# BIANCA: Preventing Bug Insertion at Commit-Time Using Dependency Analysis and Clone Detection

Mathieu Nayrolles, Abdelwahab Hamou-Lhadj SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
{mathieu.nayrolles, wahab.hamou-lhadj}@concordia.ca

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

**Abstract**—Preventing the introduction of software defects at commit-time is a growing line of research in the software maintenance community. Existing approaches leverage code and process metrics to build statistical models that can effectively prevent defect insertion and propose fixes in a software project. Metrics, however, may vary from one project to another, hindering the reuse of these models. Moreover, these techniques operate within single projects only despite the fact that many projects share dependencies and are, therefore, vulnerable to similar faults. In this paper, we propose a novel approach, called BIANCA, that relies on clone detection and dependency analysis to detect *risky* commits within and across related projects. When applied to 42 projects, BIANCA achieves an average precision, recall and F-measure of 90.75%, 37.15% and 52.72%, respectively. We also found that only 8.6% of the risky commits detected by BIANCA match other commits from the same project, suggesting that relationships across projects need to be considered for effective prevention of risky commits. In addition, BIANCA is able to propose qualitative fixes to transform *risky* commits into *non-risky* ones in 78.67% of the cases.

**Keywords**—Bug Prediction; Risky Software Commits; Clone Detection; Software Maintenance

## 1 Introduction

Research in software maintenance continues to evolve to include areas like mining bug repositories, bug analytic, and bug prevention and reproduction. The ultimate goal is to develop 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. Metrics, however, may vary from one project to another, hindering the reuse of these models. Consequently, these techniques tend to operate within single projects only, despite the fact many large projects share dependencies such as the reuse of common libraries. This makes them potentially vulnerable to similar faults. A solution to a bug provided by the developers of one project may help fix a bug that occur in another (and dependant) project. Moreover, as noted by Lewis *et al.* [3] and Johnson *et al.* [4], techniques based solely on metrics are perceived by developers as black box solutions because they do not provide any insights on the causes of the risky commits or ways for improving them. As a result, developers are less likely to trust the output of these tools.

In this paper, we propose a novel bug prevention approach at commit-time, called BIANCA (Bug Insertion ANticipation by Clone Analysis at commit time). BIANCA does not use metrics to assess whether or not an incoming commit is risky. Instead, it relies on code clone detection techniques by extracting code blocks from incoming commits and comparing them to those of known defect-introducing commits.

One particular aspect of BIANCA is its ability to detect risky commits not only by comparing them to commits of a single project but also to those belonging to other projects that share common dependencies. This is important because complex software systems are not designed in a monolithic way. They have dependencies that make them vulnerable to similar faults. For example, Apache BatchEE [5] and GraphWalker [6] both depend on JUNG (Java Universal Network/Graph Framework) [7]. BatchEE provides an implementation of the jsr-352 (Batch

Applications for the Java Platform) specification [8] while GraphWalker is an open source model-based testing tool for test automation. These two systems are designed for different purposes. BatchEE is used to do batch processing in Java, whereas GraphWalker is used to design unit tests using a graph representation of code. Nevertheless, because both Apache BatchEE and GraphWalker rely on JUNG, the developers of these projects made similar mistakes when building upon JUNG. The issue reports Apache BatchEE #69 and GraphWalker #44 indicate that the developers of these projects made similar mistakes when using the graph visualization component of JUNG. To detect commits across related projects, BIANCA resorts to project dependency analysis.

Another advantage of BIANCA is that it uses commits that are used to fix previous defect-introducing commits to provide guidance to the developers on how to improve risky commits. This way, BIANCA goes one step further than existing techniques by providing developers with a potential fix for their risky commits.

We validated the performance of BIANCA on 42 open source projects, obtained from Github. The examined projects vary in size, domain and popularity. Our findings indicate that BIANCA is able to flag risky commits with an average precision, recall and F-measure of 90.75%, 37.15% and 52.72%, respectively. Moreover, we found that only 8.6% of the risky commits detected by BIANCA match other commits from the same project. This finding stresses the fact that relationships across projects should be taken into consideration for effective prevention of risky commits.

The remaining parts of this paper are organized as follows. In Section 2, we present related work. Sections 3, 4 and 5 are dedicated to the BIANCA approach, the case study setup, and the case study results. Then, Sections 6 and 7 present the threats to validity and a conclusion accompanied with future work.

## 2 Related Work

The work most related to ours comes 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.

### 2.1 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 [9] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [10]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [11].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [12], El Emam *et al.* [13], Subramanyam *et al.* [14] and Gyimothy *et al.* [15] 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.* [16], [17], Demange *et al.* [18] and Palma *et al.* [19] 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.* [20], [21] and Zimmerman *et al.* [22], [23] 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 [24] 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., [25], [26]). 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 [25]. They find that locations that have been recently modified and fixed locations are the most defect-prone. Similarly, Ostrand *et al.* [26] 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.* [27] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [25]. Rahman and Devanbu found that, in general, process-based metrics perform as good as or better than code-based metrics [28].

