BIANCA: Prevent Bug Insertion at Commit-Time Using Dependencies Analysis and Clone Detection

Mathieu Nayrolles, Wahab 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 Analytics; Software Metrics; Risky Software Commits; Software Prediction; Clone Detection**

I. Introduction

Software maintenance activities such as debugging or feature enhancement are known to be challenging and costly [1]. Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development life cycle [2]. Much of this is attributable to several factors including the increase in software complexity, the lack of traceability between the various artifacts of the software development process, the lack of proper documentation, and the unavailability of the original developers of the systems.

Research in software maintenance has evolved over the years to include areas like mining bug repositories, bug analysis, 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. Despite the recent advances in the field, the literature shows that many existing software maintenance tools have yet to be adopted by industry [3–11]. We believe that this is caused by the following factors. (a) Integration with the developer's workflow: Most existing maintenance tools ([12–18] are some noticeable examples) are not integrated well with the workflow of software developers (i.e., coding, testing, debugging, committing). (b) Corrective actions: The outcome of these tools does not always lead to corrective actions that the developers can implement. Most of these tools return several results that are often difficult to interpret by developers. Take, for example, FindBugs [10], a popular bug detection tool. This tool detects hundreds of bug signatures and reports them using an abbreviated code such as `CO_COMPARETO_INCORRECT_FLOATING`. Using this code, developers can browse the FindBug's dictionary and find the corresponding definition *"This method compares double or float values using a pattern like this:* `val1 > val2 ? 1 : val1 < val2 ? -1 : 0`*"*. While the detection of this bug pattern is accurate, the tool does not propose any corrective actions to the developers that can help them fix the problem and developers decide simply to ignore them [19–23].
(c) Leverage of historical data: These tools do not leverage cross-project knowledge.
For defect prevention, for example, the state of the art approaches consists of adapting statistical models built for one project to another project [24], [25].
In addition, Lewis *et al.* [3] and Johnson *et al.* [8] argued that, approaches based solely on statistical models are perceived by developers as black box solutions. Developers are less likely to trust the output of these tools.

In this paper, we propose an approach named BIANCA (Bug Insertion ANticipation by Clone Analysis at commit time) that assesses these challenges by using dependencies analysis and clone detection to prevent bug insertion at commit-time. More specifically, BIANCA lies on the idea that complex softwares systems are not monolithic; they have dependencies. Softwares sharing the same dependencies are likely to perform related tasks and, therefore, share misunderstandings leading to defect introduction. For example, Apache BatchEE [26] and GraphWalker [27] both depends on JUNG (Java Universal Network/Graph Framework) [28]. BatchEE provides an implementation of the jsr-352 (Batch Applications for the Java Platform) specification [29] while GraphWalker is an open source model-based testing tool for test automation. These two softwares' functionalities are very distant from each other: (a) be able to do batch processing in Java and (b) design unit tests using graph rather than code. Nevertheless, Apache BatchEE and GraphWalker share the same dependency and, unfortunately, developers of these projects made similar mistakes while building upon JUNG. Indeed, issue Apache BatchEE #69 and issue GraphWalker #44 share a misunderstanding on how to use the graph visualization and, more specifically, the change listeners.

BIANCA integrates itself seamlessly with developers' workflow by acting at commit-time, propose concrete actions to implement to avoid bug insertion and leverage cross-project historical data without statistical models.

The remaining of this paper is organized as follows. In section II, we present works related to ours. Sections III and IV present the BIANCA approach and its validation, respectively. Then, Sections V and VI assess the threats to validity and present a conclusion accompanied with future work, respectively.

II. Related Work

Predicting crashes, faults, and bugs is very large and popular research area. The main goal behind the plethora of papers is to save on manpower—being the most expensive resource to build software—by directing their efforts to locations likely to contain a bug, fault or crash. 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. Then, we present relevant clones detection approaches as BIANCA uses such an approach to achieve bug prevention at commit-time.

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 [30] while studying Windows Server 2003.

