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 in an effective and efficient manner.

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, 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 with 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 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 organization. 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, specializes 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 product 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 in order 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 are able to provide a solution, in the form of a contextualized diff, to the developer that would remove the fault. A contextualized diff is a diff that is tempered with to match the current workspace of the developer in terms of variable types and names.

This novel approach slightly outperforms state-of-the-art approaches such as Code Guru [29] when it comes to defect introduction detection. Indeed, we are able to detect defects introduction with a 71% precision and a 70% recall. In addition we ask in-house experts to manually evaluat the quality of the fix we propose to remove the faults from the current code submission. More than 62% of the analyzed proposed fixes were judged highly relevant and another 12% were classified as relevant. Overall, 74% 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 adotpion 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 organized 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 the work 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

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).

The project tracking component of misfire listens to bug (or issue) closing events of major open-source projects (currently, misfire is tested with 42 large projects). These projects share many dependencies. Projects can depend on each other or common external tools and libraries. We perform project dependency analysis to identify groups of highly-coupled projects.

In the second process (Figure 3), misfire identifies risky commits within each group to increase the chances of finding risky commits caused by project dependencies. For each project group, we extract code blocks from defect-commits and fix-commits.

The extracted code blocks are saved in a database that is used to identify risky commits before they reach the central repository. For each match between a risky commit and a *defect-commit*, we pull out from the database the corresponding *fix-commit* and present it to the developer as a potential way to improve the commit content. These phases are discussed in more detail in the upcoming subsections.

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. Project dependencies can be automatically retrieved if projects use a dependency manager such as Maven.

Figure 4 shows a simplified view of a dependency graph for a project named `com.badlogicgames.gdx`. As we can see, `com.badlogicgames.gdx` depends on projects owned by the same organization (i.e., badlogicgames) and other organizations such as Google, Apple, and Github.

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 [57], [58], 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 [58]. 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: 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 [59]. 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 make the link between fix-commits and their related issues, we turn to a modified version of the back-end of commit-guru [29]. Commit-guru is a tool, developed by Rosen *et al.* [29] to detect *risky commits*. In order to identify risky commits, Commit-guru builds a statistical model using change metrics (i.e., amount of lines added, amount of lines deleted, amount of files modified, etc.) from past commits known to have introduced defects in the past.

Commit-guru's back-end has three major components: ingestion, analysis, and prediction. We reuse the ingestion and the analysis part for misfire. 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. Commits are classified using the list of keywords proposed by Hindle *et al.* [60]. Commit-guru implements the SZZ algorithm [61] to detect risky changes, where it performs 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). Prior work showed that commit-guru is effective in identifying defect-