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 [29]. Hassan [30] 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 [31]. 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, BIANCA takes a different approach in that it leverages dependencies of a project to determine risky changes.

### 2.2 Automatic Patch Generation

Since BIANCA 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.* [32] 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.* [33] generated patches from human-written patches and showed that their tool, PAR, successfully generated patches for 27 of 119 bugs. Tao *et al.* [34] 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. [35], [36], and determine what bugs are best fit for automatic patch generation [37].

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.

## 3 The BIANCA Approach

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

In the first process (Figures 1 and 2), BIANCA 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, BIANCA 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 BIANCA listens to bug (or issue) closing events of major open-source projects (currently, BIANCA is tested with 42 large projects). These projects share many dependencies. Projects can depend on each other or on common external tools and libraries. We perform project dependency analysis to identify groups of highly-coupled projects.

In the second process (Figure 3), BIANCA identifies risky commits within each group so as 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.

### 3.1 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 no-SQL 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, `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 [38], [39], used to detect communities by progressively removing edges from the original network. The connected components of the remaining network form distinct communities. 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 [39]. Other clustering algorithms can also be used.

### 3.2 Building a Database of Code Blocks of Defect-Commits and Fix-Commits

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

**Extracting Commits:** BIANCA listens to bug (or issue) closing events happening on the project tracking system. Every time an issue is closed, BIANCA 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 [40]. 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. But this good practice is not always followed. To make the link between fix-commits and their related issues, we turn to a modified

