Clones: What *is* that Smell?

Foyzur Rahman, Christian Bird, Premkumar Devanbu
Department of Computer Science
University of California, Davis
Davis, California, USA
{mfrahman,cabird,ptdevanbu}@ucdavis.edu

***Abstract*—Clones are generally considered bad programming practice in software engineering folklore. They are identified as a *bad smell* and a major contributor to project maintenance difficulties. Clones inherently cause code bloat, thus increasing project size and maintenance costs. In this work, we try to validate the conventional wisdom empirically to see whether cloning makes code more defect prone.**

This paper analyses relationship between cloning and defect proneness. We find that, first, the great majority of bugs are not significantly associated with clones. Second, we find that clones may be *less* defect prone than non-cloned code. Finally, we find little evidence that clones with more copies are actually more error prone. Our findings do not support the claim that clones are really a "bad smell". Perhaps we can clone, *and* breathe easy, at the same time.

***Keywords*-software clone; empirical software engineering; software maintenance; software evolution;**

I. Introduction

The software life cycle comprises two major parts; first we define the specification and implement it; then, we need to maintain the finished product and evolve it to better suit user needs. For most other industries, development cost is the major factor in a project's lifetime cost. However for software development it has been found that maintenance and evolution are also critical activities from the cost perspective and might comprise upto 80% of the overall cost and effort [1]. Researchers have long sought to ameliorate maintenance costs. There have been quite a bit of work on improving process models, tool support, language support, *etc.,* to improve development process and reduce bad attributes of code which might negatively impact maintenance cost. Often, however, poor maintainability can be traced back to poor code which is difficult to understand, modify or more error prone. For a taxonomy of bad code attributes refer to [2], [3].

Fowler [2] suggests that code duplication or cloning is a *bad smell* and thus one of the major indicators of poor maintainability. Cloning is an easy, tempting alternative to the hard work of actually refactoring the code. Unfortunately, if a piece of code is buggy or has a latent bug, then a clone can replicate a bug silently. To aggravate the situation, cloning is often performed hastily and without much care about the context. This could mean that even bug-free code could become buggy after cloning [4]. Furthermore, developers often copy others' code without fully understanding it. This introduces another classic fault proneness through poorly understood code. For these reasons, clones have been vilified for many years and a considerable body of research work has been devoted to automatically find clones; some even try to automatically refactor them [5], [6], [7].

At the same time, another body of research presents evidence that clones improve productivity and they may not be as bad as some claim. Kim et al. [8] argued that aggressive refactoring is not worth the effort, as most clones are short lived. Also, they suggested that long lived clones may not be refactorable due to language limitations. Kasper and Godfrey [9] presented evidence that clones are made deliberately and improves developer productivity. Thummalapenta *et al* assert that developers are actually quite capable of remembering and updating clones consistently whenever required, even when they reside in very different parts of the system [10]. However, prior research has not tried to establish a direct relation between end product quality and cloning. We take the view that product quality is a major barometer of product success and if clones have much impact on product quality, we claim it a serious disadvantage for cloning proponents. One good approximation of product quality is the number of defects found in the product. More defects could make a system unusable and make its users unhappy. In this paper we try to assess clones' impact on defect occurrence of software products.

Considering the entire population of bugs, it would be interesting to determine how many of these are associated with cloned content. Do clones contribute a very small proportion of bugs, or the vast majority? This gives us an indication of how important clones are in overall project quality.

RQ1: To what extent does cloned code contribute to bugs?

Next, we examine the converse question. Considering the code implicated in defect repair ("buggy code"). Are clones unduly over-represented in this code? If buggy code contains a lot of clones, then this suggests that we'd do well to refactor out clones, or at least inspect all the clone code.

RQ2: Do clones occur more often in buggy code than

elsewhere?

Finally, we'd like to know whether clones with many copies ("prolific clones") are worse than clones with fewer copies ("non-prolific clones"). One can easily imagine that as copies proliferate, it is likely that the chance of accidentally introducing errors will increase.

RQ3: Are prolific clone groups more buggy than non-prolific clone groups?

We try to answer our questions empirically by analyzing four major open source projects, namely: Apache, Evolution, Gimp and Nautilus. Our study suggests that clones occur less often in bugs than overall code. In all projects, we found that most bugs have nothing at all to do with cloned code (RQ1). Furthermore, we found buggy code is less likely to have cloned code, when compared to the project overall (RQ2). Finally, we found *no evidence* to support the claim that prolific clones have more buggy code than the non-prolific ones.

Our results might encourage researchers to put more effort on automatic clone maintenance than refactoring and eliminating them, so that when consistent change is required, a developer could be pointed to all the available clone fragments. In addition, and rather surprisingly, one might well conclude that bug-prediction tools could use cloned content as a *negative* indicator of defect-proneness!

II. Related Works

A. Clone Evolution

Several studies have investigated the extent and evolution of cloning in different software projects. These studies report between 5 to 50% of the source code being cloned [11], [12]. Kim et al. [8] have investigated evolution of clones and built a clone genealogy. Their findings indicate that most of the clones are short lived, and therefore over-aggressive refactoring may be overkill. They also found that the long-lived clones diverge so much, that they can no longer be refactored with existing language support. Geiger et al. [13] examined whether clones in different files induce change coupling. Kim *et al*. [14] have studied the copy-paste behavior of programmers and have proposed a taxonomy of clones in their paper. Kapser *et al*. [9] proposed a categorization of patterns of clones, and analyzed the motivation, maintenance impact, advantage, disadvantage, structural manifestation of the patterns. They conclude that cloning is a reasonable design decision and tools should be developed with long term maintenance of duplicates in mind. Krinke [15] studied consistent and inconsistent changes to clones and found that only 50% of the clone groups underwent consistent changes; once made inconsistent, the groups remained inconsistent. Krinke studied cloned code stability [16] where he concluded that cloned code is more stable than non-cloned code.

