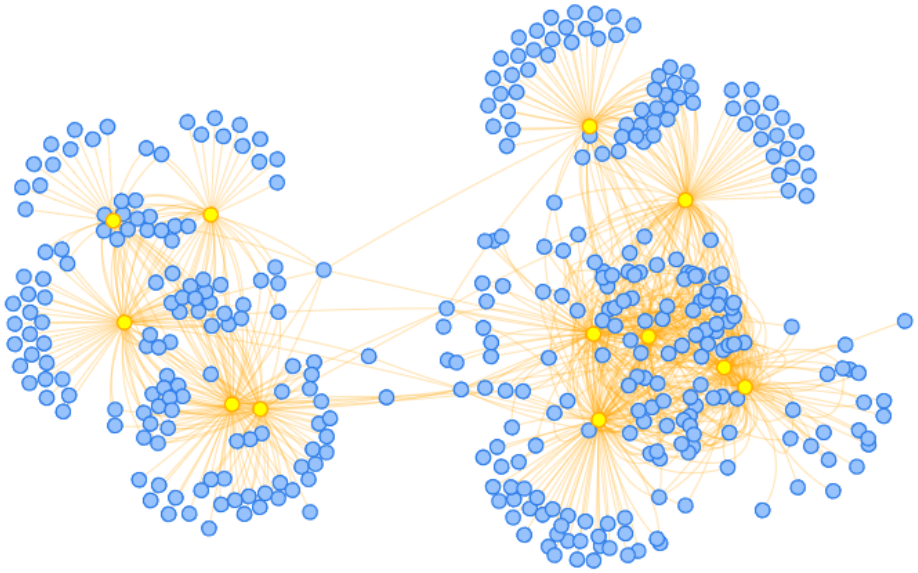

Fig. 4: Dependency Graph

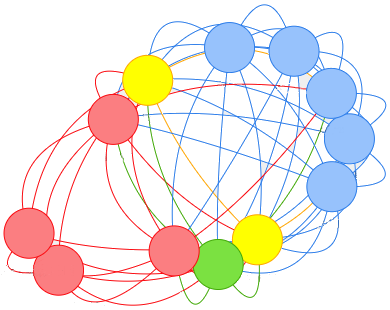

Fig. 5: Clusters

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 [61]. Other clustering algorithms can also be used.

The clusters are then used to devide the known defect introducing commits and their associated fixes. When in search for a solution to a *risky* commit we will only look for solutions applied to defect introducing commit in the same cluster. This allows to reduce the search space while enhancing the quality of the proposed solution as belonging to the same cluster is a mark of intrinsec similarity between projects regarding dependencies.

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: Misfire listens to bug (or issue) closing events happening on the project tracking system. Every time an issue is closed, misfire 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). Retrieving fix-commits, however, is known to be a challenging task [62]. This is because the link between the project tracking system and the code version control system is not always explicit. In an ideal situation, developers would add a reference to the issue they work on inside the description of the commit. However, this good practice is not always followed. To link fix-commits and their related issues we implemented the SZZ algorithm [63]. In addition to the SZZ algorithm, we build a statistical model using the following code change 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

Finally, a web api is able to receive new commits and provide

a way to access to the statistical model

It already exists an open-source implementation of such a system developed by Rosen *et al.* called Commit-Guru [29]. More specifically, we ported a Python versio developed by Rosen *et al.* [29] in GoLang for performances and internal maintenability reasons.

As Commit-guru's back-end, we have 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 entirely downloaded on a local server, each commit history is analysed. Unlike commit-guru that classifies commit using the list of keywords proposed by Hindle *et al.* [64], MISFIRE classifies the commit using the internal project tracking system. 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 the defect introducing commits (i.e., the defect-commits).

The SZZ algorithm, used by commit-guru and MISFIRE has been shown to be effective in detecting risky commits [29], [50]. Moreover, we are more accurate in identfying fixing commits than any approaches using a keyword classification [64] and we do not rely on linking tools to re-construct the relationship between the commit and an issue such as Relink [62] as 100% of the commits are linked to tasks by design.