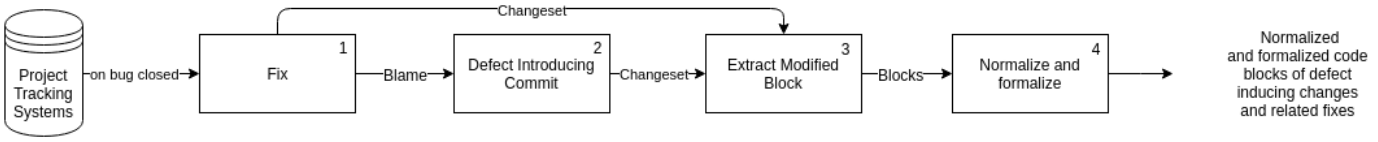


Fig. 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

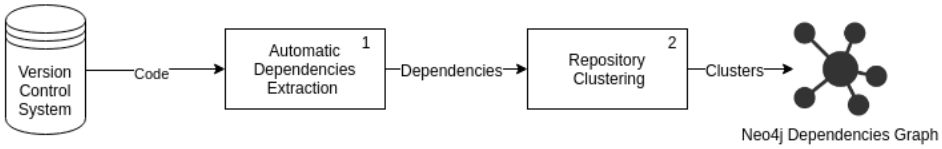


Fig. 2: Clustering by dependency

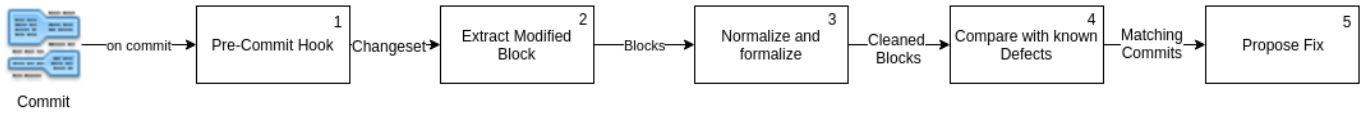


Fig. 3: Classifying incoming commits and proposing fixes

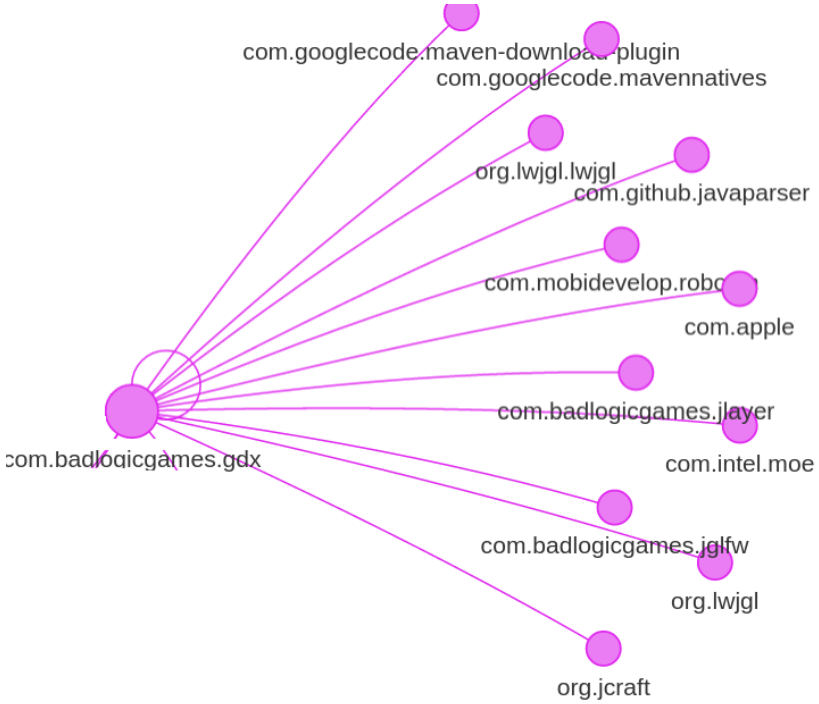


Fig. 4: Simplified Dependency Graph for `com.badlogicgames.gdx`

version of the back-end of commit-guru [41]. Commit-guru is a tool, developed by Rosen *et al.* [41] 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 part of the analysis components for BIANCA. 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.* [42]. Commit-guru implements the SZZ algorithm [43] 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 bug introducing commits (i.e.,

the defect-commits). Priori work showed that Commit-guru is effective in identifying defect-commits and their corresponding fixing commits [44] and to date, the SZZ algorithm, which Commit-guru uses, is considered to be the state-of-the-art in detecting risky commits and its accuracy has been repetitively proven in the literrature. Note that we could use a simpler and more established tool such as Relink [40] to link the commits to their issues and re-implement the classification proposed by Hindle *et al.* [42] on top of it. However, commit-guru has the advantage of being open-source, making it possible to modify it to fit our needs and fine-tune its performance.

**Extracting Code Blocks:** To extract code blocks from fix-commits and defect-commits, we rely on TXL [45], which is a first-order functional programming over linear term rewriting, developed by Cordy et al. [45]. 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]
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* [46], which allow developers to redefine the rules of the grammar and, therefore, apply different rules than the original ones. BIANCA 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 work flow 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.

**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

Algorithm 1 presents an overview of the *extract* and *save* blocks operations of BIANCA. This algorithm receives as argument, 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 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 [47]. 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 BIANCA 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, BIANCA is applicable at the beginning of every project. The only requirement is to use a dependency manager.

### 3.3 Analysing New Commits Using Pre-Commit Hooks

Each time a developer makes a commit, BIANCA 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 $\alpha$ is used to assess the extent beyond which two commits are considered similar. The setting of $\alpha$ is discussed in the case study section.

Pre-commit hooks are custom scripts set to fire off when certain important actions of the versionning 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 run 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. BIANCA 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 [48]–[53], we selected NICAD as the main text-based method for comparing code blocks [54] 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 [55]. 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. BIANCA detects Type 3 clones since they can contain added or deleted code statements, which make them suitable for comparing commit code blocks. The one caveat of using NICAD is that it only works with complete JAVA, C# and C files. To circonvent this issue, we spent a significant amount of time writing a txl grammar that accepts changesets (instead of complete files) helped by TXL creators and maintainers on http://www.txl.ca/forum/.

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 1 [56] 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 [57]. 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 [58]. Then, a percentage of unique statements can be computed and, given the threshold $\alpha$, the blocks are marked as clones.

Another important aspect of the design of BIANCA is the ability to provide guidance to developers on how to improve the 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 BIANCA a practical approach for the developers as they will know why a given

TABLE 1: Pretty-Printing Example

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

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., [41], [44]).
A tool that supports BIANCA should have enough flexibility to allow developers to enable or disable the recommendations made by BIANCA. Furthermore, because BIANCA acts before the commit reaches the central repository, it prevents unfortunate pulls of defects by other members of the organization.

## 4 Case Study Setup

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

### 4.1 Project Repository Selection

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. A different threshold could be used. 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 3 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.

### 4.2 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.
In average, the projects composing our dataset have 77 dependencies. Among the 77 dependencies, in average, 62 dependencies are shared with at least one other project from our dataset.

Table 2 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 data sets 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).

### 4.3 Building a Database of Defect-Commits and Fix-Commits for Performances Evaluation

To build the database against which we assess the performance of BIANCA, we use the same process as discussed in Section 3.2. 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 [31]. We use the commit-guru labels as the baseline to compute the precision and recall of BIANCA. Each time BIANCA classifies a commit as *risky*, we can check if the *risky* commit is in the database of defect-introducing commits.