B. Tool Support

There has been quite a bit of research on tools for clone maintenance. Ekoko et al. [17] proposed a tool for tracking clones in evolving software. Their tool supports simultaneous editing of clones, along with notification to developer when one of the clones changes. A clone tracking tool could reduce possible bug inducing inconsistent changes while allowing developers greater latitude. Bruntink et al. [18] proposed automatic aspect mining based on clone detection. SHINOBI [19] tries to identify clones in real time and is integrated with Microsoft Visual Studio to aid maintenance. Clever [20] integrates with SVN to facilitate better management of clones. Toomim et al. [21] suggested linked editing to edit multiple regions without much programmer intervention.

C. Clones and Bugs

Researchers have studied the effect of clones on software quality. Juergens et al. [22] studied inconsistent clones as detected by their tool. They used manual annotations by developers to determine faults in inconsistent clones, and concluded that unintentionally made inconsistent clones are more likely to contain defects. Statistical tests of significance are not presented. As described below, our approach relies on data mined from bug repositories, rather than manual annotation. Jiang et al. [4] proposed an approach on detecting clone related bugs based on context. Their approach tries to detect similar sections of clones, and then based on their contextual difference suggests whether a possible bug is lurking. Thummalapenta et al. studied clone maintenance [10] and their evolution pattern. They found that clones were consistently propagated when needed and developers actually seem to remember the clone locations that require such propagation. They also found cloning often used as a templating mechanism. They found that clone characteristics such as clone granularity or clone radius have little impact on clone evolution. As a whole their study views clones positively; they argue that while better tool support for clone maintenance would help, aggressive refactoring out of clones was probably not worthwhile. Sliwerski et al. [23] studied source code changes that induce fixes. Their approach of determining fix-inducing-change is similar to our buggy code determination approach. However, instead of finding the origination of a buggy code, we map the buggy code to some intermediate snapshot and analyze its properties at that point in time.

III. Terminologies

In this section we will define all the terminologies and background of our experiment.

A. Snapshot and Revision

Source code management systems (SCM) typically provide a rich version history of software projects. This information includes file history, such as when a file was

added/removed/modified; author history, such as who wrote a particular line in a file; commit history, such as when a file was committed; commit log, such as what is the contribution of a commit etc. In our study we identify each of these commits as a revision, where a revision $r = \langle A, T, f_1, f_2, \ldots, f_n \rangle$. Here A is the author of the revision who modified a set of files $\{f_i\}$ and committed the revision at time T. Our study examines the impact of cloning throughout the project life cycle, and thus must find clones in all the revisions committed into the SCM. Checking for clones on every revision of every file is not feasible. Instead we run clone detection only once a month from project inception to the end of available project history. We call each of these chosen monthly revisions a "snapshot". So, we have a collection of snapshots $S = \langle s_1, s_2, \ldots, s_n \rangle$, where s_i is the *first* revision committed in month i, i.e. $s_i = r_{i1}$, where $\langle r_{i1}, r_{i2}, \ldots, r_{im} \rangle$ are the revisions committed in month i. Note that our months may not coincide with calender months; we start monthly epochs from the first revision date of a project. For each such snapshot, we check out all the files extant at that time in the project history and run clone detection on them.

We used git for our repository; for speed, we migrated other repositories (SVN and CVS) to git.

B. Finding Clones

In this paper, the term "clone" refers to a code clone, i.e. similar fragments of code sections, as output by the clone detector. A clone detector's output O typically consists of a set of clone groups; $O = \{g_1, g_2, \ldots g_n\}$, where each of the groups g_i contains a set of code sections that are similar to each other, i.e. clone group $g_i = \{c_1, c_2, \ldots, c_n\}$, and each of the clones are defined as $c_i = \langle s_j, f_k, l_s, l_e \rangle$. Here s_j refers to the snapshot in which this clone was found, f_k refers to the file that contains clone c_i and l_s and l_e indicates start and end line number.

We detect clones on all the snapshots s_i. For each of the snapshots, we ran DECKARD [24] on that snapshot to get all the clone information. From DECKARD output, we extract filename, line number, which clone a line belongs to and the sibling clones. For our study, we ran DECKARD with a *conservative* and a *liberal* clone detection parameter setting. This is to reduce study bias towards a particular clone detector parameter setting and to understand system behavior as the clones become more dissimilar. For the conservative mode, we set minimum token parameter for DECKARD to 50 (clones must be at least 50 tokens in length) and similarity to 1.0 (clones must be nearly identical). In liberal parameter setting, we set minimum token to 50 and similarity to 0.99 (to allow greater divergence). In both cases we set DECKARD stride to 2. We also experimented with several other parameter settings such as <50, 1.0, 16> and <50, 0.95, 4>, <50, 1.0, Infinity> and <30, 0.95, Infinity> where they are represented as <Min Token, Similarity, Stride>, and found similar results. We chose DECKARD as it is previously [24] shown to be a very scalable, and finds more clones than CCFinder or CP-Miner with few false positives.