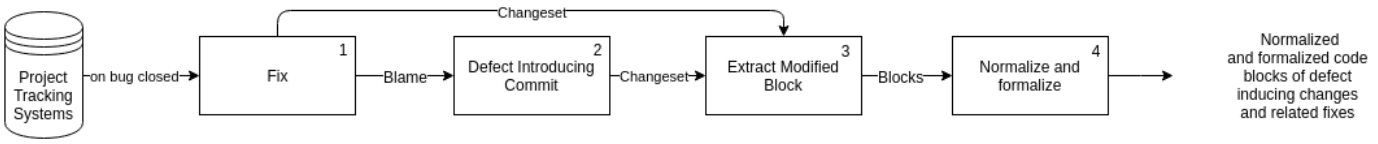

Fig. 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

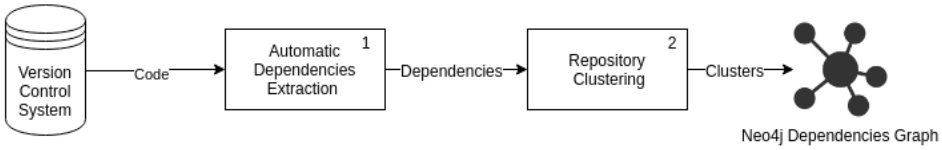

Fig. 2: Clustering by dependency

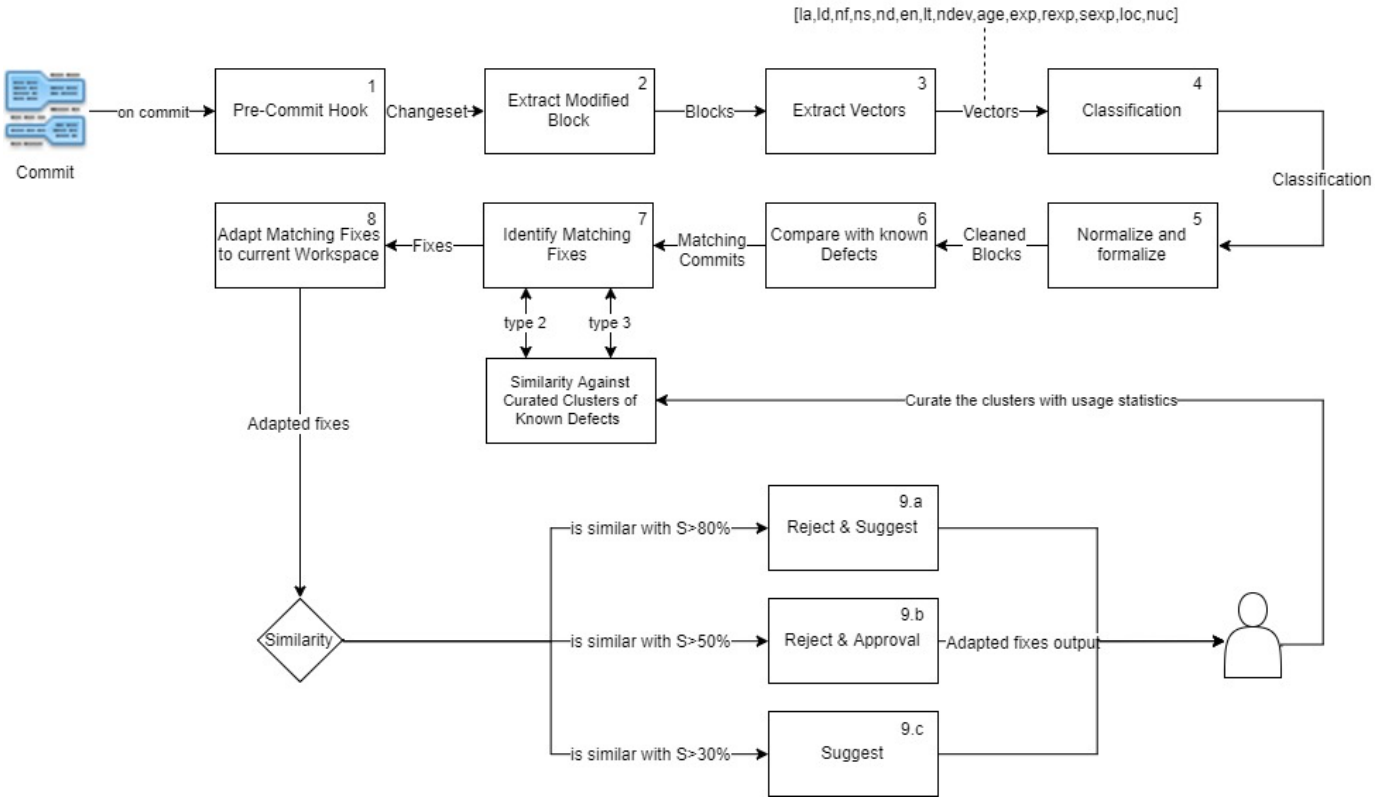

Fig. 3: Classifying incoming commits and proposing fixes

commits and their corresponding fixing commits [62] and the SZZ algorithm, used by commit-guru, is shown to be effective in detecting risky commits [29], [50]. Note that we could use a simpler and more established approach such as Relink [59] to link the commits to their issues and re-implement the classification proposed by Hindle *et al.* [60] on top of it. However, commit-guru has the advantage of being open-source, making it possible to modify it to fit our needs by fine-tuning its performance.

Extracting Code Blocks: To extract code blocks from fix-commits and defect-commits, we rely on TXL [63], which is a first-order functional programming over linear term rewriting, developed by Cordy et al. [63]. For TXL to work, one has to write a grammar describing the syntax of the source language and the transformations needed. TXL has three main phases: *parse*, *transform*, *unparse*. In the parse phase, the grammar controls not only the input but also the output forms. The following code sample—extracted from the official documentation—shows a grammar matching an *if-then-else* statement in C with some special keywords: [IN] (indent), [EX] (exdent) and [NL] (newline) that will be used in the output form.

```
define if_statement
  if ( [expr] ) [IN][NL]
[statement] [EX]
[opt else_statement]
```

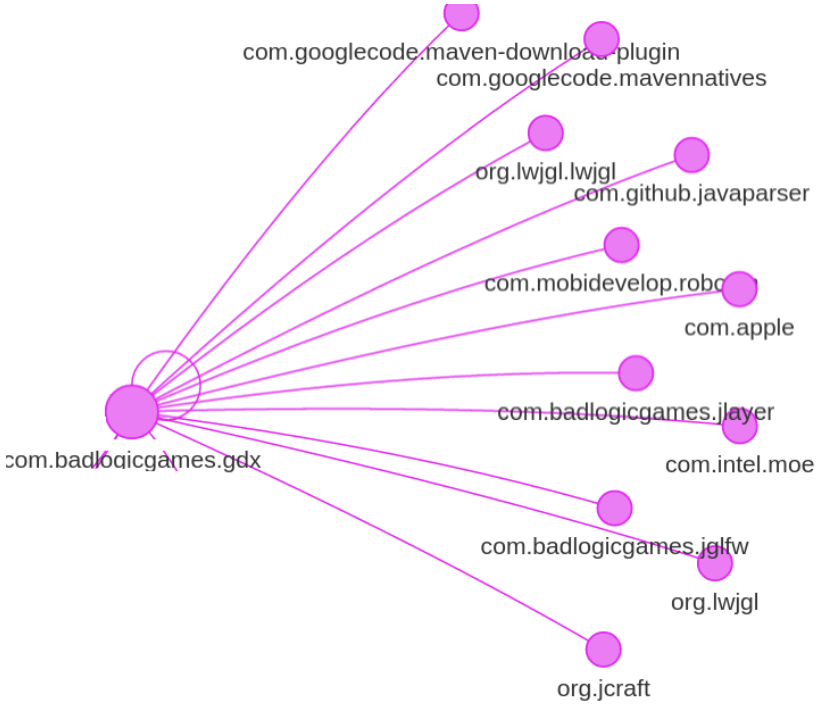

Fig. 4: Simplified Dependency Graph for `com.badlogicgames.gdx`

```
end define

define else_statement
  else [IN][NL]
[statement] [EX]
end define
```

Then, the *transform* phase applies transformation rules that can, for example, normalize or abstract the source code. Finally, the third phase of TXL, called *unparse*, unparses the transformed parsed input to output it. Also, TXL supports what its creators call *Agile Parsing* [64], which allow developers to redefine the rules of the grammar and, therefore, apply different rules than the original ones. misfire takes advantage of that by redefining the blocks that should be extracted for the purpose of code comparison, leaving out the blocks that are out of scope. More precisely, before each commit, we only extract the blocks belonging to the modified parts of the source code. Hence, we only process, in an incremental manner, the latest modification of the source code instead of the source code as a whole.

We have selected TXL for several reasons. First, TXL is easy to install and to integrate with the normal workflow of a developer. Second, it was relatively easy to create a grammar that accepts commits as input. This is because TXL supports C, Java, Csharp, Python and WSDL grammars, with the ability to customize them to accept changesets (chunks of the modified source code that include the added, modified, and deleted lines) instead of the whole code.

Algorithm 1 presents an overview of the *extract* and *save* blocks operations of misfire. This algorithm receives as argument, the changesets and the blocks that have been previously

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

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 below, 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]. Then, we send these expanded changesets to TXL for block extraction and formalization.

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. Analysing New Commits Using Pre-Commit Hooks

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 α is used to assess the extent beyond which two commits are considered similar. The setting of α is discussed in the case study section.

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. It is used to inspect the modifications that are about to be committed. misfire is based on a set of bash and python scripts, and the entry point of these scripts lies in a pre-commit hook. These scripts intercept the commit and extract the corresponding code blocks.