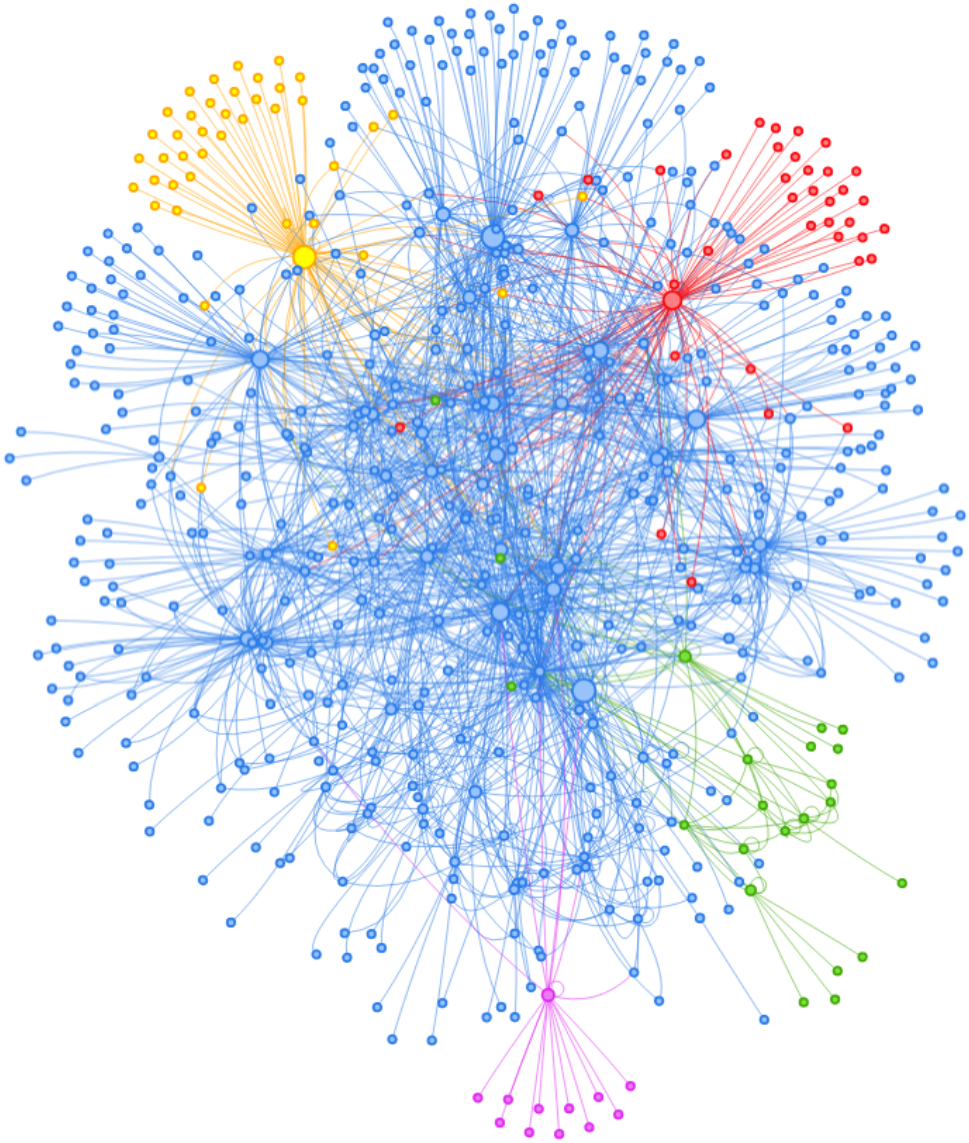

Fig. 5: Dependency Graph

TABLE 2: 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 |

The same evaluation process is used by related studies [13], [59]–[61].

### 4.4 Process of Comparing New Commits

Because our approach relies on commit pre-hooks to detect risky commits, we had to find a way to *replay* past commits. To do so, we *cloned* our test subjects, and then created a new branch called *BIANCA*. 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 *BIANCA* 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 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 4.3. 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 BIANCA

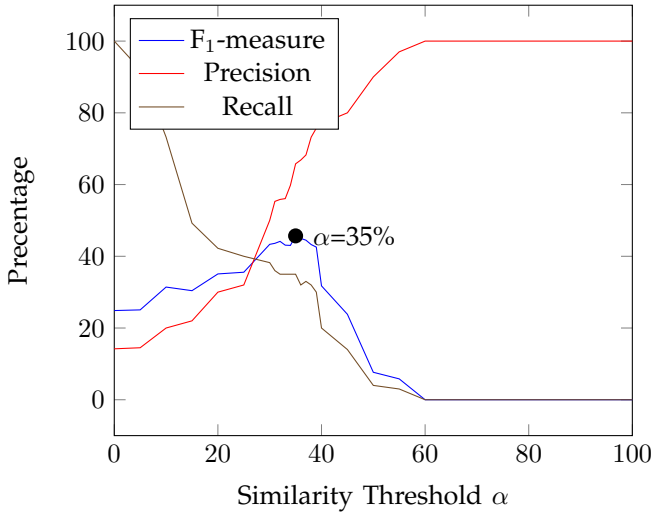


Fig. 6: Precision, Recall and $F_1$-measure variations according to $\alpha$

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 $\alpha$. One possibility would be to test for all possible values of $\alpha$ 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 BIANCA with different values of $\alpha$. Running experiments with all possible $\alpha$ turned out to be computationally demanding given the large number of commits. Testing with all the different values of $\alpha$ amounts to 4e10 comparisons.

