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*—abstract goes here**

***Keywords*-Software Metrics; Risky Software Commits; Bug Prediction; Clone Detection; Software Maintenance**

I. Introduction

Research in software maintenance continues to evolve to include areas like mining bug repositories, bug analytics, 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. In addition, these techniques operate within single projects only, despite the fact many large projects share dependencies such as the reuse of common libraries. This makes them vulnerable to similar faults [REF]. 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 and ways to 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 upcoming commit is risky. Instead, it relies on code clone detection techniques by extracting code blocks from upcoming 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 errors when using the graph visualization component of JUNG. To detect commits across 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 improve risky commits. This way, BIANCA goes one step further than existing techniques by providing developers with a potential fix for their risk commits.

We validate 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 and precision, recall and F-measure of 90.75%, 37.15% and 52.72%, respectively. Moreover, we find that only 8.6% of the risky commits detected by BIANCA match other commits from the same project. This finding indicates that relationships across projects need to be considered for effective prevention of risky commits.

The remaining parts of this paper are organized as follows. In Section II, we present related work. Sections III and IV are dedicated to presenting the BIANCA approach and its evaluation. Then, Sections V, VII and, VI assess the threats to validity, present the key lessons learned and, a conclusion accompanied with future work, respectively.

II. Related Work

Predicting crashes, faults, and bugs is very popular research area. The main goal of existing studies is to save on manpower when dealing with bugs and crashes. There are two distinct trends in crash, fault and bug prediction: History analysis and current version analysis.

In the history analysis, researchers extract and interpret information from the system. The idea being that the files or locations that are the most frequently changed are more likely to contain a bug. Additionally, some of these approaches also assume that locations linked to a previous bug are likely to be linked to a bug in the future. On the other hand, approaches using only the current version to predict bugs assume that the current version, i.e., its design, call graph, quality metrics and more, will trigger the appearance of the bug in the future. Consequently, they do no require the history and only need the current source-code.

In the remaining of this section, we describe approaches belonging to the two families.

A. Change logs approaches

Change logs based approaches rely on mining the historical data of the application and more particularly, the source code *diffs*. A source code *diffs* contains two versions of the same code in one file. Indeed, it contains the lines of code that have been deleted and the one that has been added.

Naggapan *et al.* studied the churns metric and how it can be connected to the apparition of new defects in complex software systems. They established that relative churns are, in fact, a better metric than classical churn [9] while studying Windows Server 2003.

Hassan interested himself with the entropy of change, i.e. how complex the change is [10]. Then, the complexity of the change, or entropy, can be used to predict bugs. The more complex a change is, the more likely it is to bring the defect with it. Hassan used its entropy metric, with success, on six different systems. Before this work, Hassan, in collaboration with Holt proposed an approach that highlights the top ten most susceptible locations to have a bug using heuristics based on *diffs* file metrics [11]. Moreover, their heuristics also leverage the data of the bug tracking system. Indeed, they use the past defect location to predict new ones. The conclusion of these two approaches has been that recently modified and fixed locations where the most defect-prone compared to frequently modified ones.

Similarly to Hassan and Hold, Ostrand *et al.* predict future crash location by combining the data from changed and past defect locations [12]. The main difference between Hassan and Holt and Ostrand *et al.* is that Ostrand *et al.* validate their approach on industrial systems as they are members of the AT&T lab while Hassan and Hold validated their approach on open-source systems. This proved that these metrics are relevant for open-source and industrial systems.

Kim *et al.* applied the same recipe and mined recent changes and defects with their approach named bug cache [13]. However, they are more accurate than the previous approaches at detecting defect location by taking into account that is more likely for a developer to make a change that introduces a defect when being under pressure. Such changes can be pushed to the revision-control system when deadlines and releases date are approaching.

B. Single-version approaches

Approaches belonging to the single-version family will only consider the current version of the software at hand. Simply put, they do not leverage the history of changes or bug reports. Despite this fact, that one can see as a disadvantage compared to approaches that do leverage history; these approaches yield interesting results using code-based metrics.

Chidamber and Kemerer published the well-known CK metrics suite [14] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [15]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [16].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [17], El Emam *et al.* [18], Subramanyam *et al.* [19] and Gyimothy *et al.* [20] 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.* [21], [22], Demange *et al.* [23] and Palma *et al.* [24] used Moha et *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.