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]–[71], we selected NICAD as the main text-based method for comparing code blocks [72] for several reasons. First, NICAD is built on top of TXL, which we also used in the previous phase. Second, NICAD 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. The problem with the current implementation of NICAD is that it only considers complete Java, C#, and C files. We needed to adapt NICAD to process changesets. With the aid of TXL designers (through their online forum), we developed a TXL grammar for changesets.

NICAD works in three phases: *Extraction*, *Comparison* and *Reporting*. During the *Extraction* phase all potential clones are identified, pretty-printed, and extracted. We do not use the *Extraction* phase of NICAD as it has been built to work on programs that are syntactically correct, which is not the case for changesets. We replaced NICAD's *Extraction* phase with our scripts for building code blocks (described in the previous phase).

In the *Comparison* phase, the extracted blocks are transformed, clustered and compared to find potential clones. Using TXL sub-programs, blocks go through a process called pretty-printing where they are stripped of formatting and comments. When code fragments are cloned, some comments, indentation or spacing are changed according to the new context where the new code is used. This pretty-printing process ensures that all code will have the same spacing and formatting, which renders the comparison of code fragments easier. Furthermore, in the pretty-printing process, statements can be broken down into several lines. Table I [74] shows how this can improve the accuracy of clone detection with three `for` statements:

$$for(i = 0; i < 10; i++) \tag{1}$$

$$for(i = 1; i < 10; i++) \tag{2}$$

$$for(j = 2; j < 100; j++) \tag{3}$$

The pretty-printing allows NICAD to detect Segments 1 and 2 as a clone pair because only the initialization of i changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc [70]. In addition to the pretty-printing, code can be normalized and filtered to detect different classes of clones and match user preferences.

The extracted, pretty-printed, normalized 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

TABLE I: Pretty-Printing Example

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

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], [62]).
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 organization.

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 three simple criteria. First, the projects need to be in Java and use Maven to manage dependencies. This way, we can automatically extract the dependencies and perform the clustering of projects. The second criterion is to have projects that enjoy a large community support and interest. We selected projects that have at least 2000 followers. Finally, the projects must have a public issue repository to be able to mine their past issues and the fixes. We queried Github with these criteria and retrieved 42 projects (see Table III for the list of projects), including those from some of major open-source contributors such as Alibaba, Apache Software Foundation, Eclipse, Facebook, Google and Square.

B. Project Dependency Analysis

Figure 5 shows the project dependency graph. The dependency graph is composed of 592 nodes divided into five clusters shown in yellow, red, green, purple and blue. The size of the nodes in Figure 5 is proportional to the number of connections from and to the other nodes.

As shown in Figure 5, these Github projects are very much interconnected. On average, the projects composing our dataset have 77 dependencies. Among the 77 dependencies, on average, 62 dependencies are shared with at least one other project from our dataset.

Table II shows the result of the Girvan–Newman clustering algorithm in terms of centroids and betweenness. The blue cluster is dominated by Storm from The Apache Software Foundation. Storm is a distributed real-time computation system. Druid by Alibaba, the e-commerce company that provides consumer-to-consumer, business-to-consumer and business-to-business sales services via web portals, dominates the yellow cluster. In recent years, Alibaba has become an active member of the open-source community by making some of its projects publicly available. The red cluster has Hadoop by the Apache Software Foundation as its centroid. Hadoop is an open-source software framework for distributed storage and distributed processing of very large datasets on computer clusters built from commodity hardware. The green cluster is dominated by the Persistence project of OpenHab. OpenHab proposes home automation solutions and the Persistence project is their data access layer. Finally, the purple cluster is dominated by Libdx by Badlogicgames, which is a cross-platform framework for game development.

A review of each cluster shows that this partitioning divides projects in terms of high-level functionalities. For example, the blue cluster is almost entirely composed of projects from the Apache Software Foundation. Projects from the Apache Software Foundation tend to build on top of one another. We also have the red cluster for Hadoop, which is by itself an ecosystem inside the Apache Software Foundation. Finally, we obtained a cluster for e-commerce applications (yellow), real-time network application for home automation (green), and game development (purple).

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 used Commit-guru to retrieve the complete history of each project and label commits as defect-commits if they appear to be linked to a closed issue. The process used by Commit-guru to identify commits that introduce a defect is simple and reliable in terms of accuracy and computation time [50]. We use the commit-guru labels as the baseline 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], [76]–[78].