We call the cardinality of the clone group g_i as its order. So, $Order_i = |g_i|$. We also partition clone groups into two sets: *prolific* clone groups, with more than 3 members and *non-prolific* clone groups, with up to 3 members.

C. Copy and Unique

For this study, we flatten all the clones detected by DECKARD and consider them at individual line level. So, for each of the line in any of the file f_i, of snapshot s, if that line is part of any of the detected clones by DECKARD, we call that a *copy*, otherwise it is called *unique*. Here to note that: a single line may occasionally appear in multiple clones but we declare a line as copy whether it appears in one clone or many.

D. Bug Fixing History

We focus on bugs which were discovered and recorded in the project's issue tracking history, for example Bugzilla. Typically bugs are discovered and recorded in an issue tracking system such as Bugzilla and later on fixed by the developers. We consider any change associated with a report in the Bugzilla database as a bug. However, Bugzilla has the "enhancement" type entry which does not designate a bug. We excluded any such entry in our observation. We define bug as $B = \langle OD, FD, D \rangle$, where OD represents date when a bug was opened, FD is the date when the bug was fixed and marked in the system as fixed and D is the description of the bug.

We link a fixed bug from issue tracker to a particular revision in the SCM. We call this a bug fixing revision. We identify a bug fixing revision based on several different heuristics. Various key words such as "bug", "fixed" etc. in the SCM commit log typically indicates a bug fixing revision [25]. Also, a numerical bug id is typically mentioned in a bug fixing commit log, which can then be linked back to issue tracking system's issue identifier [26], [27]. We also crosscheck with the issue tracking system to see whether such issue identifier exists and whether its status changes after fixing the bug. Finally we use manual inspection to remove spurious linking as much as possible. Our approach uses Bachmann's linking heuristics; in fact, we gratefully acknowledge the direct use of data derived by Bachmann [28].

E. Buggy Code

In an ideal situation, a set of source code lines that introduced a bug can be defined as buggy code. However, it is very difficult to precisely find the culpable code, so we approximated the notion of buggy code. In this paper buggy code refers to a set of source code lines which were

modified to fix a bug. So, buggy code for i-th bug fixed in revision r: $BC_i = \{L_{f,j}\}$ where $L_{f,j}$ is the j-th changed line in file f for fixing that bug (note: changed lines in a file may not be contiguous and buggy code for a single bug can span multiple files).

To determine buggy code, we first identify a revision which fixes a bug. If a bug is fixed in revision r we take the immediate preceding revision $r-1$ and then we identify all the files that were changed in revision r. We then find the lines changed in each of these files. $\{L_{f,j}\} = diff(f_r, f_{r-1})$. Where diff is traditional Unix diff tool and f_r is the version of the file f at revision r. For all changed files f the set of changed lines $\{L_{f,j}\}$ comprises our buggy code for i-th bug. Note: we ignore any newly introduced lines at revision r as they, by definition, could not be the cause of original bug.

F. Bug Staging Snapshot

Each of the bugs is associated with its closest preceding snapshot which is called its staging snapshot (ss_b). So, if a bug b is fixed in revision r and revision $r-1$ (the last revision prior to fixing that bug) occurs in month i of the project history, then i-th snapshot is its staging snapshot. The staging snapshot is where buggy code for a bug is analyzed. This is necessary because we do not have clone information available for some arbitrary revisions other than the chosen snapshots.

Due to possible intervening changes to buggy files between ss_b and $r-1$, each of the buggy lines in a buggy code at revision $r-1$ may have different line number at its staging snapshot. But for our purposes we need the older line number at ss_b instead of the newer line number at $r-1$. To map a line at l_{r-1} to l_{ss}, we used Unix diff utility to find all the changes made to that file during this time period. So, if n lines were added and m lines were deleted on top of a given line number l_{r-1} between releases ss_b and $r-1$, we adjust the overall difference to find l_{ss}. Also, if l_{r-1} was newly added some time after revision ss_b (i.e. l_{r-1} was nonexistent in ss_b), then we ignore that line.

G. Buggy Cloned Code and Bug Clone Ratio

Each of the lines in a buggy code fragment can be classified as either a copy or unique, based on whether that line is part of any of the clones recognized by DECKARD. We called the copied lines of buggy code *buggy cloned code*. We then calculate the ratio of such copied code in the buggy code, which we call *bug clone ratio*. Note, to determine any such partitioning of buggy code, we first mapped all the buggy codes to its staging snapshot and then determined intersection between buggy code and copied lines of that snapshot.

IV. Experimental Methods

We chose 4 different medium- to large-sized open-source projects for our study. All have long development history, but hail from different domains. All of our projects are written in C. We summarize our projects below.

1) **Apache httpd** – Apache httpd is a widely used open source web server. We converted the repository from SVN to git for ease of use.
2) **Nautilus** – Nautilus is the default file manager for the Gnome desktop. We were able to use their git repository directly.
3) **Evolution** – Evolution is the default email client for the Gnome desktop with support for integrated mail, address book and calender functionality. We used their git repository directly.
4) **Gimp** – Gimp is most popular open source image manipulation program. We used their git repository directly.

A summary of descriptive statistics of the projects studied is presented in table I. They range in size from 124K lines to about 755k lines. The table presents quite a bit of details about the number of snapshots, and average (computed over all snapshots) statistics on the average total number of clone lines, number of members (clone) per clone group, clone size in lines, number of cloned lines per snapshot, and total number of linked bugs (over the entire period).