Finally, Nagappan *et al.* [25], [26] and Zimmerman [27], [28] further refined metrics-based detection by using statical analysis and call-graph analysis.

mathieu: Emad, we need to compare to other approaches in terms of precision, recall and methodology. Is there any papers we must compare to ?
Emad: Sure, I will rework the related works section. I will divide it based on file/module-level prediction and commit-level prediction

III. The BIANCA Approach

Figure 1 shows an overview of the BIANCA approach, which consists of two main phases. In the first phase, 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 introduce 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).

Emad: Figure 1 needs to be split, maybe into 2 figures. Also, we should add numbers in the figures that correspond to each step and refer to these numbers in the text. As it is now, the figure is very hard to follow.

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. 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 in the second phase 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.

Emad: A possible discussion point to examine is how much history is needed vs. accuracy of BIANCA

A. Clustering project repositories

We cluster project repositories 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 **??**. 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 [REF].

Figure 2 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 [29], [30], 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 [REF]. 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- and fixing-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 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 [31]. 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 version of the back-end of commit-guru [32]. Commit-guru is a tool, developed by Rosen *et al.* to detect *risky commits*. A risky commit is a commit that introduces a defect in the program. 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.* [33] (see Table I). Commit-guru implements the SZZ algorithm [REF] 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

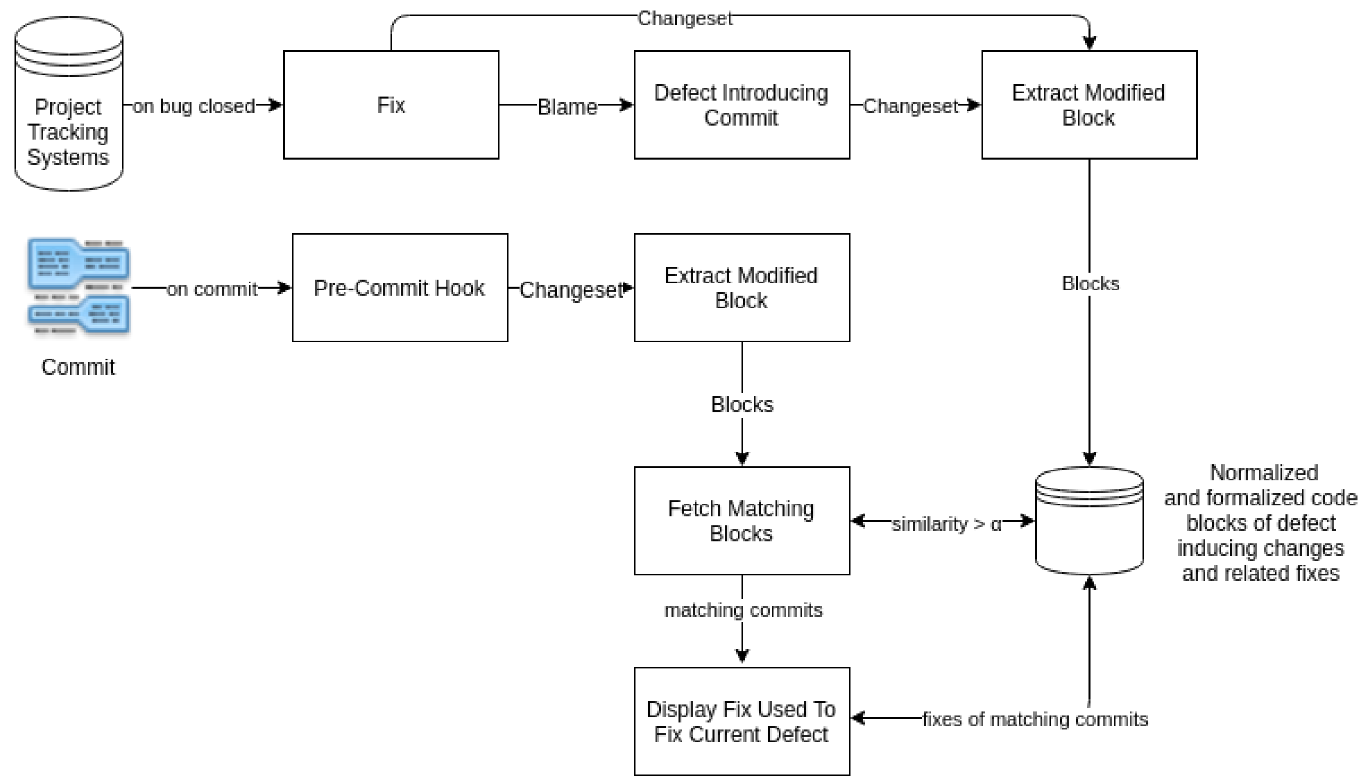

Fig. 1. Overview of the BIANCA Approach

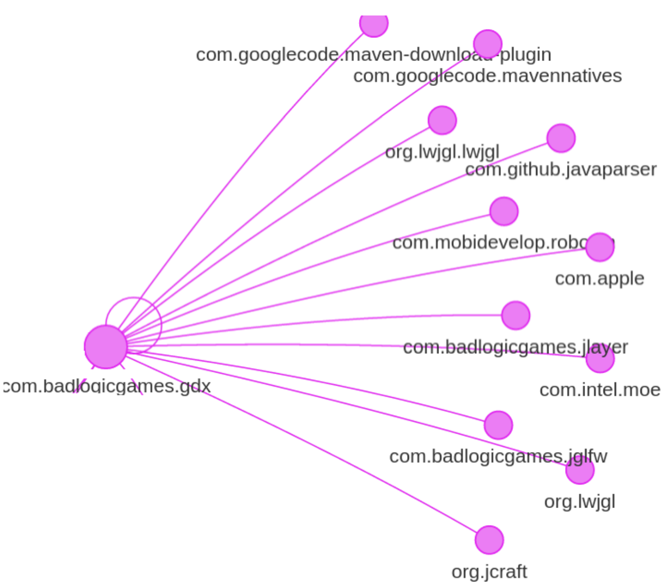

Fig. 2. Simplified Dependency Graph for `com.badlogicgames.gdx` (Zoomed from south of Figure 3)

TABLE I
WORDS AND CATEGORIES USED TO CLASSIFY COMMITS

Category	Associated Words	Explanation
Corrective	bug, fix, wrong, error, fail, problem, patch	Processing failure
Feature Addition	new, add, requirement, initial, create	Implementing a new requirement
Merge	merge	Merging new commits
Non Functional	doc	Requirements not dealing with implementations
Perfective	clean, better	Improving performance
Preventive	test, junit, coverage, asset	Testing for defects