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

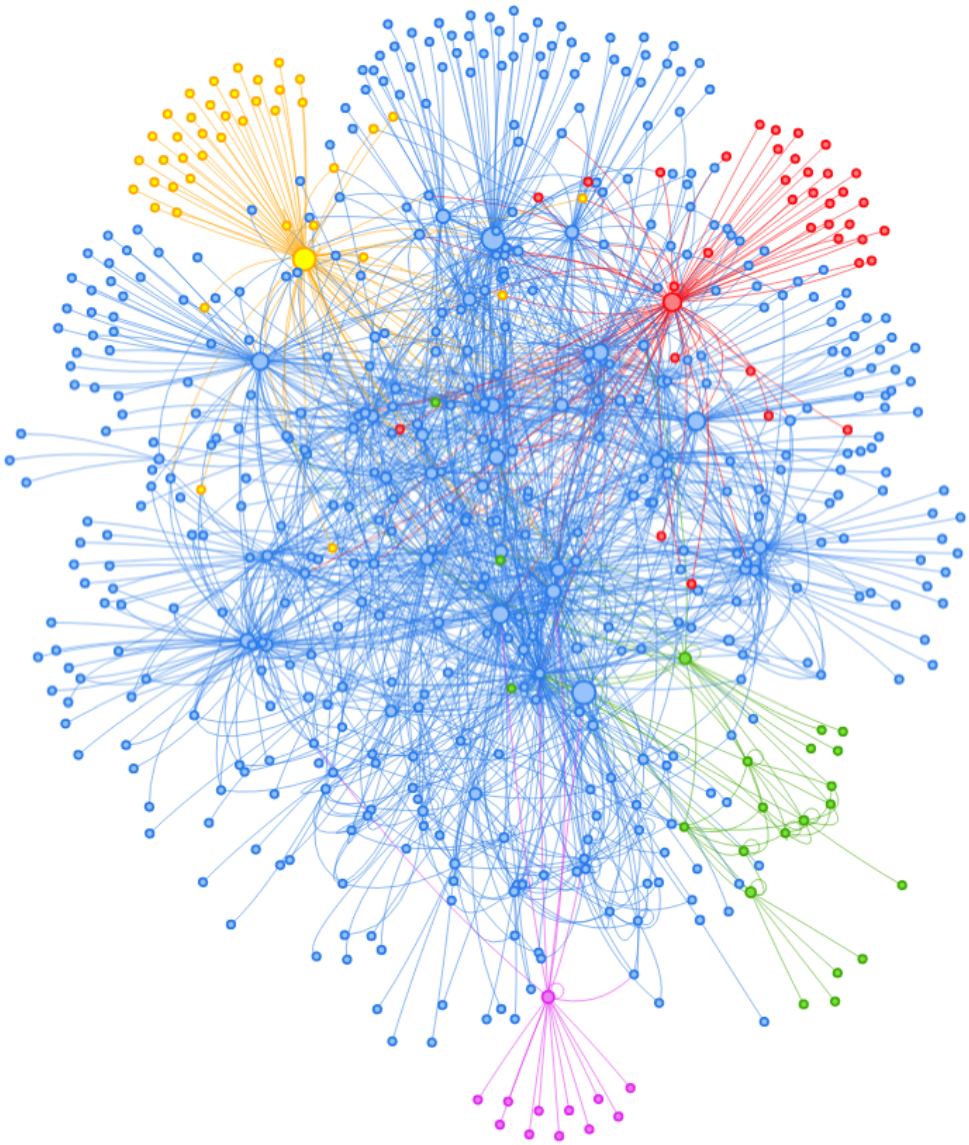

Fig. 5: Dependency Graph

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 an 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.

To measure the similarity between pairs of commits, we need to decide on the value of α. One possibility would be to test for all possible values of α and pick the one that provides best accuracy (F_1-measure). The ROC (Receiver Operating Characteristic) curve can then be used to display the performance of misfire with different values of α. Running experiments with all possible α turned out to be computationally demanding given the large number of commits. Testing with all the different values of α amounts to 4e10 comparisons.

To address this, we randomly selected a sample of 1700 commits from our dataset and checked the results by varying α from 1 to 100%. Figure 6 shows the results. The best trade-off between precision and recall is obtained when α = 35%. This threshold is in line with the findings of Roy et al. [72], [79] who showed through empirical studies that using NICAD with a threshold of around 30%, the default setting, provides good results for the detection of Type 3 clones. For these reasons, we set α = 35% in our experiments.

TABLE II: Communities in terms of ID, Color code, Centroids, Betweenness and number of members

#ID	Community	Centroids	Betweenness	# Members
1	Blue	Storm	24,525	479
2	Yellow	Alibaba	24,400	42
3	Red	Hadoop	16,709	37
4	Green	Openhab	3,504	22
5	Purple	Libdx	6,839	12

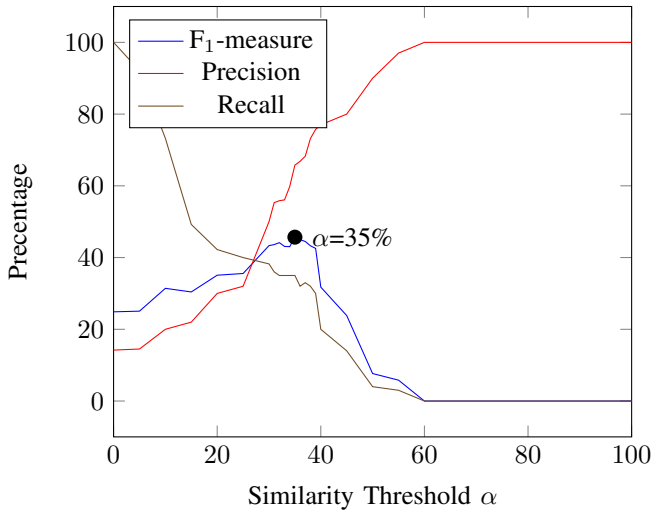

Fig. 6: Precision, Recall and F_1-measure variations according to α

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 using code comparison within and across related projects, and if so, what would be the accuracy?*

Table III shows the results of applying misfire in terms of the organization, project name, a short description of the project, the number of classes, the number of commits, the number of defect-commits, the number of defect-commits detected by misfire, precision (%), recall (%), F_1-measure and the average difference, in days, between detected commit and the *original* commit inserting the defect for the first time.

With α = 35%, misfire achieves, on average, a precision of 90.75% (13,899/15,316) commits identified as risky. These commits triggered the opening of an issue and had to be fixed later on. On the other hand, misfire achieves, on average, 37.15% recall (15,316/41,225), and an average F_1 measure of 52.72%.

TABLE III: BIANCA results in terms of organization, project name, a short description, number of class, number of commits, number of defect introducing commits, number of risky commit detected, precision (%), recall (%), F_1-measure (%), the average similarity of first 3 and 5 proposed fixes with the actual fix and the average time difference between detected and original.