For all the projects, we first identify monthly snapshots and then run DECKARD to detect clones in those snapshots. We tag each of the lines of a snapshot as either a copy or a unique line. We then identify all the bug fix revisions. Buggy code is then identified by running diff on the bug fix revision and its immediately preceding revision. We then map those buggy lines to their corresponding staging snapshots. A simple set intersection is performed to classify each of the buggy lines as either copy or unique. We then find the buggy cloned code and calculate the clone ratio in the bugs. We stored all of our information in a PostgreSQL database before processing them.

In one specific Apache snapshot, we found abnormal (4 fold) increase of source code line count and a corresponding spike in the clone ratio. We believe this was due to some accidental copying of major project elements, and we therefore ignored that snapshot. All the bugs that have that snapshot as their staging snapshot, were mapped back to the immediate preceding snapshot.

Our experimental approaches to the research questions are detailed below:

RQ1 To what extent does cloned code contribute to bugs? For each bug in the project, we consider how much cloned code contributes to that bug, *viz.,* its bug clone ratio. Now we can consider cumulative bug clone ratio distribution for all the bugs in a given project. So for example, if the cumulative distribution indicates that most of the bugs have a *clone ratio* (defined earlier, in III-G above) between 80% and 100%, we can conclude that clones contribute heavily to bugs; alternatively, if most of the bugs have 1% or lower

Name	Max size	Total snapshots	Lines per snapshot	Cloned lines per snapshot (conservative)	Clones per group (conser.)	Lines per clone (conser.)	Cloned lines per snapshot (liberal)	Clones per group (liberal)	Lines per clone (liberal)	Number of linked bugs
Apache	208388	155	124462.62	13817.02	3.24	14.79	16611.14	3.25	14.76	453
Evolution	531342	129	324487.14	26322.54	2.49	15.27	33011.09	2.56	15.34	1440
Gimp	947073	130	755511.68	167160.73	3.38	22.08	176090.99	3.45	22.04	2103
Nautilus	366894	116	131062.94	14878.97	2.20	18.13	17495.76	2.24	17.85	747

Table I
SUMMARY OF STUDIED SYSTEMS

clone ratio, then we know that clones contribute almost no bugs.

RQ2 Do clones occur more often in buggy code than elsewhere? We compare all the bugs' clone ratio (proportion of cloned code, in all bugs, taken together) against the overall clone ratio in the project at the time that bug is fixed. So, if a bug is fixed at the r-th revision, and x% of the total code of the project, in the $(r-1)$-th revision, was from clones, we ask if the buggy code in that revision has a bigger or smaller proportion of cloned code, compared to the overall project code. Since we do not have clone information for all possible revisions, we just project each line number back in the history to its staging snapshot and see whether a line is a clone or not. We then compare the staging snapshot's clone ratio against all the bugs' combined clone ratio that pertain to that staging snapshot. So, if a staging snapshot ss_b has n different bugs, which include a combined total m lines, of which c total lines are contributed by clones, we compare $\frac{c}{m}$ against clone ratio of ss_b. We consider two samples: each staging snapshots' clone ratio and the corresponding coalesced clone ratio for all the bugs attributed to that snapshot. We then compare them visually using boxplots, and test if they are drawn from the same distribution (null hypothesis) using a *paired* Wilcoxon test. The null hypothesis is, both of these distributions should be same. Note: in some cases, there may not be any bug projected to a particular snapshot and we ignore that snapshot as that is not a staging snapshot for any bug.

RQ3 Are prolific clone groups more buggy than non-prolific clone groups? We compare prolific clone groups' bugginess with non-prolific clone groups' bugginess. We define defect density as the fraction of cloned lines of that group that contribute to a bug. Assuming that bugs will proliferate as clone copies proliferate, we can assume that the defect density $\frac{buggy\ lines}{total\ lines}$ will not change much. As there are many more clone groups which do not contribute any buggy code (Since the total volume of buggy codes mapped to a staging snapshot are a tiny fraction of overall project code, and thereby it is more likely that many clone groups include cloned lines that actually do not contribute to any buggy code), we only consider those clone groups which contribute at least one line in some buggy code. Also, by normalizing contributed buggy cloned lines for number of lines in that clone group we control for the disparity of total cloned lines contributed by clone groups of different size.

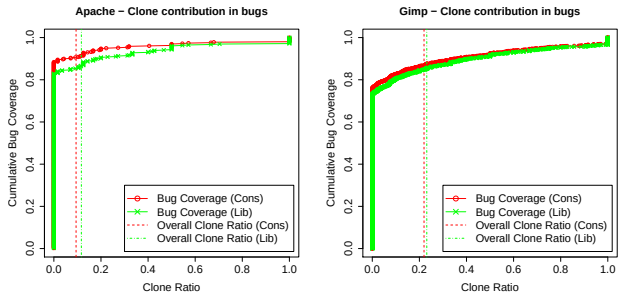

Figure 1. Cumulative coverage of bugs at a given clone ratio (a), Apache (b) Gimp.

Name	p-value (Conservative)	p-value (Liberal)
Apache	3.777e-05	3.3e-04
Evolution	1.291-04	9.4e-05
Gimp	6.482e-06	2.2e-05
Nautilus	3.394e-06	1.1e-03

Table II
WILCOXON PAIRED TEST WITH ALTERNATIVE HYPOTHESIS SET TO "SNAPSHOT CLONE RATIO > BUG CLONE RATIO". ALL P-VALUES HAVE BEEN ADJUSTED USING THE BENJAMINI-HOCHBERG PROCEDURE.