in identifying defect-commits and their corresponding fixing commits [REF TSE] and to date, the SZZ algorithm, which Commit-guru uses, is considered to be the state-of-the-art in detecting risky commits. Note that we could use a simpler and more established tool such as Relink [31] to link the commits to their issues and re-implement the classification proposed by Hindle *et al.* [33] 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 [34], which is a first-order functional programming over linear term rewriting, developed by Cordy et al. [34]. 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* [35], 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 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.

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

C. Analysing New Commits Using Pre-Commit Hooks

Emad: Another possible discussion point to examine is the impact of α vs. accuracy of BIANCA, i.e., plot α from 0-1 vs. f-measure

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 α is used to assess the extent beyond which two pair of 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 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 is run first, before the developer types in 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 [37–42], we selected NICAD as the main text-based method for comparing code blocks [43] 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 [44]. 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 to comparing commit code blocks.

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 II [45] shows how this can improve the accuracy of clone detection with three `for` statements, `for (i=0; i<10; i++)`, `for (i=1; i<10; i++)` and `for (j=2; j<100; j++)`. The pretty-printing allows 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 [46]. In addition to the pretty-printing, code can be normalized and filtered to detect different classes of clones and match user preferences.

Emad: I would move the [@Iss2009] citation to the table caption.

TABLE II
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 extracted, pretty-printed, normalized and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [47]. Then, a percentage of unique statements can be computed and, given the threshold, 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 databse the fix-commit corresponding to the matching defect-commit and present it to the developer. This way, BIANCA goes one step further than existing techniques, based mainly on statistical models, by providing a practical way on how to fix (or at least reasons about) the risky commit. A tool that supports BIANCA should have enough flexibility to allow developers to enable or disable the recommendations made by BIANCA.