Hassan interested himself with the entropy of change, i.e. how complex the change is [31]. 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 [32]. 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 [33]. The main difference between Hassan and Hold 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 [34]. 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 [35] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [36]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [37].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [38], El Emam *et al.* [39], Subramanyam *et al.* [40] and Gyimothy *et al.* [41] 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], [42], Demange *et al.* [43] and Palma *et al.* [44] 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.* [45], [46] and Zimmerman [47], [48] further refined metrics-based detection by using statical analysis and call-graph analysis.

While academia has published hundreds of bug prediction papers over the last decade, the developed tools and approaches fail to change developer behavior while deployed in industrial environments [3]. This is mainly due to the lack of actionable message, i.e. messages that provide concrete steps to resolve the problem at hand.

C. Clone Detection

BIANCA relies on code clone detection to perform bug prevention at commit-time. Consequently, we reviewed the literature of the field. This section describes major works in clone detection.

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of the code in a given software system [49], [50]. Developers often reuse code (and create clones) in their software on purpose [51]. Nevertheless, clones are considered a bad practice in software development since they can introduce new bugs in the code [52–54]. If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused to fix the bug in each place.

In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into three categories. The first category includes techniques that treat the source code as text and use transformation and normalization methods to compare various code fragments [55–58]. The second category includes methods that use lexical analysis, where the source code is sliced into sequences of tokens, similar to the way a compiler operates [49], [54], [59–61]. The tokens are used to compare code fragments. Finally, syntactic analysis has also been performed where the source code is converted into trees, more particularly abstract syntax tree (AST), and then the clone detection is performed using tree matching algorithms [62–65].

Text-based techniques use the code — often raw (e.g. with comments) — and compare sequences of code (blocks) to each other to identify potential clones. Johnson was perhaps the first one to use fingerprints to detect clones [55], [56]. Blocks of code are hashed, producing fingerprints that can be compared. If two blocks share the same fingerprint, they are considered as clones. Manber et al. [66] and Ducasse et al. [67] refined the fingerprint technique by using leading keywords and dot-plots.

Tree-matching and metric-based are two sub-categories of syntactic analysis for clone detection. Syntactic analysis consists of building abstract syntax trees (AST) and analyze them with a set of dedicated metrics or searching for identical sub-trees. Many approaches using AST have been published using sub-tree comparison including the work of Baxter et al. [62], Wahlert et al. [68], or more recently, the work of Jian et al. with Deckard [69]. An AST-based approach compares metrics computed on the AST, rather than the code itself, to identify clones [70], [71].

Another approach to detect clones is to use static analysis and to leverage the semantics of the program to improve the detection. These techniques rely on program dependency graphs where nodes are statements and edges are dependencies. Then, the problem of finding clones is reduced to the problem of finding identical sub-groups in the program dependency graph. Examples of recent techniques that fall into this category are the ones presented by Krinke et al.[72] and Gabel et al. [73].

Many clone detection tools have been created using a lexical approach for clone detection. Here, the code is transformed into a series of tokens. If sub-series repeat themselves, it means that a potential clone is in the code. Some popular tools that use this technique include, but not limited to, Dup [49], CCFinder [61], and CP-Miner [54].

Furthermore, a large number of taxonomies have been published in an attempt to classify clones and ease the research on clone detection [74–79]. Despite the particularities of each proposed taxonomy, researchers agree on the following classification. Type 1 clones are copy-pasted blocks of code that only differ from each other in terms of non-code artifacts 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. Finally, Type 4 are code blocks that perform the same tasks but using a completely different implementation.

BIANCA is different than the presented approach preventing the insertion of defects into the source code as it leverages cross-project historical data and is able to detect risky commit by using clone detection and dependencies analyses.

III. The BIANCA Approach

In this section, we describe our approach in details. BIANCA (Bug Insertion ANticipation by Clone Analysis at commit time) have two parts: Offline and Online. The offline part contains steps that are taken in an offline fashion (i.e., batch processing). The online part contains that are taken at the same time as a developer commits modifications to the code source. Figure 1 shows an overview of BIANCA.