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;
```

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 a parentheses algorithms [65].

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

One important note about this database is that the process can be cold-started. A tool supporting misfire does not need to *wait* for a project to have issues and fixes to be in effect. It can leverage the defect-commits and fix-commits of projects in the same cluster that already have a history. Therefore, misfire is applicable at the beginning of every project. The only requirement is to use a dependency manager.

C. Finding potential fixes for faults

Each time a developer makes a commit, misfire 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 [66]–[72] we had to build our own text-based clone detector for several reasons. First and foremost, the clone detector should have the ability; once a clone has been indentified, to trasnform 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. Indeed, we want non only to match clone pairs by abstracting them; we also want to transform one blocks into another. The contextualised fix that we proposed to the developers can be syntactically incorect 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 [73]. 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. misfire 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 [74]. 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 [70].

The extracted, pretty-printed, normal ised and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [75]. 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 misfire 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 misfire 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., [29], [76]).
A tool that supports misfire should have enough flexibility to allow developers to enable or disable the recommendations made by misfire. Furthermore, because misfire acts before the commit reach the central repository, it prevents unfortunate pulls of defects by other members of the organisation.

While we cannot provide actual defect introducing and bug fixing commits, we provide a fictional example using two version of the well known bubble sort algorithm in table II. In the fictional example, we assume that the bubble sort on the left column would be a fix of the bubble sort on the right column. First, both bubble sorts are pretty-printed. Then, the variables and types are abstracted using # signs. Note that we display complete abstraction, but we can also choose to abstract only the variables names and keep the variables types. Both abstractions are performed by our clone detector and each commit classified as *risky* are compared to both abstractions. Then, the parts that match in both abstraction are displayed in red. Here, we have 10/19 lines that match. Consequently, the similarity between the two code abstracted code blocks is 52.6%. Finally, in the third row, we display the original fix and the contextualised one. On the contextualised fix, the variables names and types have been modified so they match the code that the developer is currently trying to submit.

D. Classifying Incoming Commits

The classification of incoming commits within Misfire is two folds. First, the commit goes throught 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 processing here. 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 history. This serves three different goals. The first one is to improve the precision, in terms of false positives, of the approach. It is our opignion that, as an organisation, such tools must maintain an acceptable level of trust from its users. If a *risky* commit classifier emits too many false alarms (i.e. classifying sane commit as *risky*); then it risks loosing the trust of its users and being ignored altogether. The second goals behind this double validation is to be able to propose a solution to the *risky* commit for each *risky* classification. Indeed, one of the shortfall we identified of existing tool is their inability to provide adequate actions to fix the problem at hand. In our case, each alarm is accompagnied with a contextualised changeset that serves as a potential solution. Finally, this two-steps classification allow us to significantly reduce the time required, in average, to classify a commit. Indeed, if each commit were to be analyzed against all the known signature in terms of code clone similarity, then, it would take more than 268 seconds per commit using the cluster we describe in the next section while analysing a normal day worth of commit. While 268 seconds, in average, could be perceived as acceptable for a classification, it is our opinion that such system should be near-instanteneous to be used. With the two steps validation, it takes, on average, 3.75 seconds.

In short, the approach is designed to have the highest precision possible while maitaining an acceptable recall and provide hints towards the solution in a near-instanteneous manner.

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

To select the projects used to evaluate our approach, we followed one simple criterion. Each project must be a major project within the organisation (i.e. AAA games) and not a library. In addition, each project is based on the same game

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;
        }
    }
}
```

engine. Ubisoft does have many game engine for different kind of needs.

B. Project Dependency Analysis

Figure 4 shows the project dependency graph. As shown in Figure 4, these projects are very much interconnected. A review of each cluster shows that this partitioning divides projects in terms of high-level functionalities. For example, one cluster is related to a particular given family of video games while the other can be another family. 11 engineers have manually validated these clusters at Ubisoft, and they are accurate at grouping similar projects together.

C. Building a Database of Defect-Commits and Fix-Commits for Performances Evaluation

To build the database that we can use to assess the performance of Misfire, 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 [63]. Then, this baseline is used to compute the precision and recall of Misfire.

Each time Misfire 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 [34], [50], [77]–[79].

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 *misfire*. 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 *misfire* to run, the number of detected clone pairs, and the commits that match the current commit. As an example, let's assume 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 misfire and c_2 is proposed to the developer as a potential solution for the defect introduced in c_3.