We believe that this can make BIANCA 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., [REF]). Furthermore, because BIANCA acts before the commit reaches the central repository, it prevents unfortunate pulls of defects by other members of the organization.

IV. EVALUATION

Emad: I suggest splitting this section up into Case Study Setup (subsection A - E) and then a Cast Study Results section (subsection F)

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?

Emad: Another issue that we need to evaluate is do the proposed fixes help? More on this later

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. 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 IV for the list of projects), including those from some of major open-source contributors including Alibaba, Apache Software Foundation, Eclipse, Facebook, Google and Square.

Emad: I assume that these are also projects that are not forked? Should we add that

B. Project Dependency Analysis

Figure 3 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 3 are proportional to the number of connections from and to the other nodes.

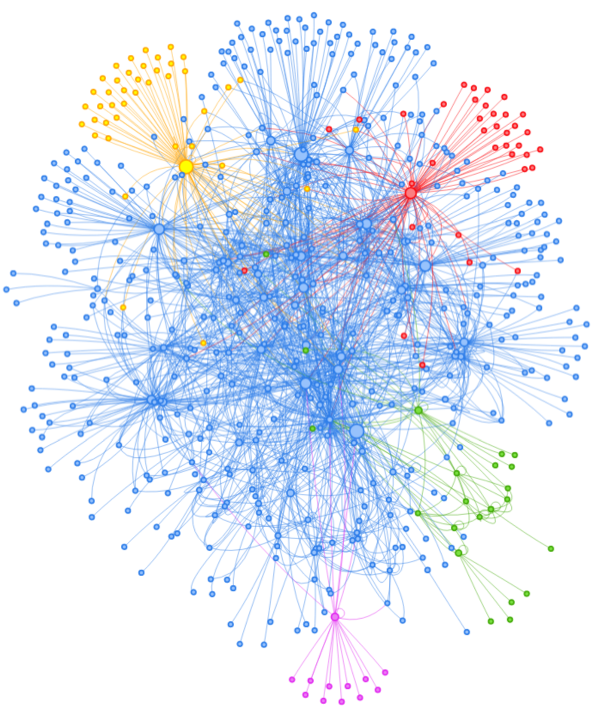

Fig. 3. Dependency Graph

As shown in Figure 3, 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 III
COMMUNITIES IN TERMS OF ID, COLOR CODE, CENTROIDS, BETWEENNESS AND NUMBER OF MEMBERS

#ID	Community	Centroids	Betweenness	# Members
1	Blue	Storm	24525	479
2	Yellow	Alibaba	24400	42
3	Red	Hadoop	16709	37
4	Green	Openhab	3504	22
5	Purple	Libdx	6839	12

Table III 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).

C. Building a database of defect-commits and fix-commits

Emad: There is a lot of repetition here about commit-guru from section 3B. I say remove the commmit-guru text and just say as discussed earlier in section 3B

To validate the results obtained by BIANCA, we needed to use a reliable approach marking defect-commits. For this, we turned to Commit-guru [48] which has the ability to unwind the complete history of a project and label commits as defect-commits if they appear to be linked to a closed issue. We use the commit-guru labels as the baseline to compute the precision and recall of BIANCA. The process used by Commit-guru to identify commits that introduce a defect is simple and reliable in terms of accuracy and computation time [REF TSE]. First, Commit-guru downloads all the issues that were classified as bug by the project team and closed by a commit from the project management system. Second, for each issue, Commit-guru extracts the commit that fixed the issue which is simply known as *fix* or *fix commit*. From the *fix* commit, Commit-guru computes the *blame/annotate* scm operation. The *blame/annotate* allows to retrieve the parent commits of the *fix commit*. The parent commits of the *fix commits* are known as *defect introducing-commits* as they introduced modification that lead to an issue and a fix. Finally, all the defect-introducing commits compose a database against which we can compare the performances of our approach in terms of precision and recall. Each time BIANCA classifies a commit as *risky*, we can confirm the correctness of the classification by checking if the *risky* commit is in the database of defect-introducing commits. The same evaluation process is used by other approaches in the field [18], [49–51].