During the offline part of BIANCA, we perform the following operations: (a) automatically extract dependencies from repositories to cluster them, and, (b) analyze the commits history to classify commit, link bug introducing commit to issues and extract bug changesets. In the online part, we compare new modifications to modifications known to have introduced a defect in the past using clone detection. In addition, we are able to show to the engineers why their modifications are *risky* and potentials fixes for the problem at hand.

The rest of this section is organized as follows: section III-A describes the Offline part while section III-B describes the online part.

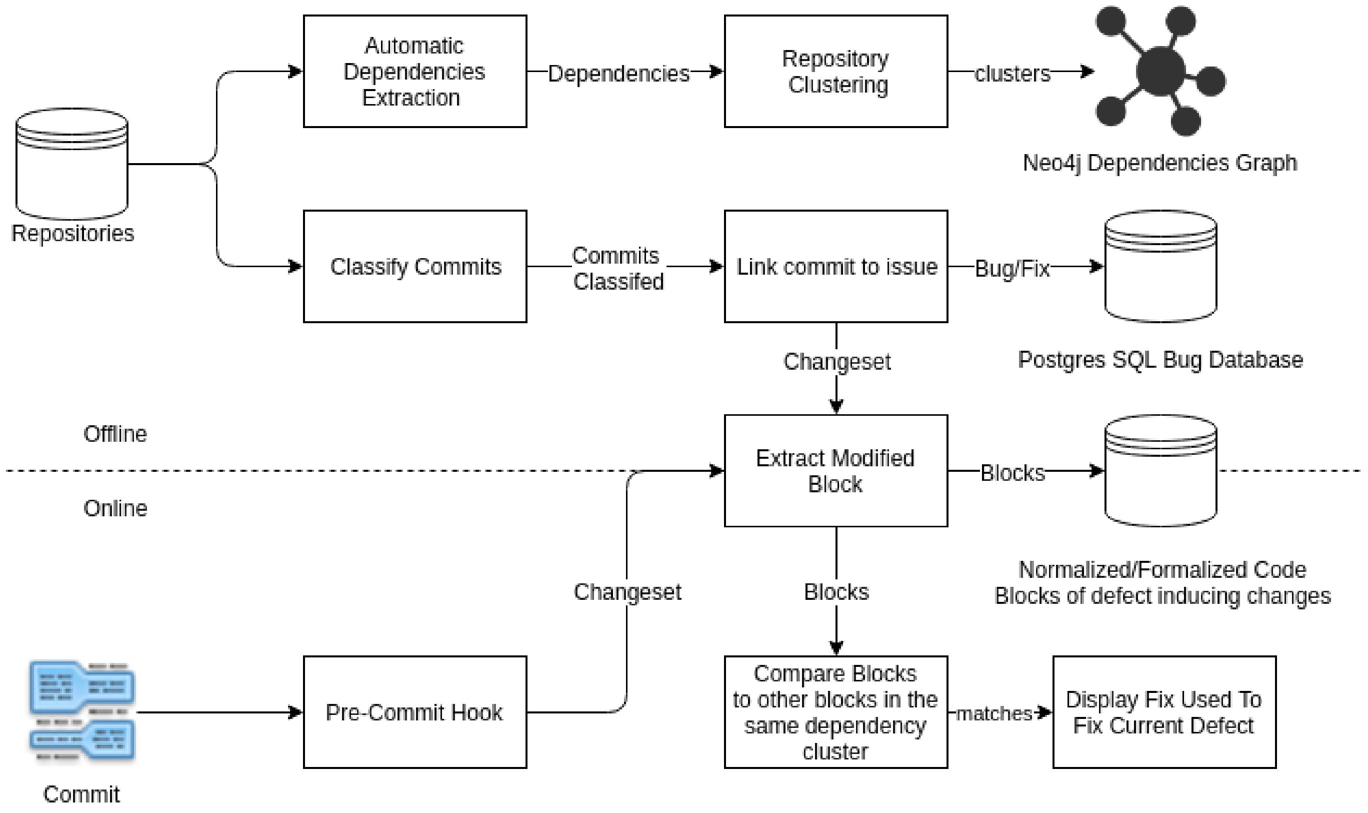

Fig. 1. Overview of the BIANCA Approach

A. *Offline*

In this section, we present the offline processes of BIANCA. Offline refers to the fact that these processes are not interactive but follow a batch processing paradigm. BIANCA first extracts dependencies of each repository (Section III-A1) and clusters repositories according to their dependencies (Section III-A2). The idea of clustering project based on their dependencies is that project sharing dependencies are likely to propose related feature, and, unfortunately, related misunderstandings about their dependencies. Then, repositories are only compared with repositories in their cluster. In our experimentations (Section IV), we demonstrate that such a clustering performs significantly better than no clustering.

In parallel of the clustering of repositories based on their dependencies, we classify defect introducing commits (Section III-A3) and extract the blocks of code contained in defect introducing commits (Section III-A4).