Organization	Project Name	Short Description	NoC	#Commits	Bug Introducing Commit	Detected	Precision	Recall	F_1	Top 5 Fixes Similarity	Top 3 Fixes Similarity
Alibaba	druid	Database connection pool	3,309	4,775	1,260	787	88.44	62.46	73.21	39.97	46.69
	dubbo	RPC framework	1,715	1,836	119	61	96.72	51.26	67.01	60.01	57.14
	fastjson	JSON parser/generator	2,002	1,749	516	373	95.71	72.29	82.37	18.19	15.23
	jstorm	Stream Process	1,492	215	24	21	90.48	87.50	88.96	22.38	30.48
Apache	hadoop	Distributed processing	9,108	14,154	3,678	851	86.84	23.14	36.54	38.94	47.68
	storm	Realtime system	2,209	7,208	951	444	86.26	46.69	60.58	53.03	61.10
Clojure	clojure	Programming language	335	2,996	596	46	86.96	7.72	14.18	53.61	59.52
Dropwizard	dropwizard	RESTful web services	964	3,809	581	179	96.65	30.81	46.72	47.54	53.56
	metrics	JVM metrics	335	1,948	331	129	95.35	38.97	55.33	22.53	31.82
Eclipse	che	Eclipse IDE	7,818	1,826	169	9	88.89	5.33	10.05	31.01	39.04
Excilys	Android Annotations	Android Development	1,059	2,582	566	9	100.00	1.59	3.13	25.60	32.13
Facebook	fresco	Images Management	1,007	744	100	68	92.65	68.00	78.43	64.14	71.03
Gocd	gocd	Continuous Delivery server	16,735	3,875	499	297	91.58	59.52	72.15	21.62	30.59
Google	auto	source code generators	257	668	124	95	100.00	76.61	86.76	47.66	55.70
	guava	Google Libraries for Java 6+	1,731	3,581	973	592	98.48	60.84	75.22	23.74	23.59
	guice	Dependency injection	716	1,514	605	104	85.58	17.19	28.63	34.77	34.53
	iosched	Android App	1,088	129	9	6	100.00	66.67	80.00	16.50	24.97
Gradle	gradle	Build system	11,876	37,207	6,896	1,557	97.50	22.58	36.67	23.58	19.93
Jankotek	mapdb	Concurrent datastructures	267	1,913	691	440	94.32	63.68	76.03	63.16	72.48
Jhy	jsoup	Parser	136	917	254	153	87.58	60.24	71.38	46.41	44.59
Libdx	libgdx	Java game development	4,679	12,497	3,514	1,366	87.70	38.87	53.87	57.70	56.31
Netty	netty	Event-driven application	2,383	7,580	3,991	1,618	89.43	40.54	55.79	63.41	62.67
Openhab	openhab	Home Automation Bus	5,817	8,826	28	2	100.00	7.14	13.33	28.46	30.66
Openzipkin	zipkin	Distributed tracing system	397	799	176	73	87.67	41.48	56.31	55.92	51.90
Orfjackal	retrolambda	Backport of Java 8's lambda	171	447	97	35	94.29	36.08	52.19	34.69	42.06
OrientTechnologie	orientdb	Multi-Model DBMS	2,907	13,907	7,441	2,894	86.77	38.89	53.71	62.20	70.00
Perwendel	spark	Sinatra for java	205	703	125	82	97.56	65.60	78.45	21.88	28.00
PrestoDb	presto	Distributed SQL query	4,381	8,065	2,112	991	90.62	46.92	61.83	23.34	20.64
RoboGuice	roboguice	Google Guice on Android	1,193	1,053	229	70	91.43	30.57	45.82	53.81	56.55
Lombok	lombok	Additions to the Java language	1,146	1,872	560	212	91.98	37.86	53.64	58.94	57.49
Scribejava	scribejava	OAuth library	218	609	72	16	93.75	22.22	35.93	30.05	38.16
Square	dagger	Dependency injector	232	697	144	84	90.48	58.33	70.93	64.29	64.97
	javapoet	Java API	66	650	163	113	100.00	69.33	81.88	51.04	53.20
	okhttp	HTTP+HTTP/2 client	344	2,649	592	474	93.04	80.07	86.07	29.09	24.91
	okio	I/O API for Java	90	433	40	24	100.00	60.00	75.00	31.51	35.50
	otto	Guava-based event bus	84	201	15	15	93.33	100.00	96.55	54.11	49.94
	retrofit	Type-safe HTTP client	202	1,349	151	111	99.10	73.51	84.41	49.88	45.46
StephaneNicolas	robospice	Android library	461	865	113	39	87.18	34.51	49.45	60.90	65.04
ThinkAurelius	titan	Graph Database	2,015	4,434	1,634	527	90.13	32.25	47.51	48.64	50.59
Xetorthio	jedis	Redis client	203	1,370	295	226	92.04	76.61	83.62	25.69	29.45
Yahoo	anthelion	Plugin for Apache Nutch	1,620	7	0	-	-	-	-	-	-
Zxing	zxing	1D/2D barcode image	3,030	3,253	791	123	94.31	15.55	26.70	29.35	37.96
Total			**96,003**	**165,912**	**41,225**	**15316**	**90.75**	**37.15**	**52.72**	**40.78**	**44.17**

The relatively *low* recall is to be expected, since misfire classifies commits as risky only if a similar defect-introducing commit happened in one of the 42 open-source projects.

Also, out of the 15,316 commits misfire classified as *risky*, only 1,320 (8.6%) were because they were matching a defect-commit inside the same project. This finding supports the idea that developers of a project are not likely to introduce the same defect twice while developers of different projects that share dependencies are, in fact, likely to introduce similar defects. We believe this is an important finding for researchers aiming to achieve cross-project defect prevention, regardless of the technique (e.g., statistical model, AST comparison, code comparison, etc.) employed.

It is important to note that we do not claim that 37.15% of issues in open-source systems are caused by project dependencies. To support such a claim, we would need to analyse the 15,316 detected defect-commits and determine how many yield defects that are similar across projects.

Studying the similarity of defects across projects is a complex task and may require analysing the defect reports manually. This is left as future work. That said, we showed, in this paper, that software systems sharing dependencies also share common issues, irrespective of whether these issues represent similar defects or not.

The experiments took nearly three months using 48 Amazon Virtual Private Servers running in parallel. When deployed, the most time consuming part of misfire was spent on building the model of known bug-introducing commits. Once the model was built, it took, on average, 72 seconds to analyze an incoming commit on a typical workstation (quad-core @ 3.2GHz with 8 GB of RAM).

In the following subsections, we compare misfire with a random classifier, analyze the best and worst performing projects and assess the quality of the proposed fixes.

A. Baseline Classifier Comparison

Although our average F_1 measure of 52.72% may seem low at first glance, achieving a high F_1 measure for unbalanced data is very difficult [83]. Therefore, a common approach to ground detection results is to compare it to a simple baseline.

To the best of our knowledge, this is the first approach that relies on code similarity instead of code or process metrics for the detection of risky commits. Comparing it to other approaches will not be accurate. In addition, existing metric-based techniques (e.g., [2]) detect risky commits within single projects only. misfire, on the other hand, operates across projects. We compared misfire with a random classifier to have a baseline and show that we perform better than a simple baseline.

The baseline classifier first generates a random number n between 0 and 1 for the 165,912 commits composing our dataset. For each commit, if n is greater than 0.5, then the commit is classified as risky and vice versa. As expected by a random classifier, our implementation detected ~50% (82,384 commits) of the commits to be *risky*. It is worth mentioning that the random classifier achieved 24.9% precision, 49.96% recall and 33.24% F_1-measure. Since our data is unbalanced (i.e., there are many more *healthy* than *risky* commits) these numbers are to be expected for a random classifier. Indeed, the recall is very close to 50% since a commit can take on one of two classifications, risky or non-risky. While analysing the precision, however, we can see that the data is unbalanced (a random classifier would achieve a precision of 50% on a balanced dataset).