V. RESULTS

A. Findings

RQ1 To what extent does cloned code contribute to bugs? Figure 1 shows the cumulative bug coverage at different clone ratios. Due to space constraints, we only show Apache and Gimp, which are representative. The plot shows the fraction of bugs that have a clone ratio <= a given clone ratio. So, if b bugs have a clone ratio <= r, and there are total t bugs, then the plot shows $\frac{b}{t}$ on the Y axis against r on the X axis. Alternatively we can say that a $1-\frac{b}{r}$ bugs portion of bugs have higher clone ratio than Y. As is evident from the plot, most of the bugs in both liberal and conservative clone detector settings contain hardly any cloned code. In fact besides Gimp, 80% or more bugs in the other projects contain no cloned code at all. Even for Gimp, this threshold is close to 80%. The vertical lines depict the average clone ratio across all snapshots for different clone detector settings. So, e.g., we can say that for Gimp about 85% of bugs have lower clone ratio than overall project clone ratio. This finding suggests that only a small number of bugs are attributable to cloning.

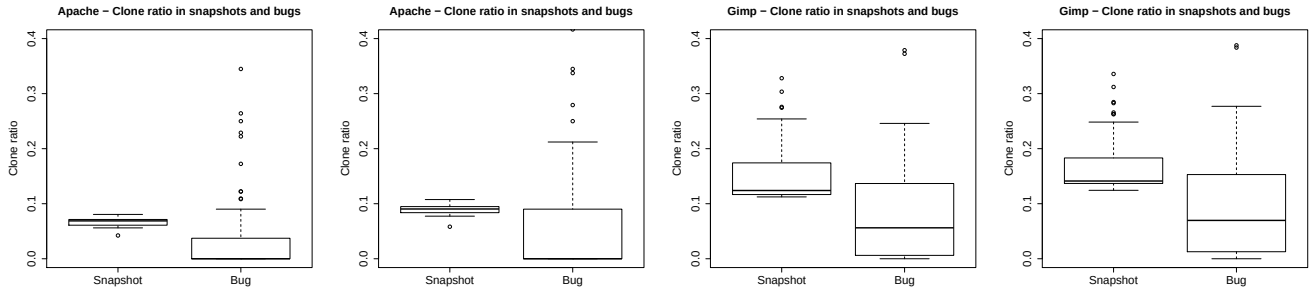

Figure 2. Clone ratio in bugs and snapshots for (a) Apache (Conservative) (b) Apache (Liberal) (c) Gimp (Conservative) (d) Gimp (Liberal).

Name	p-value (Conservative)	p-value (Liberal)
Apache	6.666e-03	4.850e-12
Evolution	1.418e-05	4.400e-16
Gimp	8.800e-16	4.400e-16
Nautilus	1.000e-02	8.900e-03

Table III
WILCOXON TEST WITH ALTERNATIVE HYPOTHESIS SET TO "DEFECT DENSITY IN NON-PROLIFIC GROUP > DEFECT DENSITY IN PROLIFIC GROUP". ALL P-VALUES HAVE BEEN ADJUSTED USING THE BENJAMINI-HOCHBERG PROCEDURE.

RQ2 Do clones occur more often in buggy code than elsewhere? Figure 2 shows boxplots of clone ratio in staging snapshots and corresponding clone ratio in bugs that were fixed in those staging snapshots. For all the projects, the boxplots clearly indicate a lower clone ratio in buggy code. For Apache with a conservative clone detector settings, the difference between the two boxplots is dramatic. Even with a liberal clone detector settings, the median of bug clone ratio is well below the median of snapshot clone ratio. This phenomenon is repeated in all the other projects. The non-parametric *paired* Wilcoxon rank sum test (with continuity correction) in all cases conclusively rejects the null hypothesis that the two samples (clone ratios in buggy code and clone ratios in the entire snapshot) are drawn from the same distribution. Corresponding p-values after Benjamini-Hochberg adjustment are presented in Table II. As we mentioned earlier, we also experimented with several other clone detector parameter settings. We found that as the similarity value is decreased and set to a very low value, such as 0.95 along with smaller token size, such as 30, clone ratio in bugs increases and the gap in median with the background distribution closes. However a Wilcoxon rank sum test shows that the overall clone ratio remains significantly lower than clone ratio in buggy code. These robust statistical results, across all 4 projects, suggest that clones are not really a major source of bugs.

RQ3 Are prolific clone groups more buggy than non-prolific clone groups? We compare defect density (number of buggy cloned lines per line of cloned code) in lines which are part of prolific clone groups against lines which are part of non-prolific clone groups. One might expect that by dint of sheer size, prolific clone groups, with more code, and with more copying, will be associated with more defects than non-prolific clone groups. As the copies proliferate, the defects will replicate in the copies, and thus we can expect that the defect density will remain a constant. Figure 3 depicts our findings for Apache and Gimp. Rest of the projects are very similar and thereby we omitted them for brevity. Note that, the bug density may occasionally go above 1.0. This is because of clone group with contribution to buggy code of multiple bugs, thereby making numerator (number of buggy lines) greater than denominator (total number of lines in clone group). We find that in fact, prolific clone group has lower defect density than non-prolific clone group. Table III shows adjusted p-values (using Benjamini-Hochberg method) of Wilcoxon one-sided rank sum test with continuity correction. The alternative hypothesis is set to "defect density in non-prolific group is greater than defect density of prolific group". All the p-values are highly significant; thus we reject null hypothesis. Clearly, there is a strong signal in all the projects, that more copies doesn't at all mean more defects: in fact, the more copies, the *lower* the defect density.