D. Process of comparing new commits

Emad: This part is a little complex and it is critical to understand it. I recommend adding a figure here to help with the explanation of the methodology described here.

As our approach relies on commit pre-hooks to detect risky commit, 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 on field 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 BIANCA and c_2 is proposed to the developer as a potential solution for the defect introduced in c_3.

To measure similarity between pairs of commits, we chose threshold of (α = 35%). We have made several incremental attempts, starting from 10%. For each attempt, we incremented α by 5% and observed the obtained precision and recall. The current value seems to offer the best results. It should also be noted that in clone detection a threshold of around 30% is considered an adequate threshold above which two code blocks are deemed to be clones, especially for clones of Type 3, which contain added or deleted code statements.

E. Evaluation Measures

Emad: We can also use ROC, which I recommend. For some reason our recall is low, but I this is due to two facts: 1) we only detect risky commits that show up in other projects and 2) the risky commit data is unbalanced (i.e., most commits are not risky). Using ROC will help alleviate the second problem.

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: This is the number of defect-commits that were properly classified by BIANCA
- FP: This is the number of healthy commits that were classified by BIANCA as risky
- FN: This is the number of defect introducing-commits that were not detected by BIANCA

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = 2.\frac{precision.recall}{precision + recall}$$

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 [32], [48], [52], [53], if a defect is not reported within six months then it does not exist.

F. Results of BIANCA

Emad: I think a big part of BIANCA is the fact that it shows you potential fixes. That part is not evaluated at all right now. I think we can evaluate it in two possible ways: 1) to run a similarity analysis, e.g., cosine distance, between the actual fix for the defect-commit and the top 3 or top X fixing commits that we recommend or 2) we can take a statistically significant sample (would be around 400 commits) of the 15K risky commits, examine their fixing commits to the top 3 or top X fixing commits. The second appraoch is manually intensive.

Table IV shows the results of applying BIANCA in terms of 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 α = 35%, BIANCA achieves in average a precision of 90.75% (13,899/15,316) commits identified as risky. Theses

TABLE IV

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 (%) AND THE AVERAGE TIME DIFFERENCE BETWEEN DETECTED AND ORIGINAL.