It is important to note that the purpose of this analysis is not to say that we outperform a simple random classifier, rather to shed light on the fact that our dataset is unbalanced and achieving an average F_1 = 52.72% is non-trivial, especially when a baseline only achieves an F_1-measure of 33.24%.

B. Performance of misfire

In this section, we provide insight on the performance of misfire by examining the projects for which the best and worst results were obtained.

misfire performed best when applied to three projects: Otto by Square (100.00% precision and 76.61% recall, 96.55% F_1-measure), JStorm by Alibaba (90.48% precision, 87.50% recall, 88.96% F_1-measure), and Auto by Google (90.48% precision, 87.50% recall, 86.76% F_1-measure). It performed worst when applied to Android Annotations by Excilys (100.00% precision, 1.59% recall, 3.13% F_1-measure) and Che by Eclipse (88.89% precision, 5.33% recall, 10.05% F_1-measure), Openhab by Openhab (100.00% precision, 7.14% recall, 13.33% F_1-measure). To understand the performance of misfire, we conducted a manual analysis of the commits classified as *risky* by misfire for these projects.

1) Otto by Square (F_1-measure = 96.5%): At first, the F_1-measure of Otto by Square seems surprising given the specific set of features it provides. Otto provides a Guava-based event bus. While it does have dependencies that make it vulnerable to defects in related projects, the fact that it provides specific features makes it, at first sight, unlikely to share defects with other projects. Through our manual analysis, we found that out of the 16 *risky* commits detected by misfire, 11 (68.75%) matched defect-introducing commits inside the Otto project itself. This is significantly higher than the average number of single-project defects (8.6%). Further investigation of the project management system revealed that very few issues have been submitted for this project (15) and, out of the 11 matches inside the Otto project, 7 were aiming to fix the same issue that had been submitted and fixed several times instead of re-opening the original issue.

2) JStorm by Alibaba (F_1-measure = 88.96%): For JStorm by Alibaba, our manual analysis of the *risky* commits revealed that, in addition to providing stream processes, JStorm mainly supports JSON. The commits detected as *risky* were related to the JSON encoding/decoding functionalities of JStorm. In our dataset, we have several other projects that supports JSON encoding and decoding such as FastJSON by Alibaba, Hadoop by Apache, Dropwizard by Dropwizard, Gradle by Gradle and Anthelion by Yahoo. There is, however, only one project supporting JSON in the same cluster as JStorm, Fastjson by

Alibaba. FastJSON has a rather large history of defect-commits (516) and 18 out of the 21 commits marked as *risky* by misfire were marked so because they matched defect-commits in the FastJSON project.

3) Auto by Google (F_1-measure = 86.76%): Google Auto is a code generation engine. This code generation engine is used by other Google projects in our database, such as Guava and Guice. Most of the Google Auto *risky* commits (79%) matched commits in the Guava and the Guice project. As Guice and Guava share the same code-generation engine (Auto), it makes sense that code introducing defects in these projects share the characteristics of commits introducing defects in Auto.

4) Openhab by Openhab (F_1-measure = 13.33%): Openhab by Openhab provides bus for home automation or smart homes. This is a very specific set of feature. Moreover, Openhab and its dependencies are alone in the green cluster. In other words, the only project against which misfire could have checked for matching defects is Openhab itself. misfire was able to detect 2/28 bugs for Openhab. We believe that if we had other home-automation projects in our dataset (such as *HomeAutomation* a component based for smart home systems [84]) then we would have achieved a better F_1-measure.

5) Che by Eclipse (F_1-measure = 10.05%): Eclipse Che is part of the Eclipse IDE. Eclipse provides development support for a wide range of programming languages such as C, C++, Java and others. Despite the fact that the Che project has a decent amount of defect-commits (169) and that it is in the blue cluster (dominated by Apache) misfire was only able to detect nine *risky* commits. After manual analysis of the 169 defect-commits, we were not able to draw any conclusion on why we were not able to achieve better performance. We can only assume that Eclipse's developers are particularly careful about how they use their dependencies and the quality of their code in general. Only 2% (169/7,818) of their commits introduce new defects.

6) Annotations by Excilys (F_1-measure = 3.13%): The last project we analysed manually is Annotations by Excilys. Similar to Openhab by Openhab, it provides a very particular set of features, which consist of Java annotations for Android projects. We do not have any other project related to Java annotations or the Android ecosystem at large. This caused misfire to perform poorly.

Our interpretation of the manual analysis of the best and worst performing projects is that misfire performs best when applied to clusters that contain projects that are similar in terms of features, domain or intent. These projects tend to be interconnected through dependencies. In the future, we intend to study the correlation between the cluster betweenness measure and the performance of misfire.

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

One of the advantages of misfire over other techniques is that it also proposes fixes for the *risky* commits it detects. In order to evaluate the quality of the proposed fixes, we compare the proposed fixes with the actual fixes provided by the developers. To do so, we used the same preprocessing steps we applied to incoming commits: extract, pretty-print, normalize and filter the blocks modified by the proposed and actual fixes. Then, the blocks of the actual fixes and the proposed fixes can be compared with our clone comparison engine.

Similar to other studies recommending fixes, we assess the quality of the first three and five proposed fixes [51]–[56]. The average similarity of the first three fixes is 44.17% while the similarity of the first five fixes is 40.78%. Results are reported in Table III.

In the framework of this study, for a fix to be ranked as qualitative it has to reach α=35% similarity threshold. Meaning that the proposed fixed must be at least 35% similar to the actual fix. On average, the proposed fixes are above the α=35% threshold. On a per-commit basis, misfire proposed 101,462 fixes for the 13,899 true positives *risky commits* (7.3 per commit). Out of the 101,462 proposed fixes, 78.67% are above α=35% threshold.

In other words, misfire is able to detect *risky* commits with 90.75% precision, 37.15% recall, and proposes fixes that contain, on average, 40-44% of the actual code needed to transform the *risky* commit into a *non-risky* one.

To further assess the quality of the fixes proposed by misfire, we randomly took 250 misfire-proposed fixes and manually compared them with the actual fixes provided by the developers. For each fix, we looked at the proposed modifications (i.e., code diff) and the actual modification made by the developer of the system to fix the bug.