To address this, we randomly selected a sample of 1% commits from our dataset and checked the results by varying $\alpha$ from 1 to 100%. Figure 6 shows the results. As we can see, there is a tradeoff between precision and recall, however, after the $\alpha$ = 35% point, we see a drop in recall; hence, we set $\alpha$ = 35% in our experiments.

While $\alpha$ = 35% is not a general rule in clone detection work a threshold of around 30% is considered an adequate by NICAD according to their user manual. This is espacially true for Type 3 clones which contain added or deleted code statements.

[54], [62].

With $\alpha$ = 35%, the experiments took nearly three months to run on 48 Amazon VPS (Virtual Private Server) running in parallel (4e8 comparisons).

### 4.5 Evaluation Measures

Similar to prior work focusing on risky commits (e.g., [29], [31]), 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 BIANCA
- FP: is the number of healthy commits that were classified by BIANCA as risky
- FN: is the number of defect introducing-commits that were not detected by BIANCA
- 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 [41; 63; 64; Kamei2013b], if a defect is not reported within six months then it is not considered.

## 5 Case Study Results

In this section, we show the effectiveness of BIANCA 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 3: 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 |
| Jedis | 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** |

Table 3 shows the results of applying BIANCA 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 BIANCA, 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 $\alpha$ = 35%, BIANCA 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, BIANCA achieves, on average, 37.15% recall (15,316/41,225), and an average $F_1$ measure of 52.72%. The relatively *low* recall is to be expected, since BIANCA considers risky commits that are also in other projects.

Also, out of the 15,316 commits BIANCA 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. This said, we showed, in this paper, that software systems sharing dependencies also share common issues, irrespective to whether these issues represent similar defects or not.

In the following subsections, we compare BIANCA with a random classifier, assess the quality of the proposed fixes, and present the findings of our manual analysis.

### 5.1 Random 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 [65]. 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 to ever use commit code similarity instead of code or process metrics. It is an entirely new way of detecting risky changes. Consequently, comparing to other approaches will not be accurate. Also, few approaches are cross-project at all because they use code or process metrics and models are not easily adaptable. [?]. The only reason we compared our approach with a random classifier is to have a baseline and show that we perform better than a simple baseline.

The random 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%.

### 5.2 Automatic Analysis of the Quality of the Fixes Proposed by BIANCA

One of the advantages of BIANCA 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 3 and 5 proposed fixes [32]–[37]. The average similarity of the first 3 fixes is 40.78% while the similarity of the first five fixes is 44.17%. Results are reported in Table 3.

In the framework of this study, for a fix to be ranked as qualitative it has to reach our $\alpha$=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 $\alpha$=35% threshold. On a per commit basis, BIANCA 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 our $\alpha$=35% threshold.

In other words, BIANCA 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. It is still too early to claim whether BIANCA's recommendations can be useful to developers. For this, we need to conduct user study, which we plan to do as future work.

BIANCA 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 BIANCA, we conducted a manual analysis of the commits classified as *risky* by BIANCA for these projects.

#### 5.2.1 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 makes 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 BIANCA, only 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 a 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.

#### 5.2.2 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 BIANCA were marked so because they matched defect-commits in the FastJSON project.

#### 5.2.3 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 Guave 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.

#### 5.2.4 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 BIANCA could have checked for matching defects is Openhab itself. BIANCA 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 [66]) then we would have achieved a better $F_1$-measure.

#### 5.2.5 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,) BIANCA was only able to detect 9 *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.

#### 5.2.6 Annotations by Excilys ($F_1$-measure = 3.13%)

The last project we analysed manually is Annotations by Excilys. Very much like 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 BIANCA to perform poorly.

Our interpretation of the manual analysis of the best and worst performing projects is that BIANCA 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 BIANCA.