Organization	Project Name	Short Description	NoC	#Commits	#Bug Introducing Commit	Detected	Precision	Recall	F_1	Time Diff
Alibaba	druid	Database connection pool	3309	4775	1260	787	88.44	62.46	73.21	599.19
	dubbo	RPC framework	1715	1836	119	61	96.72	51.26	67.01	363.34
	fastjson	JSON parser/generator	2002	1749	516	373	95.71	72.29	82.37	607.79
	jstorm	Stream Process	1492	215	24	21	90.48	87.50	88.96	635.24
Apache	hadoop	Distributed processing	9108	14154	3678	851	86.84	23.14	36.54	469.97
	storm	Realtime system	2209	7208	951	444	86.26	46.69	60.58	530.57
Clojure	clojure	Programming language	335	2996	596	46	86.96	7.72	14.18	477.33
Dropwizard	dropwizard	RESTful web services	964	3809	581	179	96.65	30.81	46.72	482.93
	metrics	JVM metrics	335	1948	331	129	95.35	38.97	55.33	444.55
Eclipse	che	Eclipse IDE	7818	1826	169	9	88.89	5.33	10.05	671.62
Excilys	Android Annotations	Android Development	1059	2582	566	9	100.00	1.59	3.13	258.00
Facebook	fresco	Images Management	1007	744	100	68	92.65	68.00	78.43	532.91
Gocd	gocd	Continuous Delivery server	16735	3875	499	297	91.58	59.52	72.15	567.28
Google	auto	source code generators	257	668	124	95	100.00	76.61	86.76	594.48
	guava	Google Libraries for Java 6+	1731	3581	973	592	98.48	60.84	75.22	539.79
	guice	Dependency injection	716	1514	605	104	85.58	17.19	28.63	423.22
	iosched	Android App	1088	129	9	6	100.00	66.67	80.00	578.56
Gradle	gradle	Build system	11876	37207	6896	1557	97.50	22.58	36.67	500.55
Jankotek	mapdb	Concurrent datastructures	267	1913	691	440	94.32	63.68	76.03	479.93
Jhy	jsoup	Parser	136	917	254	153	87.58	60.24	71.38	505.34
Libdx	libgdx	Java game development	4679	12497	3514	1366	87.70	38.87	53.87	483.06
Netty	netty	Event-driven application	2383	7580	3991	1618	89.43	40.54	55.79	569.02
Openhab	openhab	Home Automation Bus	5817	8826	28	2	100.00	7.14	13.33	857.50
Openzipkin	zipkin	Distributed tracing system	397	799	176	73	87.67	41.48	56.31	569.40
Orfjackal	retrolambda	Backport of Java 8's lambda	171	447	97	35	94.29	36.08	52.19	272.24
OrientTechnologie	orientdb	Multi-Model DBMS	2907	13907	7441	2894	86.77	38.89	53.71	511.80
Perwendel	spark	Sinatra for java	205	703	125	82	97.56	65.60	78.45	453.16
PrestoDb	presto	Distributed SQL query	4381	8065	2112	991	90.62	46.92	61.83	479.81
RoboGuice	roboguice	Google Guice on Android	1193	1053	229	70	91.43	30.57	45.82	401.58
Lombok	lombok	Additions to the Java language	1146	1872	560	212	91.98	37.86	53.64	514.00
Scribejava	scribejava	OAuth library	218	609	72	16	93.75	22.22	35.93	633.18
Square	dagger	Dependency injector	232	697	144	84	90.48	58.33	70.93	681.95
	javapoet	Java API	66	650	163	113	100.00	69.33	81.88	504.25
	okhttp	HTTP+HTTP/2 client	344	2649	592	474	93.04	80.07	86.07	500.80
	okio	I/O API for Java	90	433	40	24	100.00	60.00	75.00	348.66
	otto	Guava-based event bus	84	201	15	15	93.33	100.00	96.55	635.80
	retrofit	Type-safe HTTP client	202	1349	151	111	99.10	73.51	84.41	563.83
StephaneNicolas	robospice	Android library	461	865	113	39	87.18	34.51	49.45	832.37
ThinkAurelius	titan	Graph Database	2015	4434	1634	527	90.13	32.25	47.51	443.74
Jedis	jedis	Redis client	203	1370	295	226	92.04	76.61	83.62	535.03
Yahoo	anthelion	Plugin for Apache Nutch	1620	7	0	-	-	-	-	-
Zxing	zxing	1D/2D barcode image	3030	3253	791	123	94.31	15.55	26.70	465.59
Total			**96003**	**165912**	**41225**	**15316**	**90.75**	**37.15**	**52.72**	**524.86**

commits triggered the opening of an issue and had to be fixed later on. On the other hand, BIANCA achieves in average 37.15% recall (15,316/41,225) and an average F_1 measure of 52.72%. 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 project using the same dependencies are, in fact, likely to introduce similar defects. Indeed, if developers were to introduce the same defect several time inside the same project, then the proportion of commit classified as *risky* per BIANCA because they match a defect-commit inside the same project would be higher than 8.6%. This is a potentially a crucial finding for researchers aiming to achieve cross-project defect prevention, regardless of the technique (i.e. statistical model, AST comparison, code comparison, etc.) employed.

1) Manual Analysis: In what follows, we will describe, in details, the three projects with the highest and the lowest F_1-measure, respectively.

The three highest performing projects are otto by square, JStorm by Alibaba and auto by Google. They reach F_1-measures of 96.5% (100.00% precision and 76.61% recall), 88.96% (90.48% precision and 87.50% recall) and 86.76% (90.48% precision and 87.50% recall), respectively. The three lowest performing projects are Android Annotations by Excilys (100.00% precision and, 1.59% recall, 3.13% F_1-measure), che by Eclipse (88.89% precision, 5.33% recall and, 10.05% F_1-measure) and, openhab by Openhab 100.00% precision, 7.14% recall and, 13.33% F_1-measure). To interpret such high and low F_1-measures, we conducted a manual analysis of the commit classified as *risky* by BIANCA for the six projects.

a) Otto by Square (96.5%):