We were able to identify the statements from the proposed fixes that can be reused to create fixes similar to the ones that developers had proposed in 84% of the cases. For the remaining cases, it was difficult to understand the changes that the developers made, mainly because of our lack of familiarity with the systems under study. We recognize that a better evaluation of the quality of misfire-proposed fixes would be to conduct a user study. We intend to do this as part of future work. In what follows, we present examples of misfire-proposed fixes that were detected as similar to fixes proposed by developers.

In Figures 7 and 8, we show two commits that belong to the Okhttp and Druid systems, respectively. In these figures, the statements shown in red are the ones that triggered the match between the two commits. The Okhttp commit was submitted in February 2014, while the one from Druid was submitted in April 2016. The Druid commit was introduced to fix a bug, which was caused by a prior commit, submitted in March 2016. The bug consisted of invoking a function on a null reference, which led to a null pointer exception, causing the system to crash. This bug could have been avoided if the Druid developers had access to the Okhttp commit.

In a second example, we present a case where misfire could have been used to avoid inserting a bug related to race conditions in multi-threaded code.

In Figures 9 and 10, we show two commits that belong to the Netty and Okhttp systems, respectively. For Figure 9, we present an excerpt of the commit that triggered the match. The whole commit affected 44 files with 1,994 additions and 1,335 deletions. The Netty commit was submitted in June 2014 while

```
@@ -293,7 +293,9 @@ private Android(
             byte[] alpnResult = (byte[]) getAlpnSelectedProtocol.invoke(socket);
             if (alpnResult != null) return ByteString.of(alpnResult);
           }
-          return ByteString.of((byte[]) getNpnSelectedProtocol.invoke(socket));
+          byte[] npnResult = (byte[]) getNpnSelectedProtocol.invoke(socket);
+          if (npnResult == null) return null;
+          return ByteString.of(npnResult);
         } catch (InvocationTargetException e) {
           throw new RuntimeException(e);
         } catch (IllegalAccessException e) {
```

Fig. 7: okhttp commit #0ca4c82dd1032625831a5814ea2ddcf165029bdc

```
@@ -760,12 +760,14 @@ protected String aliasWrap(String name) {
         Map<String, String> aliasMap = getAliasMap();

         if (aliasMap != null) {
+            if (aliasMap.get(name) == null) {
+                return null;
+            }

-            if (aliasMap.containsKey(name)) {
+            if (aliasMap.containsKey(name)
+                    && aliasMap.get(name) != null) {
                 return aliasMap.get(name);
             }

             String name_lcase = name.toLowerCase();
-            if (name_lcase != name && aliasMap.containsKey(name_lcase)) {
+            if (name_lcase != name && aliasMap.containsKey(name_lcase)
+                    && aliasMap.get(name_lcase) != null) {
                 return aliasMap.get(name_lcase);
             }
```

Fig. 8: Druid commit #1091861bb15876131653191ae409a523aa8ec0c5

the one from OKHttp was submitted in January 2017. The bug consisted of resource leakage in a multi-threaded environment. The similarity between the two commits comes from the `try` and `catch` blocks associated with the used exceptions, more precisely, the fact of freeing resources in case a thread crashes with a `finally` block to follow the `try` and `catch` blocks. In the `try` block, the threads are launched and, in case an exception happens, the `catch` block is executed. However, if the developer closes the resources consumed by the thread at the end of the `try` block then, in the case of an exception, the resources would not be freed. Instead of duplicating the resource management code in the `try` and `catch` blocks, a good practice would be to have it in a `finally` block that always executes itself, regardless of whether an exception is thrown or not. In the commit presented by Figure 9, we can see that a large refactoring has been done in order to prevent crashed threads to keep using resources. This bug could have been avoided if the Okhttp developers had access to the Netty commit.

Another example is the one depicted in Figures 11 and 12, showing two commits that belong to the JSoup and Orientdb systems, respectively. The first commit was submitted in November 2013, while the Orientdb was submitted two years later in October 2015. The Orientdb commit was used to fix a bug introduced by a commit that was submitted earlier in October 2015. This bug would have been avoided if the developer had access to the JSoup commit, that is proposed by misfire as the closest match.

In these fixes, we can see that the developers are working

```
 +         try {
+             Object v = threadLocalMap.indexedVariable(variablesToRemoveIndex);
+             if (v != null && v != InternalThreadLocalMap.UNSET) {
+                 @SuppressWarnings("unchecked")
+                 Set<FastThreadLocal<?>> variablesToRemove = (Set<FastThreadLocal<?>>) v;
+                 FastThreadLocal<?>[] variablesToRemoveArray =
+                         variablesToRemove.toArray(new FastThreadLocal[variablesToRemove.size
+                 for (FastThreadLocal<?> tlv: variablesToRemoveArray) {
+                     tlv.remove(threadLocalMap);
+                 }
+             }
+         } catch (IOException e) {
+         } catch (InterruptedException e) {
+         } finally {
+             InternalThreadLocalMap.remove();
+         }
```

Fig. 9: netty commit #085a61a310187052e32b4a0e7ae9700dbe926848

```
@@ -682,16 +682,21 @@ private void handleWebSocketUpgrade(Socket socket,
     BufferedSource source, Buffer
       response.getWebSocketListener().onOpen(webSocket, fancyResponse);
       String name = "MockWebServer WebSocket " + request.getPath();
       webSocket.initReaderAndWriter(name, 0, streams);
-      webSocket.loopReader();
-
-      // Even if messages are no longer being read we need to wait
   for the connection close signal.
       try {
-        connectionClose.await();
-      } catch (InterruptedException ignored) {
-      }
+        webSocket.loopReader();

-      closeQuietly(sink);
-      closeQuietly(source);
+        // Even if messages are no longer being read we need to wait
  for the connection close signal.
+        try {
+          connectionClose.await();
+        } catch (InterruptedException ignored) {
+        }
+
+      } catch (IOException e) {
+        webSocket.failWebSocket(e, null);
+      } finally {
+        closeQuietly(sink);
+        closeQuietly(source);
+      }
     }
```

Fig. 10: okhttp commit #a96c3a8007d8e1a166f7aec423c7add1ea0e3522

with the `StringBuilder` class. According to the Java documentation, the `StringBuilder` class *provides an API compatible with StringBuffer, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.* Developers usually use the `StringBuilder` class to build strings using the `append` and `insert` methods. Using the `StringBuilder` class rather than plain string concatenation (i.e., using the + operator) is known to be a good Java practice as it improves performance.