### 5.3 Manual Analysis of the Quality of the Fixes Proposed by BIANCA

In order to further assess the quality of the fixes proposed by BIANCA we manually analysed 250 randomly selected solution proposed by BIANCA (1.63% of 15,316). A proposed solution, in BIANCA terms, is composed of two bug-introducing change and fix couple; $c_a$ and $c_b$. To analyse the quality of the fixes we manually compare the fix applied in $c_a$ to the one applied in $c_b$. In this section, we provide a sub-sample of the 250 proposed fixes we analyzed. This sample is not randomly selected but composed of pertinent examples hand-picked by one of the authors. Out of the 250 manually analyzed proposed fixes, we were able to confirm that 34.4% of them (86/250) would have provided a direct solution to the bug to fix. A direct solution is where the fix applied in $c_a$ could be easily adapted to fix the bug of $c_b$ by, for example, changing only variables and function names. Ultimately the fix applied in $c_b$ is extremely similar to the one applied in $c_a$. Another 126 (50.4%) proposed fixes have been manually identified as providing directions towards the fix. Fixes that provide a direction toward a fix means that the fix applied in $c_a$ would need significant transformation in order to be used in $c_b$. Significant transformation can involve large refactoring, eleminating or adding part of the fix, adapt data structures and so on. Identifying proposed fixes that provide a direction toward the actual fix require a significant time investement as $c_a$ and $c_b$. have be toroughly understood. In addition, the systems where $c_a$ and $c_b$ have been applied have to be understood too in order to assess the pertinence and quality of $c_a$ with regards to $c_b$. In order to identify proposed fixes that provide directions towards the actual fix, we allowed ourselves a 30 minutes period. In this 30 minutes period, we were investigating both systems and trying to understand the proposed fixes in their contexts. After the 30 minutes, if we were unable to distinguish a similarity between the fix of $c_a$ and the one of $c_b$; then, we categorize the proposed fix as not helpfull. While 30 minutes can be perceived as long period of time to analyze the output of tools such as BIANCA, we have to keep into account that developers exposed to these results would have already have a comprehensive understanding of their system and, most likely, would be able to understand the other system quicker than we could. In addition, 30 minutes out of the bug fixing process of industrial java systems is representing only 10.4% of the total time required to fix a bug according to Weiss *et al* [**?**]. In their attempt to predict *how long would it take to fix a bug* they discovered that the average time to craft a fix was 4.8±6.3 hours in one open-source java industrialize-sized system called JBoss [?]. We argue that, even if 30 minutes are required to identify a direction from a proposed fix it will still save time over the 4.8 ±6.3 hours required on JBoss. Indeed, fixes proposed the developers have to be from systems that are in the same cluster with regards to their depedencies. Consequently, it is likely that the developer would recognize types and structures comming from the other system as they are also used in the system at hand. To summarize, 86 proposed fixes have been identified as direct solution in a few minutes while 126 fixes have been identified as providing directions towards the acutal fix within a 30 minutes period. Finally, we were not able to manually identify any relevant similarity, inside our 30 minutes period, for 38 proposed fixes (15.2%). Overall, the manual analysis took over 100 hours.

In what follows we present hand picked diffs that shows example of directly applicable fixes and fixes that give direction toward the actual fix. Diffs are a representation of two different version of the same source code. The version before the commit and the version after the commit. The modifications between the two versions are identified by "-"s and "+"s. If a line starts with a "-" it means that the given line has been deleted and is not longer present in the after-commit version of the code. In the opposite, if a line begins with a "+"; then, this line has been added to the source code and will be present in the after-commit version of the code. Finally, the "@@" signs determine where the code snippet take place in the overall file by specifying the line number in the before- and after- commit versions of the source code.

In this next example, we present two couples $c_{a_1}$ (figure 7) and $c_{b_1}$ (figure 8 that belong to X and Y, respectively. null check

In this next example, we present two couples $c_{a_3}$ (figure 7) and $c_{b_3}$ (figure 8 that belong to Jsoup and Orientdb, respectively. $c_{a_3}$ has been commited in November 2013 while $c_{b_3}$ came two years later in October 2015. This is an example where the proposed fix give a direction towards the actual fix. In $c_{a_3}$ and $c_{b_3}$ we can see that the developers are working with the `StringBuilder` class. The `StringBuilder` class is explained as follows in the Java documentation: *A mutable sequence of characters. This 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 a good Java practice as it improves performances. In both cases, the code have been modified to avoid the appending of `null` string. On JSoup, it is done by the method `shouldCollapseAttribute` which is responsible to find the value is empty. On Orientdb, it is done by a simple null check on the string named `right`. Note that this kind of *bug* would not have been picked up by a static analysis tool such as PMD [**?**] 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.

In addition to the presented couples of fix/proposed-fix presented our manual analysis yield some interesting insights on fix patterns that span accross clusters. The first pattern is linked to the desire of a developer to produce a backward compatible source code. As presented in Figure 11 the developer replace the invocation to the `isEmpty` method to an invocation of `length() == 0` in order to make its code compatible with Java 1.5. Indeed, Java, in its 1.5 version does not provide the `isEmpty` method which was introduced later. Here, we can suspect that the development environment of the developer does not match the production environment and, consequently, a regression has been introduced. This fix-pattern of replacing a newly added method in a library by an old one occured 3 times in our direct fix category. Consequently, we investigated further in the 15,316 results and found 363 instances of this pattern. This pattern accounts for 2.35% of the total detection.

Another interesting patterns we were able to detect in the 250 bugs randomly selected (2/250) and confirm in the remaining 15,116 (is a pattern involving the java keyword `final` (48/15,116). In Java, the keyword refers to the non-transitiviy or immutability of variables. ««««< HEAD As investigated by Colblenz *et al.* that important requirements, such as expressing immutability constraints, were completely understood nor available in Java [@coblenz2016exploring]. ======= As investigated by Colblenz *et al.* that important requirements, such as expressing immutability constraints, were completely understood nor available in Java. »»»»> cf51ed833009fbbc7b7e2b740266d6f52571ffbf The use of this keyword by junior software developer is often adhoc. In many instances, we found developer fixing a bug report stating that a given variable should not change regardless of the on-going event. Rather than fixing the root cause (i.e. understanting why and where the variable changes) developers use the final keyword. It is noteworthy that an attempt to modify the value of a final variable, in Java, would result in a compilation error. Figure 10 show a commit displaying this fix-pattern.