At first, the F_1-measure of Otto by Square seems surprising given what features it provides. Indeed, otto by Square provides a Guava-based event bus. While it does have dependencies that could open it up to the same issues as other project, the fact that it does something this specific makes it, at first sight, unlikely to share defect with other projects. Through our manual analysis, we found out that out of the 16 *risky* commit detected by BIANCA 11 (68.75%) were matching defect introducing commit inside the Otto project itself. This is significantly higher than the average of *single project* defect (8.6%). Further investigation of the project management system led us to discover that very few issues have been submitted for this project (15) and, out of the 11 matches inside the Otto project, 7 were trying to fix the same issue that have been submitted and fixed several times instead of re-opening the original issue.

b) JStorm by Alibaba (88.96%):

For JStorm by Alibaba, our manual analysis of the *risky* commits revealed that, in addition of providing stream processes, JStorm mainly supports JSON. Unsurprisingly, the commit 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 introducing commits (516) and 18 out of the 21 commits marked as *risky* by BIANCA were so because they matched defect introducing commit in the FastJSON project.

c) Auto by Google (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%) have been because they matched commit 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.

d) Openhab by Openhab (13.33%):

Openhab by Openhab provides bus for home automation or smart homes. This is a very specific 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 defect 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 datasets (such as *HomeAutomation* a component based for smart home systems [54]) then, Openhab would not be alone in its cluster and we would have achieved a better F_1-measure.

e) Che by Eclipse (10.05%):

Eclipse che is part of the Eclipse IDE which provides development support for a wide range of programming languages such as C, C++, Java and others. Despite the facts that the Che project has a decent amount of defect introducing 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 introducing commit, we were not able to draw any conclusion on why we were not able to achieve better performances. We can only assume that Eclipse's developers are particularly careful on how they use their dependencies and the quality of their code in general. Only 2% of their commits are introducing new defects (169/7818).

f) Annotations by Excilys (3.13%):

The last project we analyzed manually is Annotations by Excilys. Very much like openhab by Openhab it provides a very particular feature: Java annotations for Android project. We do not have any other project remotely related to Java annotations or the android ecosystem at large. Consequently, BIANCA performs poorly.

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

2) Random Classifier Comparison: The relatively *low* recall is to be expected, since BIANCA only considers risky commits that are also in other projects. We built a random classifier in order to further assess the efficiency of BIANCA. Our random classifier first generate a random number n between 0 and 1 for the 165,912 commits composing our dataset. For each commit, if n is below 0.5, then, the commit is classify as non risky. If n is above 0.5, then, the commit is classified as risky. As expected by a random classifier, our implementation detected ~50% (82,384 commits) of the commits to be *risky*. It is worth mentioning is that the random classifier achieved 24.9% precision, 49.96% recall and 33.24 F_1-measure. As our data are heavily unbalanced (i.e. there is much more *healthy* than *risky* commits) these numbers are to be expected for a random classifier. Indeed, the recall is very close to 50% as one out of two commits is classified as *risky*, then 50% of the commits shall be found. While analyzing the precision, however, we can see that the data is unbalanced. We have only two classes to classify our commit into: *risky* or *healthy* and ff the data were balanced (i.e. as many *risky* and *healthy* commits), then, the precision while predicting *risky* commit with a random classifier would also be around 50%.

In order to assess the statistical difference between the precision, the recall and the F_1-measure of the random classifier compared to BIANCA with ran Mann-Whitney tests comparing the precision, the recall and the F_1-measure obtained for each project. The precision of BIANCA and the F_1-measure are significantly superior to the random classifier with a p-values $< .0001$. The recall of the random classifier is not significantly higher than the recall of BIANCA with a p-value = 0.2327.

Doing better than a random classifier is not a measure of performance per se, however, it does showcase that predicting the least represented class in an unbalanced bi-class distribution is challenging.

3) Analysis of the fixes similarity:

V. 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 BIANCA were not selected per se. Indeed, we use all the systems available and matching our experimental criterion. 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. 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).

VI. Discussion

Mathieu: THIS IS STILL A WORK IN PROGRESS. I'D LIKE TO HAVE YOUR INPUTS ON WHAT SHOULD GO HERE.