E. Evaluation Measures

Similar to prior work focusing on risky commits (e.g., [48], [50]), 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 misfire
- FP: is the number of healthy commits that were classified by misfire as risky
- FN: is the number of defect introducing-commits that were not detected by misfire
- 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 [29], [80]–[82], 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 misfire in detecting risky commits using clone detection and project dependency analysis. The main research question addressed by this case study is: *Can we detect risky commits by combining metrucs 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 longest part of the experiment is to construct the baseline as each commit must be analysed with the SZZ algorithm. Once the basline 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 Misfire by comparing it to several other classifiers. We compare it to a Commit-guru [29] as it is a good indicator of what a statical model only approach can achieve in terms of performances and it is accuracy have been proven. Then, we compare Misfire with itself but without leveraging our clusters by dependencies. In other words, we applied the same approach again but without integrating the defects from other projects in the statistical model nor the clone detection. This allows us to validate the usefulness of our clusters and, consequently, validate the approach for large multi-projects ecosystems.

A. Performance of Misfire

This novel approach outperforms state-of-the-art approaches such as Commit Guru [29] when it comes to defect introduction detection. Indeed, we are able to detect defects introduction 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*. On the same dataset Commit-Guru 66.71% precision and 63.01% recall. While applying only the first step of classification (i.e. the stastical one) and not code clone comparison, then, Misire obtains 70.05% precision and 71.40% recall.

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 Misfire 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 [83]. Several approaches have already been proposed to conteract these problems, however, they all require to classify *similar* project by hand and then, maninipulate the feature space in order to adapt the model learnt from one project to the other [2]. With our approach, the *similarity* between projects is computed automatically and, results show that we do not actually need to maninipulate the feature space. Figures 6, 7 and 8 show the performances of the Misfire classification, in terms of ROC-curve and recall over precision curves, for

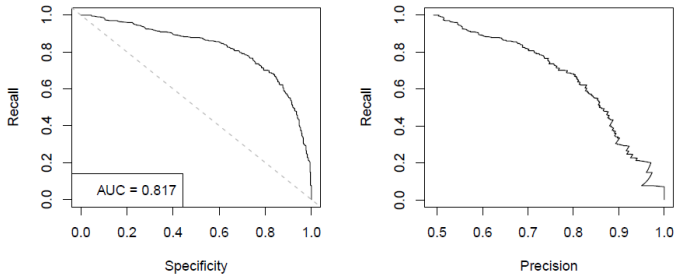

Fig. 6: Performances of Misfire while cold-starting the last project in the blue cluster

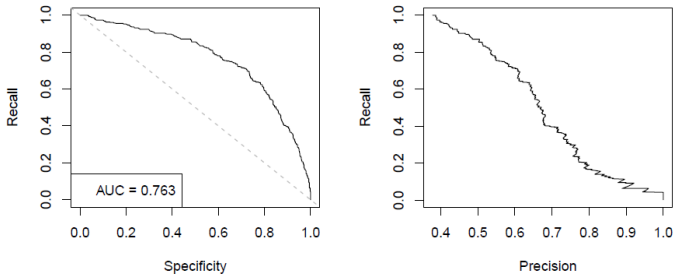

Fig. 7: Performances of Misfire while cold-starting the last project in the yellow cluster

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 efficient in identfying similar projects for defect learning transfert but also provide excellent performances 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 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.

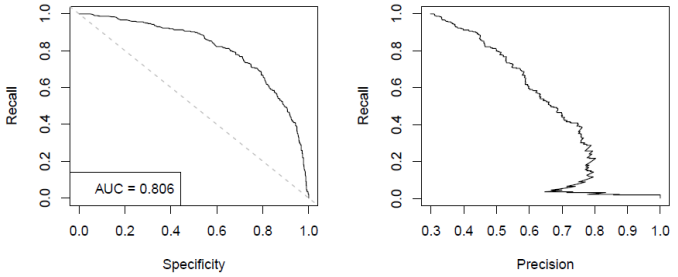

Fig. 8: Performances of Misfire while cold-starting the last project in the red cluster

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

Mathieu: This is plannified this week with architects at Ubisoft

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, misfire, which require further studies.

Misfire is designed to work on multiple related systems. Applying misfire 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. [29], [50]. 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 misfire 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 misfire 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 misfire proposed fixes that we did was based on manual comparisons of the misfire 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 misfire (MetrIcS and clone detection for Fault Interception and Removing in ultra-large rEpositories), 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. Misfire combines code metrics, clone detection techniques and project dependency analysis to detect risky commits within and across projects. Misfire operates at commit-time, i.e., before the commits reach the central code repository. Also, because it relies on code comparison, misfire does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes misfire 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 Misfire 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 Yves Jacquier, Olivier Pomarez, Nicolas Fleury, Alain Bedel, Mark Besner, David Punset, Paul Vlasie, Cyrille Gauclin, Luc Bouchard and Chadi Lebbos from Ubisoft for their participations in validating MISFIRE hypothesis, efficiency and the fixes proposed by our approach.