The last pattern we discovered consists in null-checking wrapped types such as `Integer`. In Java, eight primitive types exist: boolean, byte, char, short, int, long, float and double. These types are the most basic data types available and are not extending the top-most class `Object`. The primitive types cannot have `null` as value. For example, the default value of an `int` is `0`. All other classes of Java are, by default, specialization of `Object`. In addition to these primitive type, the JDK (Java Development Kit) contains wrapped types that wraps the value of a primitive types in an `Object`. These wrapped types contain a single field (the primitive type they wrap) and several methods for conversion and comparison, for example. The wrapped types, as `Object`, can be `null` and invoking one of their method on a `null` instance would result in a `null pointer exception`. We found 2/250 and 192/15,316 instances of null-checking wrapped types as displayed in Figure **??**. Unlike the first two patterns, this last pattern could be detected by static checker such as PMD. However, it seams that this particuliar case evades the detection of potentially null objects access.

## 6 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 analysed by BIANCA were selected from Github based on their popularity and the ability to mine their past issues and also 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 pojects written in other languages. However, similar to Java, one can write a TXL grammar for a new language then BIANCA can work since BIANCA relies on TXL. Finally, we use NICAD as the code comparison engine. The accuracy of NICAD affects the accuracy of BIANCA. 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 us from using other text-based code comparisons engines, if need be. 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).

## 7 Conclusion

In this paper, we presented BIANCA (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 90.75% precision and 37.15% recall. BIANCA uses clone detection techniques and project dependency analysis to detect risky commits within and across dependant projects. BIANCA operates at commit-time, i.e., before the commits reach the central repository. In addition, because it relies on code comparison, BIANCA does not only detect risky commits but

```
@@ -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. 7: OrientDB commit #444db817ee9404b17c1208df51781ce9cb6a2666

```
     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('"');
+        }
     }

     /**
@@ -100,6 +111,15 @@ 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. 8: Jsoup commit #6c4f16f233cdfd7aedef33374609e9aa4ede255c

```
@@ -178,7 +178,7 @@ static String getCharsetFromContentType(String contentType) {
         if (m.find()) {
             String charset = m.group(1).trim();
             charset = charset.replace("charset=", "");
-            if (charset.isEmpty()) return null;
+            if (charset.length() == 0) return null;
             try {
                 if (Charset.isSupported(charset)) return charset;
                 charset = charset.toUpperCase(Locale.ENGLISH);
```

Fig. 9: Jsoup commit #bb16e0693819afb821bce6943d8cbd178266a63e

```
@@ -387,8 +387,31 @@
        /*
         * Set all bytes between {@code startOffset} and {@code endOffset} to zero.
         * Area between offsets must be ready for write once clear finishes.
         */
-     public abstract void clear(long startOffset, long endOffset);
+     public abstract void clear(final long startOffset, final long endOffset);
```

Fig. 10: Mapdb commit #e4542a9b0e7907a11cab7a98467a770532a87e09

```
@@ -478,9 +478,9 @@ private void syncMetaFromCache(String topologyId,
    TopologyMetricContext context)
      private void syncMetaFromRemote(String topologyId, TopologyMetricContext context) {
          try {
              int memSize = context.getMemMeta().size();
-             int zkSize = (Integer) stormClusterState.get_topology_metric(topologyId);
+             Integer zkSize = (Integer)stormClusterState.get_topology_metric(topologyId);

-             if (memSize != zkSize) {
+             if (zkSize != null  && memSize != zkSize.intValue()) {
                  ConcurrentMap<String, Long> memMeta = context.getMemMeta();
                  for (MetaType metaType : MetaType.values()) {
                      List<MetricMeta> metaList = metricQueryClient.getMetricMeta(
                          clusterName, topologyId, metaType);
@@ -908,4 +908,4 @@ public String toString() {
          public Assignment newAssignment;
          public Map<Integer, String> task2Component;
      }
-}
+}
```

Fig. 11: JStorm commit #6dc60b06a0a8880cf37c88a462a213737703be80

also makes recommendations to developers on how to fix them. We believe that this makes BIANCA a practical approach for preventing bugs and proposing corrective measures that integrates well with the developers work flow 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 BIANCA. Finally, another improvement to BIANCA would be to support Type 4 clones.

## 8 Reproduction Package & Dataset

As described in section 4.4 our experimentations rely heavily on virtual machines instrumentation and coordination. Providing a straightforward reproduction package in this condition is very challenging. However, we are happy to share our consilolidated dataset: https://github.com/MathieuNls/bianca-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 [?]. Finally, the repository table describe the repository used in terms of url, name and ingestion status.