We hasten to point out that others, for example [8], [9], [16], [10] have argued that the fear of clones is perhaps overstated. To our knowledge however, this is the first study to use data mined from version-control repositories and reported bug-fixes to provide quantitative evidence that clones are not necessarily to be feared. Also, to our knowledge, ours is the first study to indicate that larger clone groups are different from smaller clone groups with respect to defect attribution.

However, there could be another possible explanation of the observed phenomenon in RQ3. Prolific clone groups by definition have many members. A developer may fix the same bug in multiple copies, but do so in multiple commits; he may not identify every commit as a fix of a bug and/or present the bug id in the commit log. In such situations, our linking algorithm may miss some of the delayed fixes altogether. This will deflate bug density in prolific clone groups and poses a significant threat to RQ3 findings. However, Thummalapenta et al. [10] found that

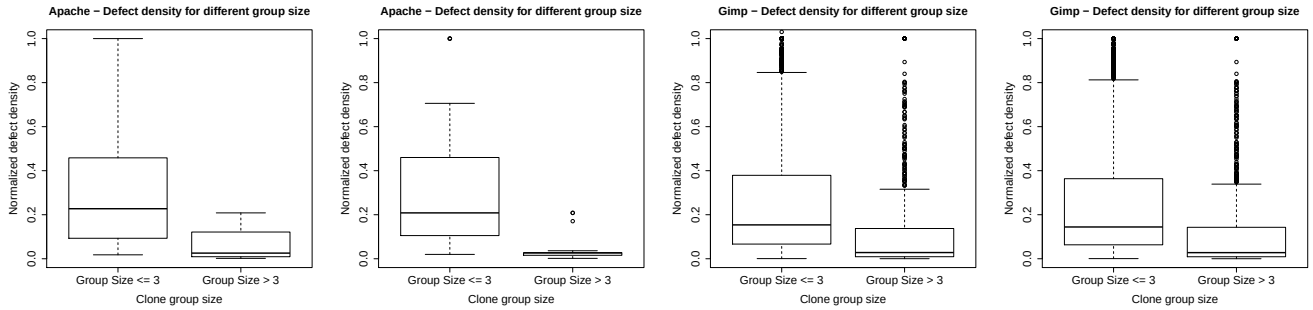

Figure 3. Defect density in clone groups of different sizes for different projects (a) Apache (Conservative) (b) Apache (Liberal) (c) Gimp (Conservative) (d) Gimp (Liberal).

developers are able to remember location of clone copies and propagate changes consistently. In only a small percentage of cases, usually less than 16% they actually underwent late propagation.

We however want to stress that, the above mentioned threat to validity does not affect our findings in RQ1 and RQ2. In RQ1, we consider cloned code in buggy code, which is immune to above mentioned bug linking problem. Unless there is any systematic bias in bug linking which only links non cloned bugs while leaving out others, our result is robust and statistically sound. Even if only one copy is linked with a bug, we adjust both numerator and denominator when calculating clone ratio. In RQ2 we again work with clone ratio which is robust against the mentioned linking problem. We ignore bugs that are not linked and consider clone ratio in linked bugs. As long as there is no systematic bias in linking process to leave out bugs that have cloned code in them, our results of RQ1 is also robust and statistically sound.

B. Case Study

To gain further insights as to why clones appear less buggy, we did a case study of 20 good quality (has very few bugs) clones (3 from conservative and 2 from liberal for each of the 4 projects). In Listing 1, we show one very good quality (no buggy code) clone which comes from a group of 2 clone members. Both of the members come from the file "libnautilus-private/nautilus-file.c" in a snapshot taken on 20th November, 2000. This code tries to set a file's owner and before doing that it checks to see whether the user has required privileges or whether the user is same as the current file owner. If everything goes well, then the code proceeds to change the owner of the file. A very similar role of a file manager is to change the group of the file. Another clone from the same group achieves that and it copies the above code exactly, but the sequence of helper method calls are different (e.g. instead of calling get_user_id_from_user_name, it calls get_group_id_from_group_name; instead of calling nautilus_file_can_set_owner, it calls nautilus_file_can_set_group). This file has 4552 lines of code in that snapshot, of which 2779 lines were declared as cloned code by DECKARD. Also, our linked bug data shows that a total of 58 bugs were fixed during the project lifetime and a total of 798 lines were modified during bug fixing, but not a single bug has any cloned code in them.