1) Acquiring Dependencies: In this first offline step, the dependencies of repositories are analyzed and saved into a single no-SQL graph database. Graph databases use graphs structures as a way to store and query information. In our case, each project is a node and is connected to its dependencies. Dependencies can be retrieved, in an automatic fashion, if projects use a dependency manager such as Maven. Dependency manager gathers, in one file, every dependencies of a project. Figure 2 shows a simplified zoom of a dependencies graph for a project named `com.badlogicgames.gdx`. As shown, `badlogicgames.gdx` depends on projects owned by the same organization (i.e. badlogicgames) and other organization such as Google, Apple, and Github.

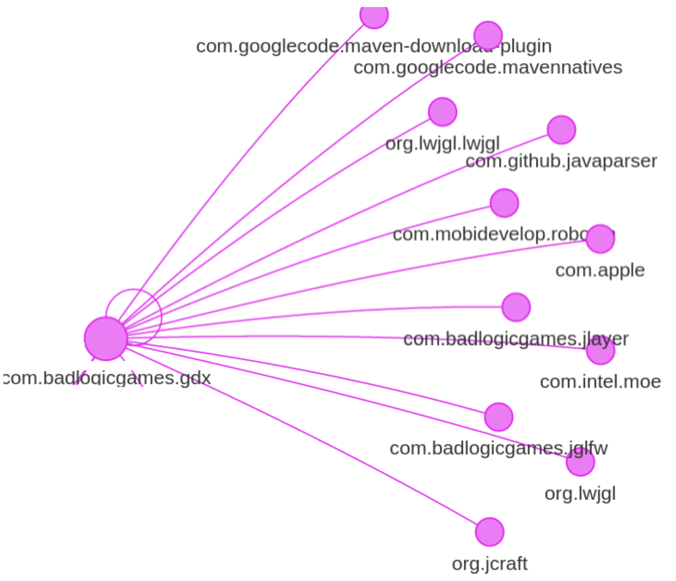

Fig. 2. Simplified Dependency Graph for `com.badlogicgames.gdx` (Zoomed from south of Figure 3)

2) Clustering Projects: The Girvan–Newman algorithm [80], [81] detects communities by progressively removing edges from the original network. The connected components of the remaining network are the communities. Instead of trying to construct a measure that tells which edges are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities. This algorithm is highly effective at discovering community structure in both computer-generated and real-world network data.

The Girvan–Newman algorithm fits our problem as we are interested in discovering the *communities* of repositories that depend on a similar set of dependencies.

3) Identifying Bug Introduction: A risky commit is a commit that introduces a bug in the future [82], [83]. For BIANCA to detect risky commits, four steps are required. The first step consists in retrieving the commits that we know have introduced defect in the past by mining software and issues repositories. Then, we can extract the blocks of code that were modified during these risky commits. The extracted blocks compose our *database* against which new commit will be compared. To compare blocks from previous commits that have introduced a defect to blocks from incoming commits, we use a text-based detection technique.