## 9 Acknowledgments

We are thankful to amazon for their awseducate plateform without which this study would have cost thousands of dollars in server fees. We are also greatful to the students and engineers that participated in our user study.

## 10 References

[1] D. Lo, "A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction," in *2013 17th european*

*conference on software maintenance and reengineering*, 2013, pp. 331–334.

[2] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th international conference on software engineering (iCSE)*, 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] 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 35th international conference on software engineering*, 2013, pp. 672–681.

[5] The Apache Software Foundation, "Apache BatchEE." 2015.

[6] Graphwalker, "GraphWalker for testers." 2016.

[7] Y.-b. B. Joshua O'Madadhain, Danyel Fisher, Scott White, Padhraic Smyth, "Analysis and Visualization of Network Data using JUNG," *Journal of Statistical Software*, vol. 10, no. 2, pp. 1–35, 2005.

[8] Chris Vignola, "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 352." 2014.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] M. Nayrolles, F. Palma, N. Moha, and Y.-G. Guéhéneuc, "SODA : A Tool Support for the Detection of SOA Antipatterns," in *International conference on service oriented computing lNCS 7759*, 2012, pp. 451–456.

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

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

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

[20] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on software engineering - iCSE '05*, 2005, p. 580.

[21] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceeding of the 28th international conference on software engineering - iCSE '06*, 2006, p. 452.

[22] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *Third international workshop on predictor models in software engineering (pROMISE'07: ICSE workshops 2007)*, 2007, pp. 9–9.

[23] 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.

[24] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings. 27th international conference on software engineering, 2005.*, 2005, pp. 284–292.

[25] A. Hassan and R. Holt, "The top ten list: dynamic fault prediction," in *21st iEEE international conference on software maintenance (iCSM'05)*, 2005, pp. 263–272.

[26] 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.

[27] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *29th international conference on software engineering (iCSE'07)*, 2007, pp. 489–498.

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

[29] 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.

[30] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 iEEE 31st international conference on software engineering*, 2009, pp. 78–88.

[31] 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.

[32] 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.

[33] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th international conference on software engineering (iCSE)*, 2013, vol. 1, pp. 802–811.

[34] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the 22nd aCM sIGSOFT international symposium on foundations of software engineering*, 2014, pp. 64–74.

[35] V. Dallmeier, A. Zeller, and B. Meyer, "Generating Fixes from Object Behavior Anomalies," in *24th iEEE/ACM international conference on automated software engineering*, 2009, pp. 550–554.

[36] 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 *2012 34th international conference on software engineering (iCSE)*, 2012, pp. 3–13.

[37] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?" in *Soft-*

*ware reliability engineering (iSSRE), 2015 iEEE 26th international symposium on*, 2015, pp. 427–437.

[38] 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.

[39] 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.

[40] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th aCM sIGSOFT symposium and the 13th european conference on foundations of software engineering.*, 2011, pp. 15–25.

[41] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the10th joint meeting on foundations of software engineering*, 2015, pp. 966–969.

[42] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?" in *Proceedings of the 2008 international workshop on mining software repositories - mSR '08*, 2008, p. 99.

[43] S. Kim, T. Zimmermann, K. Pan, and E. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *21st iEEE/ACM international conference on automated software engineering (aSE'06)*, 2006, pp. 81–90.

[44] 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.

[45] J. R. Cordy, "Source transformation, analysis and generation in TXL," in *Proceedings of the 2006 aCM sIGPLAN symposium on partial evaluation and semantics-based program manipulation - pEPM '06*, 2006, pp. 1–11.

[46] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile Parsing in TXL," in *Proceedings of iEEE international conference on automated software engineering*, 2003, vol. 10, pp. 311–336.

[47] 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.

[48] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the conference of the centre for advanced studies on collaborative research: Software engineering-volume 1*, 1993, pp. 171–183.

[49] 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.

[50] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th annual international conference on automated software engineering*, 2001, pp. 107–114.

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

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

[53] 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.

[54] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *Proceedings of the 19th iEEE international conference on program comprehension*, 2011, pp. 219–220.

[55] 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.

[56] CHANCHAL K. ROY, "Detection and Analysis of Near-Miss Software Clones," PhD thesis, Queen's University, 2009.

[57] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings iEEE international conference on software maintenance*, 1999, pp. 109–118.

[58] 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.

[59] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th aCM sIGSOFT symposium and the 13th european conference on foundations of software engineering - sIGSOFT/FSE '11*, 2011, p. 311.

[60] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceeding of the 8th working conference on mining software repositories - mSR '11*, 2011, p. 207.

[61] 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.

[62] C. K. Roy and J. R. Cordy, "An Empirical Study of Function Clones in Open Source Software," in *2008 15th working conference on reverse engineering*, 2008, pp. 81–90.

[63] T.-h. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An Empirical Study of Dormant Bugs Categories and Subject Descriptors," in *Mining software repository*, 2014, pp. 82–91.

[64] 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.

[65] 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.

[66] 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. 5, pp. 1–26, 2012.