Listing 1. Example Clone in Nautilus

```
void nautilus_file_set_owner(NautilusFile *file,
     const char *user_name_or_id,
     NautilusFileOperationCallback callback,
     gpointer callback_data)
{
  uid_t new_id;
  if (!nautilus_file_can_set_owner (file)) {
    nautilus_file_changed (file);
    (* callback) (file, GNOME_VFS_ERROR_ACCESS_DENIED,
          callback_data);
    return;
  }
  if (!get_user_id_from_user_name(user_name_or_id,&new_id)
  && !get_id_from_digit_string (user_name_or_id,&new_id))
  {
    nautilus_file_changed (file);
    (*callback)(file,GNOME_VFS_ERROR_BAD_PARAMETERS,
          callback_data);
    return;
  }
  if (new_id == file->details->info->uid) {
    (*callback)(file,GNOME_VFS_OK,callback_data);
    return;
  }
  set_owner_and_group (file,
           new_id,
           file->details->info->gid,
           callback, callback_data);
}
```

Listing 2. Example Clone in Gimp

```
Tool* tools_new_color_balance ()
{
  Tool * tool;
  ColorBalance * private;
  if (!color_balance_options)
    color_balance_options = tools_register_no_options
      (COLOR_BALANCE, "Color_Balance_Options");

  tool = (Tool *) g_malloc (sizeof (Tool));
  private = (ColorBalance *) g_malloc (sizeof (ColorBalance));

  tool->type = COLOR_BALANCE;
  tool->state = INACTIVE;
  tool->scroll_lock = 1; /* Disallow scrolling */
  tool->private = (void *) private;
  tool->auto_snap_to = TRUE;
  tool->button_press_func = color_balance_button_press;
  tool->button_release_func = color_balance_button_release;
  tool->motion_func = color_balance_motion;
  tool->arrow_keys_func = standard_arrow_keys_func;
  tool->cursor_update_func = color_balance_cursor_update;
  tool->control_func = color_balance_control;
  return tool;
}
```

In Listing 2, we show another clone from one of the largest clone groups, with 27 members totaling 775 lines of cloned code. All the clones come from different files, so this group spans 27 different files. Interestingly, all

these clones share a common API protocols. All of these clones first check whether some option is set, then they allocate an object, set some properties and then return that object. The code shown creates a ColorBalance object. Other clones likewise create different types of objects such as HueSaturation, BrightnessContrast, ByColorSelect etc. Our linked bug data indicates that a total of 50 bugs were fixed in all the files containing these clones during project lifetime, of which only 1 bug has trace of cloned code. This buggy cloned code came from some other clone in one of these files, but not from the above mentioned 27 member group!

We also did a case study on 250 randomly picked clone groups (100 from Apache, 100 from Nautilus, 25 from Gimp and 25 from Evolution), all with liberal settings to assess clone quality of DECKARD and to understand clone patterns. We used PostgreSQL random() function to pick random samples and found very few false positives. A great many of our observed clone groups contain direct copy/paste, or embody protocols for carrying important, common operations. Arguably, programmers copying from well-written code, or regurgitating familiar programming logic from memory, are less likely to produce error-prone code. Others were an artifact of the C language, and could be avoided using object oriented techniques. For example, in one Gimp clone group, members create different type of drawing objects (e.g. brush editor, gradient editor, palette editor) with slight change of code. This could have been avoided using a FACTORY METHOD or BUILDER pattern. Clearly, the availability of bounded polymorphism would have avoided code bloat: however, it appears, at least in this case, developers can manually generate bloated code to mimic bounded polymorphism without unduly impacting quality.

On the other hand, some clones simply cannot be avoided. E.g. in Nautilus, one clone group has two member functions for handling going back/forward in the file browser. Based on the action performed, these methods reorder two linked list (in different direction) and perform other actions on those list. A forced refactoring using linked list and function abstraction could render the code overly unintuitive. We also found some duplicate files in the projects.

In summary, all our evidence points to one conclusion: *Clones don't really smell that bad!*

VI. THREATS TO VALIDITY

A. Construct Validity

Bugs were collected from the Bugzilla databases for each project, and thus may not represent the complete set of all bugs. As the primary method by which users report problems, per community norms, and as they are reported manually and confirmed, we claim that project databases represent an important class of bugs which are indicative of aberrant behavior.

We used an automated bug linking process which may not be completely accurate. As a result, there may be both false positives and false negatives in the linked set. As discussed in Section V-A under RQ3, this does not pose an undue threat to RQ1 and RQ2, but some plausible failures to link might specially threaten the validity of our conclusion for RQ3. In a prior study [29] we evaluated the false positive and false negative rates and found the upper bounds on 95% confidence intervals to be less than 1% for bugs which were indeed linked by developers. Moreover, our bug introduction identification algorithm uses the `diff` tool. It is entirely possible that some of the changes in a revision marked as a bug fix are not, in fact, fixing lines which caused the bug. In lieu of this problem we use an approach used by well known prior studies [23]. Accuracy in identifying bug introducing changes may be increased by using advanced algorithms [30] and we are currently involved in additional studies assessing the quality of such data.

We use monthly snapshots instead of running analysis on every revision. This may introduce some imprecision as some of the buggy lines may not be mapped back to its staging snapshot because of their introduction into the system after their staging snapshot. We ignore such lines, but given the life of the projects (an average age of 132 monthly snapshots) and the level of significance observed in our findings, the results presented are robust. Also, our choice of monthly snapshot may not capture some late propagation of changes in different clone members (we do not build a clone genealogy, so once they have different staging snapshots, they are considered to affect different clone groups). However, we evaluated our datasets to determine the effect of such late propagation and found that on an average only 3.3% of bugs have fixes with late propagation that has different staging snapshots. So, this should not pose a significant threat to validity to RQ3. Note however, that RQ1 and RQ2 are not affected by this threat. In addition, although clone identification is not a sound and precise type of analysis (indeed, the very definition of a clone remains fuzzy and up for debate to some degree), we benefit by making use of DECKARD, which represents the current state of the art in clone detection.