To construct our *database* of commits introducing defect, we use a modified version of the backend of commit guru [84]. Commit guru's backend have three major components: ingestion, analysis, and prediction. We use the ingestion and 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 of history is analyzed and change-level metrics proposed by Kamei *et al.* are computed [82]. Also, commits are classified using the list of keyword proposed by Hindle *et al.* [85] and reported in Table I. After this has been completed, Commit guru performs the SCM blame/annotate function on all modified lines of code for their corresponding files on the fixing commit's parent. This will return the commit that previously modified them. These commits have introduced the bugs corrected by the fixing commit and mark them as such.

Note that we could have used simpler and more established tool such as Relink [86] to link the commits to their issue and re-implement the classification by Hindle *et al.* [85] on top of it. However, Commit Guru have the advantage to be open-source. Consequently, we were able to modify it to our needs and reach satisfactory performances promptly.

4) Extract Blocks: A block is a set of consecutive lines of code that will be compared to all other blocks to identify clones. To achieve this critical part of BIANCA, we rely on TXL [87], which is a first-order functional programming over linear term rewriting, developed by Cordy et al. [87]. 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.

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

```
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* [88], 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 clone 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 is shipped with C, Java, Csharp, Python and WSDL grammars that define all the particularities of these languages, with the ability to customize these grammars 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 ← $extract\_blocks(changesets)$;
 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 arguments, the changesets and the blocks that have been previously extracted. Then, from Lines 1 to 5 lie 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 [89]. Then, we send these expanded changesets to TXL for block extraction and formalization.

In summary, this step receives the files and lines, modified by the latest changes made by the developer and produces an up to date block representation of the system at hand.

B. Online

In this section, we describe the online part of our approach. We call it *online* because it is an interactive process with the engineer committing new modification. In this part, each time a commit is made, a pre-commit hook kicks in (Section III-B1) and we extract the modified block using the technique previously presented (Section III-A4). The newly modified blocks are compared to block modification known to have introduced a defect (Section III-B2) and we recommend to apply the fixes that were used to correct the identified defect (Section III-B3).

1) Pre-Commit Hook: 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 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. Depending on the exit status of the hook, the commit will be aborted and not pushed to the central repository. Also, developers can choose to ignore the pre-hook. In Git, for example, they will need to use the command `git commit -no-verify` instead of `git commit`. This can be useful in case of an urgent need for fixing a bug where the code has to reach the central repository as quickly as possible. Developers can do things like check for code style, check for trailing white spaces (the default hook does exactly this), or check for appropriate documentation on new methods.

We believe that approaches integrating themselves at commit-time will not disturb developers' workflow.

BIANCA is a set of bash and python scripts, and the entry point of these scripts lies in a pre-commit hook. Note that even though we use Git as the main version control to present BIANCA, we believe that the techniques presented in this paper are readily applicable to other version control systems.

2) Compare Extracted Blocks: To compare the extracted blocks and detect potential clones, we can only resort to text-based techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based comparisons) would require a complete program to work, a program that compiles. In the relatively wide-range of tools and techniques that exist to detect clones by considering code as text [50], [55], [56], [66], [90], [91], we selected NICAD as the main text-based method for comparing clones [57] for several reasons. First, NICAD is built on top of TXL, which we also used in the previous step. Second, NICAD can detect Types 1, 2 and 3 software clones.

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, described in the previous section.

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