Emad: A possible discussion point to examine is how much history is needed vs. accuracy of BIANCA

Emad: Can we examine the relationship between α and accuracy.

Emad: Can we examine the relationship between the number of dependencies a project has and accuracy of BIANCA as another discussion point.

In this section, we discuss the key lessons learned while designing and validating BIANCA.

Despite the recent advances in the field, the literature shows that many existing software maintenance tools have yet to be adopted by industry [3], [4], [55–61] and, we believe that the root factors (Integration with the developer's workflow, corrective actions, cross-project knowledge, false positives) can be partly addressed by steering away from statistical model.

- It is possible to prevent defect insertion by using code changeset analysis
- It is possible to propose corrective actions after *risky* detection
- Project sharing dependencies are subjects to the same issues
- A high precision (i.e. true positives) is achievable using code analysis.

VII. Conclusion

In this paper, we presented BIANCA (Bug Insertion ANticipation by Clone Analysis at commit time) an approach that can detect risky commit (i.e. commit likely to lead to a bug report) with a 90.75% precision and a 37.15% recall. BIANCA uses code comparison of similar projects rather than statistical models built on change-level metrics. Moreover, BIANCA is able to perform its detection of risky commit before the commit actually reaches the central repository using pre-commit hooks. Finally, BIANCA is able to show the engineers the root cause behind the *riskiness* of their commit with code. Indeed, BIANCA presents engineers the defect introducing change against which the current commit has been matched and the related fixes. We believe that all these characteristics can make engineers more interested and likely to adopt bug prediction tools as we (a) integrate ourselves with the developer's workflow, (b) propose concrete corrective action and (c) leverage historical code from similar projects.

To build on this work, we need to conduct an human study with developers and engineers in order to gather their feedback on the approach. These feedbacks will help us to fine-tune the approach. Also, we want to improve BIANCA to support Type 4 clones.

VIII. Reproduction Package

We provide a reproduction package for this paper. Our reproduction package is available at http://bit.ly/bianca-icse and provides the data at different stages of processing (initial, after clustering, final result) in addition to scripts and code used.

IX. 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 *International conference on software engineering (iCSE) (2013)*, 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 *35th international conference on software engineering (iCSE)*, 2013, pp. 672–681.

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

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

[7] Joshua O'Madadhain, Danyel Fisher, Scott White, Padhraic SmythY.-b. B., "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] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] N. Moha, F. Palma, M. Nayrolles, B. Joyen-Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and Detection of SOA Antipatterns," *International Conference on Service Oriented Computing*, pp. 1–16, 2012.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering - eSEC/fSE 2015*, 2015, pp. 966–969.

[33] 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.

[34] 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, p. 1.

[35] 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*, vol. 10, pp. 311–336.

[36] 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.

[37] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *CASCON '93 proceedings of the 1993 conference of the centre for advanced studies on collaborative research: software engineering - volume 1*, 1993, pp. 171–183.

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

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

[40] U. Manber, "Finding similar files in a large file system," in *Usenix winter*, 1994, pp. 1–10.

[41] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code."

[42] R. Wettel and R. Marinescu, "Archeology of code duplication: recovering duplication chains from small duplication fragments," in *Seventh international symposium on symbolic and numeric algorithms for scientific computing (sYNASC'05)*, 2005, p. 8 pp.

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

[44] M. W. G. Cory Kapser, "Toward a Taxonomy of Clones in Source Code: A Case Study."

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

[46] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings iEEE international conference on software maintenance - 1999 (iCSM'99). 'software maintenance for business change' (cat. no.99CB36360)*, 1999, pp. 109–118.

[47] 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.

[48] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering - eSEC/fSE 2015*, 2015, pp. 966–969.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. I. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[54] 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.

[55] S. L. Foss and G. C. Murphy, "Do developers respond to code stability warnings?" pp. 162–170, Nov. 2015.

[56] 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 *First international symposium on empirical software engineering and measurement (eSEM 2007)*, 2007, pp. 176–185.

[57] 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.

[58] 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, p. 1.

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

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

[61] N. Lopez and A. van der Hoek, "The code orb," in *Proceeding of the 33rd international conference on software engineering - iCSE '11*, 2011, p. 824.