B. Internal Validity

We have presented strong evidence that clones occur less frequently in buggy code than in the entire body of code. While strong correlation exists, the stringent requirements for causality have not been shown [31]. Despite this, our results do indeed cast doubt on the belief that code clones actually *cause* more bugs than non-cloned code, and provide support for further research examining *why* cloned code is decidedly less buggy.

C. External Validity

In attempt to address the generalizability of our findings, we have studied four real software projects that represent varying software processes and governance styles [32], with fairly consistent results across the different projects. However, while it is reasonable to believe that our results are representative of open source software, it is unclear how well they generalize to commercial software. Again, we have provided evidence that clones may in fact benefit code and plan to evaluate the relationship of clones with software quality in more diverse contexts.

VII. Conclusion

We have studied several medium to large projects to verify whether cloning is really a "bad smell". We took an empirical approach, based on actual bug-fix data to evaluate the extent to which clones are associated with code implicated in bug fixes. We find that 1) most bugs have very little to do with clones, 2) cloned code, in fact contains less "buggy code" (*viz.,* code implicated in bug fixes) than the rest of the system and 3) larger clone groups *don't* have more bugs than smaller clone groups, and in fact, making more copies of code *doesn't* introduce more defects, and furthermore, larger clone groups have *lower* bug density per line. than smaller clone groups. While others have made the argument before that clones aren't to be feared, our study is the first to quantitatively validate this claim using data mined from version control and bug repositories. In addition, to our knowledge ours is the first study to consider differences between smaller and larger clone groups.

Acknowledgment

We would like to thank Adrian Bachmann and Avi Bernstein for the Univ. of Zurich bug linking data. We also thank Lingxiao Jiang, Ghassan Mishergi, Zhendong Su and Stephane Glondu for providing us DECKARD. We extend our gratitude to anonymous reviewers for valuable comments on earlier version of this paper. We acknowledge support from an IBM Faculty Fellowship, and a gift from Microsoft Research. Most of all we acknowledge with gratitude support from the NSF Science of Design Program, grant No. SoD-TEAM 0613949. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] G. Alkhatib, "The maintenance problem of application software: An empirical analysis," *Journal of Software Maintenance: Research and Practice*, vol. 4, no. 2, pp. 83–104, 1992. [Online]. Available: http://dx.doi.org/10.1002/smr.4360040203

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, July 1999. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201485672

[3] M. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, September 2006. [Online]. Available: http://dx.doi.org/10.1007/s10664-006-9002-8

[4] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 55–64. [Online]. Available: http://dx.doi.org/10.1145/1287624.1287634

[5] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 33+. [Online]. Available: http://portal.acm.org/citation.cfm?id=857023

[6] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Aries: refactoring support tool for code clone," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–4, 2005. [Online]. Available: http://dx.doi.org/10.1145/1082983.1083306

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of java software systems based on clone analysis," in *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 326+. [Online]. Available: http://portal.acm.org/citation.cfm?id=837061

[8] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, September 2005. [Online]. Available: http://dx.doi.org/10.1145/1095430.1081737

[9] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," *Reverse Engineering, Working Conference on*, vol. 0, pp. 19–28, 2006. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2006.1

[10] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, pp. 1–34, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-009-9108-x

[11] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 86+. [Online]. Available: http://portal.acm.org/citation.cfm?id=836911

[12] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. IEEE Int. Conf. on Software Maintenance 1999 (ICSM'99)*, Oxford, UK, Aug 1999, pp. 109–118.

[13] R. Geiger, B. Fluri, H. Gall, and M. Pinzger, "Relation of code clones and change couplings," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, L. Baresi and R. Heckel, Eds. Berlin/Heidelberg: Springer-Verlag, 2006, vol. 3922, ch. 31, pp. 411–425. [Online]. Available: http://dx.doi.org/10.1007/11693017_31

[14] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," *Empirical Software Engineering, International Symposium on*, vol. 0, pp. 83–92, 2004. [Online]. Available: http://dx.doi.org/10.1109/ISESE.2004.1334896

[15] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 170–178. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2007.7

[16] ——, "Is cloned code more stable than non-cloned code?" in *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, October 2008, pp. 57–66. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2008.14

[17] E. D. Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 158–167. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.90

[18] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, pp. 804–818, November 2005. [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.114

[19] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," *Reverse Engineering, Working Conference on*, vol. 0, pp. 313–314, 2009. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.36

[20] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Clone-aware Configuration Management," in *ASE*, vol. 9, pp. 16–20.

[21] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 173–180. [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2004.35

[22] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 485–495. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070547

[23] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1145/1083142.1083147

[24] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[25] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, August 2002, pp. 120–130. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2000.883028

[26] D. Čubranić and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 408–418. [Online]. Available: http://portal.acm.org/citation.cfm?id=776816.776866

[27] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 23+. [Online]. Available: http://portal.acm.org/citation.cfm?id=943568

[28] A. Bachmann and A. Bernstein, "Data retrieval, processing and linking for software process data analysis," University of Zurich, Technical Report, 2009, published May, 2009. http://www.ifi.uzh.ch/ddis/people/adrian-bachmann/pdq/.

[29] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1145/1595696.1595716

[30] S. Kim, T. Zimmermann, K. Pan, and J. Jr, "Automatic identification of bug-introducing changes," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.23

[31] S. Kan, *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[32] J. Berkus, "The 5 types of open source projects," 2007, march 20, 2007 http://www.powerpostgresql.com/5_types.