In the *Comparison* phase, extracted blocks are transformed, clustered and compared to find potential clones. Using TXL subprograms, 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 [92] 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 [67]. In addition to the pretty-printing, code can be normalized and filtered to detect different classes of clones and match user preferences.

Finally, the extracted, pretty-printed, normalized and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [93]. Then, a percentage of unique statements can be computed and, given threshold (see Section IV), the blocks are marked as clones.

3) Propose Modifications: In the previous step, we matched the new modification contained in the commit at hand with modification known to have introduced a defect in the past. The defect can be in any of the repositories belonging to the same cluster as the project at hand, including the project itself.

As we linked bug introducing changes to their commits (Section III-A3), we can propose to the engineers not only the code that *looks like* their modification and we know introduced a defect in the past but also what fixes have been deployed to eradicate it.

In summary, BIANCA does not rely on statistical models to predict risky commit but on code. We believe that this can make BIANCA a trustworthy approach in the eyes of the developers as they will know why a given modification has been reported as risky in terms of code and not fit (or lack thereof) to a statistical model. Furthermore, we propose concrete actions, in the form of code, which can be taken to reduce the risk of introducing defects to the system. Finally, the online part happens before the commit reach the central repository (Section III-B1), thus, preventing unfortunate pull of defect by other members of the organization.

IV. EXPERIMENTATIONS

In this section, we present our experimentations. We begin by explaining how we selected the repositories we analyzed (Section IV-A) and the results of the clustering algorithm (Section IV-B). Then, we present the efficiency of BIANCA in Section IV-C in terms precision, recall and F_1 measure.

A. Repository Selection

The selection of the open source repositories under analyze was made automatically according to the following criteria:

a) Java project using Maven for its dependencies.
b) Followed by at least 2000 unique developers.
c) Use a public issue repository.

These criteria ensure that (a) the dependencies of each project are retrievable in an automatic fashion and (b) that the repository has a significant interest expressed by the community. When developers follow a project, they receive notification for each new commit and issue. Finally, (c) we must be able to consult past issues and their resolution to be able to evaluate the efficiency of our approach. A high number of followers demonstrates an interest of the community. For the reader to appreciate this hard limit of 2,000 followers, on Github, 298 Java projects have at least 2,000 followers. The highest ranked repository, regarding followers, is react-native by Facebook with a total of 36,071 unique followers. React native is a framework for building native apps with the React library (a JavaScript library for building user interface, also by Facebook).

Out of the 298 Java projects with at least 2,000 followers acquired using the Github API, 21 were referencing projects that have either gone private or been deleted. On the 277 remaining projects, 42 repositories matched our criterion. This 42 repositories composed our dataset.

Our dataset is composed of 28 different ecosystems including majors open-source contributors such as Alibaba, Apache Software Foundation, Eclipse, Facebook, Google and Square.

B. Dependencies analyses

Figure 3 presents the complete dependency graph. The dependency graph is composed of 592 nodes and showcases five clusters: yellow, red, green, purple and blue. The size of the nodes are proportional to the number of connection from and to the node.

As depicted by our dependency map, the dependency landscape of the most popular Github repositories is very much interconnected and interdependent. Indeed, we have an average of 77 dependencies per projects and, as seen earlier, 592 unique dependencies. Meaning that, in average, our 42 repositories

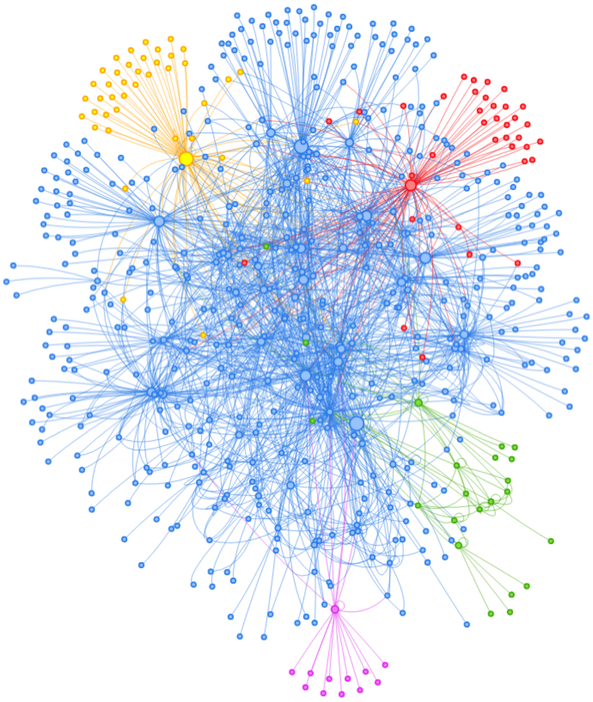

Fig. 3. Dependency Graph

share 62 of their 77 dependencies with at least one other repository. Table III presents the clusters, computed according to Section III-A2, in terms of centroids, betweenness—which is the metric used by the Girvan–Newman algorithm [80], [81] algorithm. The blue community (or cluster) is dominated by Storm from The Apache Software Foundation. Storm is a distributed realtime computation system. Druid by Alibaba, the Chinese 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 and make some of its project publicly available. The red cluster have 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 layer 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. Libdx is a cross-platform framework for game development.