References

[1] D. Lo, "A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction," in *Proceedings of the european conference on software maintenance and reengineering*, 2013, pp. 331–334.

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

[3] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? Findings from a google case study," in *Proceedings of the international conference on software engineering*, 2013, pp. 372–381.

[4] S. L. Foss and G. C. Murphy, "Do developers respond to code stability warnings?" in *Proceedings of the 25th annual international conference on computer science and software engineering*, 2015, pp. 162–170.

[5] L. Layman, L. Williams, and R. S. Amant, "Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools," in *Proceedings of the first international symposium on empirical software engineering and measurement (eSEM 2007)*, 2007, pp. 176–185.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th aCM sIGPLAN-sIGSOFT workshop on program analysis for software tools and engineering - pASTE '07*, 2007, pp. 1–8.

[7] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 workshop on defects in large software systems - dEFECTS '08*, 2008, pp. 1–5.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the international conference on software engineering*, 2013, pp. 672–681.

[9] D. A. Norman, *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013, p. 347.

[10] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[11] N. Lopez and A. van der Hoek, "The code orb: supporting contextualized coding via at-a-glance views," in *Proceeding of the 33rd international conference on software engineering*, 2011, pp. 824–827.

[12] S. Kim, K. Pan, and E. E. J. Whitehead, "Memories of bug fixes," in *Proceedings of the 14th aCM sIGSOFT international symposium on foundations of software engineering - sIGSOFT '06/FSE-14*, 2006, p. 35.

[13] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[14] Findbugs, "FindBugs Bug Descriptions." 2015.

[15] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[16] F. Palma, M. Nayrolles, and N. Moha, "SOA Antipatterns : An Approach for their Specification and Detection," *International Journal of Cooperative Information Systems*, vol. 22, no. 04, pp. 1–40, 2013.

[17] M. Nayrolles, "Improving SOA Antipattern Detection in Service Based Systems by Mining Execution Traces," PhD thesis, 2013.

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

[19] 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.

[20] M. Nayrolles, E. Beaudry, N. Moha, and P. Valtchev, "Towards Quality-Driven SOA Systems Refactoring through Planning," in *6th international mCETECH conference*, 2015.

[21] T. J. Robertson, "Impact of interruption style on end-user debugging," in *Proceedings of the sIGCHI conference on human factors in computing systems*, 2004, pp. 287–294.

[22] T. J. Robertson, J. Lawrance, and M. Burnett, "Impact of high-intensity negotiated-style interruptions on end-user debugging,"

Journal of Visual Languages and Computing, vol. 17, no. 2, pp. 187–202, 2006.

[23] L. Beckwith, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the sIGCHI conference on human factors in computing systems*, 2006, pp. 231–240.

[24] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "A gamified tool for motivating developers to remove warnings of bug pattern tools," in *Proceedings - 2014 6th international workshop on empirical software engineering in practice, iWESEP 2014*, 2014, pp. 37–42.

[25] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proceedings - iCSE 2007 workshops: Fourth international workshop on mining software repositories, mSR 2007*, 2007.

[26] S. Kim and M. D. Ernst, "Which warnings should I fix first?" *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering ESECFSE 07*, p. 45, 2007.

[27] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *Proceedings of the 19th international symposium on software testing and analysis - iSSTA '10*, 2010, p. 241.

[28] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective Error Ranking for FindBugs," in *2011 fourth iEEE international conference on software testing, verification and validation*, 2011, pp. 299–308.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] I. Code and C. Search, *MuP-oWhaoongng Un Liveeersityl oof nScgi-eWncoena R ndoT hechSnoelougnyg(-PwOoSnTEH CwHa)ng HonSguK nongghuU nnivK erimsityo f.*

[58] Lange, D. B, and Y. Nakamura, "Object-Oriented Program Tracing and Visualization," *IEEE Computer*, vol. 30, no. 5, pp. 63–70, 1997.

[59] E. M. Altmann and J. G. Trafton, "Task Interruption: Resumption Lag and the Role of Cues," Aug. 2004.

[60] 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.

[61] 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.

[62] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the european conference on foundations of software engineering*, 2011, pp. 15–25.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

[70] 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.

[71] 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.

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

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.