In both cases, the code has been modified to avoid the appending of `null` string. In JSoup, it is done by the method `shouldCollapseAttribute`, which checks for empty values. In Orientdb, the same operation is performed by a simple null check on the string named `right`. Note that this kind of *bug* would not have been spotted by a static analysis tool such as PMD [85] because it is *legal* to pass a null string as a parameter of function expecting a string. In both cases, however, the developers were tasked to avoid the appending of null strings.

VI. Discussion

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

A. Limitations

We identified three main limitations of our approach, misfire, that require further studies.

misfire is designed to work on multiple related systems. Applying misfire on a single system will most likely be ineffective; 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 is related to scalability of the approach. Because misfire operates on multiple systems, we need to build a model that comprises all their commits, which is a time consuming process. It took nearly three months using 48 Amazon Virtual Private Servers running in parallel to build and test the model for our experiments.

The third limitation we identified has to do with the fact that misfire is designed to work with Java systems only. It is however common to have a multitude of programming 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. However, the systems analyzed by misfire were selected from Github based on their popularity and the ability to mine their past issues and to retrieve their dependencies. Any project that satisfies these criteria would be included in the analysis. Moreover, 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 open-source systems. The results may not be generalizable to industrial systems. We intend to undertake these studies in future work.

The programs we used in this study are all based on the Java programming language. This can limit the generalization of the results to projects written in other languages. However, similar to Java, one can write a TXL grammar for a new language then misfire can work since misfire relies on TXL.

Moreover, we use NICAD as the code comparison engine. The accuracy of NICAD affects the accuracy of misfire. This said, since NICAD has been tested on large systems, we are confident that it is a suitable engine for comparing code using TXL. Also, there is nothing that prevents from using other text-based code comparisons engines. Another threat related to the use of NICAD is the use of 35% as a similarity threshold. A different threshold may affect the results. We chose this threshold because it resulted in best trade-off between precision and recall when analysing a subset of the dataset. We also relied on previous studies, in which the creators of NICAD Roy et al. [72], [79] showed that a threshold of around 30% usually provides good accuracy.

Finally, part of the analysis of the misfire proposed fixes that we did was based on manual comparison of the misfire fixes with those proposed by developers. Although we exercised great care in analyzing 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 42 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 (Bug Insertion ANticipation by Clone Analysis at commit time), an approach that detects risky commits (i.e., a commit that is likely to introduce a bug) with an average of 90.75% precision and 37.15% recall. misfire uses 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 code central repository. In addition, 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

```
@@ -34,7 +35,11 @@ public Object jjtAccept(OrientSqlVisitor visitor, Object data) {
    public void toString(Map<Object, Object> params, StringBuilder builder) {
      expression.toString(params, builder);
      builder.append(" MATCHES ");
-     builder.append(right);
+     if(right!=null) {
+       builder.append(right);
+     }else{
+       rightParam.toString(params, builder);
+     }
    }
```

Fig. 11: OrientDB commit #444db817ee9404b17c1208df51781ce9cb6a2666

```
@@ -100,6 +111,15 @@   protected void html(StringBuilder accum, Document.OutputSettings out)
-         accum
-             .append(key)
-             .append("=\"")
-             .append(Entities.escape(value, out))
-             .append("\"");
+         accum.append(key);
+         if (!shouldCollapseAttribute(out)) {
+             accum.append("=\"");
+             Entities.escape(accum, value, out, true, false, false);
+             accum.append('"');
+         }
      }

      /**
protected boolean isDataAttribute() {
          return key.startsWith(Attributes.dataPrefix) && key.length()
            > Attributes.dataPrefix.length();
      }

+     /**
+      * Collapsible if it's a boolean attribute and value is empty or same as name
+      */
+     protected final boolean shouldCollapseAttribute(Document.OutputSettings out) {
+         return ("".equals(value) || value.equalsIgnoreCase(key))
+                 && out.syntax() || Document.OutputSettings.Syntax.html
+                 && Arrays.binarySearch(booleanAttributes, key) >= 0;
+     }
+
```

Fig. 12: Jsoup commit #6c4f16f233cdfd7aedef33374609e9aa4ede255c

integrates well with the developers workflow through the commit mechanism.

To build on this work, we need to conduct a human study with developers in order to gather their feedback on the approach. The feedback obtained will help us fine-tune the approach. Also, we want to examine the relationship between project cluster measures (such as betweenness) and the performance of misfire. Finally, another improvement to misfire would be to support Type 4 clones.

VIII. Reproduction Package & Dataset

As described in Section IV-D, we rely heavily on virtual machines instrumentation and coordination to run our experiments. Providing a straightforward reproduction package is therefore very challenging. However, we are happy to share our consolidated dataset: https://github.com/MathieuNls/misfire-data. The dataset is composed of three compressed PostgresSQL formatted tables: clones, commits and repository. The clone table stores the relationship between set of similar commits. The commits themselves are in the commit table with details about their author, repository, commit message and all the metrics found in commit guru [29]. Finally, the repository table describes the repository used in terms of url, name and ingestion status.

IX. Acknowledgements

We are thankful to Yves Jacquier, Olivier Pomarez, Nicolas Fleury, Alain Bedel, from Ubisoft for their participations in validating MISFIRE and particularly 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, "Specification and Detection of SOA Antipatterns," *Procedings of the International Conference on Service Oriented Computing*, pp. 1–16, 2012.

[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] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] J. R. Cordy, "Source transformation, analysis and generation in TXL," in *Proceedings of the symposium on partial evaluation and semantics-based program manipulation*, 2006, pp. 1–11.

[64] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile Parsing in TXL," in *Proceedings of the international conference on automated software engineering*, 2003, pp. 311–336.

[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] 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.

[77] 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.

[78] 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.

[79] C. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *2008 16th iEEE international conference on program comprehension*, 2008, pp. 172–181.

[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] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to‘ Comments on'Data Mining Static Code Attributes to Learn Defect Predictors"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, p. 637, 2007.

[84] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, May 2012.

[85] A. Dangel, "PMD." 2000.