We believe that these clusters are accurate as they efficiently divide repositories in terms of high-level functionalities. Indeed, we have the blue cluster which is almost entirely composed of projects from the Apache Software Foundation. Projects from the Apache Software Foundation tend to build on top one another. We also have the red cluster for Hadoop which is itself an ecosystem inside the Apache Software Foundation. Finally, we got a cluster for e-commerce applications (yellow), real-time network application for home automation (green) and game development (purple).

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

While it is difficult to assess the effectiveness of this clustering in terms of precision and recall because we do not have a ground truth, we show, in the next section, that this clustering improve our results significantly.

C. Results Analysis

In this section, we present the results of our experimentations. Table IV shows our results in terms of organization, project name, a short description of the project intent, number of class, number of commits, number of commits that introduced a defect, number of commits introducing a defect 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.

For a commit to be marked as risky (i.e., potentially introducing a defect) by BIANCA, the following conditions shall be met:

a) The commit under analysis ($commit_a$) must be a Type 3 clone of an older commit ($commit_b$) with a similarity of at least 35%.
b) $commit_a$ and $commit_b$ must belong to projects of the same cluster.
c) $commit_a$ and $commit_a$ hashes are not equal.

Type 3 clones are blocks of code that are syntactically identical except literals, identifiers, types that can be modified, indentation, whitespaces, and comments. Also, Type 3 can contain added or deleted code statements. In our opinion type 3 clones (also known as near-miss) are more adequate for BIANCA than Type 1 and 2 as they allow blocks of code to have added or deleted code statements. Type 4 clones (i.e., code blocks that perform the same tasks, but using a completely different implementation) would be an even better fit for BIANCA but (a) their detection is still a challenge [94] and (b) NICAD, our clone detection engine does not support Type 4 clone. The choice of the similarity threshold (35%) does not follow any specific indication, general law from clone detection or deeper insight into our dataset. We were only guided by the need to filter out all spurious matches while still keeping enough clones to represent the dataset. Thus, we have made several incremental attempts, starting from 10%. For each attempt, we incremented the value by 5% and observed the obtained precision and recall. The current value seems to offer the best trade-off.

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**

Overall, we achieve a precision of 90.75% (13899/15316) commits identified as risky by BIANCA are true positives. They did trigger the opening of an issue and had to be fixed later on. BIANCA reaches 37.15% for the recall measure (15316/41225) and 52.72% for the F_1 measure. Consequently, we can validate our assumption that project sharing dependencies are open to the same flaws. Moreover, detecting these common defects allow us to reach high precision. Obviously, our recall is somewhat low (37.15%) as we can only catch risky commit with common root causes.

It is important to note that we do not claim that 37.15% of open-source systems issues are directly linked to their dependencies. To support such a claim, the 15,316 detected commits would have to be analyzed manually. However, we have proven that complex software systems sharing dependencies also share common issues, directly related to their dependencies or not.

Finally, we can evaluate the efficiency of our clustering by analyzing BIANCA's result with and without it. First of all, the computation time per commit without clustering is 182% higher. Indeed, it goes from 32.2 seconds, in average, to 58.6 seconds (with an i5@1.8Mhz, 19.6 GiB of RAM and SSD disks on Debian 8). Also, without the clustering, the recall metric reaches 42.4%, which is higher than our current 37.15%. However, when compared by projects with a Wilcox test, the difference is not statistically significant (i.e. p-value > 0.05). On the precision side, the metric drops significantly to 71.6%

(i.e. p-value < 0.05).

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. 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.

VII. 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.

VIII. References

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, 2005, p. 880.

[2] Health, Social and E. Research, "The Economic Impacts of Inadequate Infrastructure for Software Testing," 2002.

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

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

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

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

[8] 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.

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

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

[11] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[24] 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.

[25] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *2013 35th international conference on software engineering (iCSE)*, 2013, pp. 382–391.

[26] The Apache Software Foundation, “Apache BatchEE.” 2015.

[27] Graphwalker, “GraphWalker for testers.” 2016.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

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

[44] F. Palma, “Detection of SOA Antipatterns,” PhD thesis, Ecole Polytechnique de Montreal, 2013.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] B. Baker, “On finding duplication and near-duplication in large software systems,” in *Proceedings of 2nd working conference on reverse engineering*, pp. 86–95.

[50] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code.”

[51] M. Kim, V. Sazawal, and D. Notkin, “An empirical study of code clone genealogies,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 187, Sep. 2005.

[52] C. Kapser and M. Godfrey, “Cloning Considered Harmful Considered Harmful,” in *2006 13th working conference on reverse engineering*, 2006, pp. 19–28.

[53] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?” in *2009 iEEE 31st international conference on software engineering*, 2009, pp. 485–495.

[54] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: finding copy-paste and related bugs in large-scale software code,” *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, Mar. 2006.

[55] 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.

[56] 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.

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

[58] 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.

[59] B. S. Baker, “A program for identifying duplicated code,” *Computing Science and Statistics*, 1992.

[60] B. S. Baker and R. Giancarlo, “Sparse Dynamic Programming for Longest Common Subsequence from Fragments,” *Journal of Algorithms*, vol. 42, no. 2, pp. 231–254, Feb. 2002.

[61] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[62] I. D. Baxter, A. Yahin, L. Moura, Sant'AnnaM., and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the iEEE international conference on software maintenance.*, 1998, pp. 368–377.

[63] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proceedings of the 27th aCM sIGPLAN-sIGACT symposium on principles of programming languages - pOPL '00*, 2000, pp. 155–169.

[64] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proceedings of the 44th annual southeast regional conference on - aCM-sE 44*, 2006, p. 679.

[65] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, Jul. 2008.

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

[67] 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.

[68] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Source code analysis and manipulation, fourth iEEE international workshop on*, pp. 128–135.

[69] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *29th international conference on software engineering*, 2007, pp. 96–105.

[70] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague, "Extending software quality assessment techniques to Java systems," in *Proceedings seventh international workshop on program comprehension*, 1999, pp. 49–56.

[71] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," in *Sixth working conference on reverse engineering (cat. no.pR00303)*, pp. 326–336.

[72] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Eighth iEEE working conference on reverse engineering, 2001*, 2001, pp. 301–309.

[73] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 13th international conference on software engineering - iCSE '08*, 2008, p. 321.

[74] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of international conference on software maintenance iCSM-96*, 1996, pp. 244–253.

[75] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proceedings sixth international software metrics symposium (cat. no.pR00403)*, 1999, pp. 292–303.

[76] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," in *2006 13th working conference on reverse engineering*, 2006, pp. 253–262.

[77] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[78] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *Proceedings of the fourth working conference on reverse engineering*, pp. 44–54.

[79] C. Kapser and M. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings. 7th international workshop on principles of software evolution, 2004.*, pp. 85–94.

[80] 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.

[81] 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.

[82] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

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

[93] 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.

[94] E. Duala-Ekoko and M. P. Robillard, "Tracking Code Clones in Evolving Software," in *29th international conference on software engineering (iCSE'07)*, 2007, pp. 158–167.