Tracing Back the History of Commits in Low-tech Reviewing Environments

A case study of the Linux kernel

Yujuan Jiang,
Bram Adams
MCIS, Polytechnique
Montréal, Canada
{yujuan.jiang,
bram.adams}@polymtl.ca

Foutse Khomh
SWAT, Polytechnique
Montréal, Canada
foutse.khomh@polymtl.ca

Daniel M. German
University of Victoria, Canada
dmg@uvic.ca

ABSTRACT

Context: During software maintenance, people typically go back to the original reviews of a patch to understand the actual design rationale and potential risks of the code. Whereas modern web-based reviewing environments like gerrit make this process relatively easy, the low-tech, mailing-list based reviewing environments of many open source systems make linking a commit back to its reviews and earlier versions far from trivial, since (1) a commit has no physical link with any reviewing email, (2) the discussed patches are not always fully identical to the accepted commits and (3) some discussions last across multiple email threads, each of which containing potentially multiple versions of the same patch.
Goal: To support maintainers in reconstructing the reviewing history of kernel patches, and studying (for the first time) the characteristics of the recovered reviewing histories.
Method: This paper performs a comparative empirical study on the Linux kernel mailing lists of 3 email-to-email and email-to-commit linking techniques based on checksums, common patch lines and clone detection.
Results: Around 25% of the patches had an (until now) hidden reviewing history of more than four weeks, and patches with multiple versions typically are larger and have a higher acceptance rate than patches with just one version.
Conclusion: The plus-minus-line-based technique is the best approach for linking patch emails to commits, while it needs to be combined with the checksum-based technique for linking different patch versions.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEM'14 September 18-19, 2014, Torino, Italy.
Copyright 2014 ACM 978-1-4503-2774-9/14/09... ...$15.00.

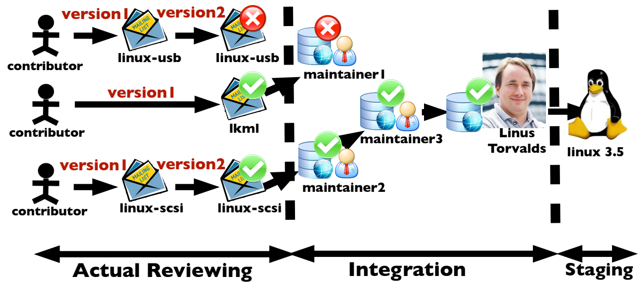

Figure 1: The reviewing and integration process of a patch in the Linux kernel project (adapted from [19]).

General Terms

Software Engineering

Keywords

review, traceability, clone detection, low-tech reviewing environment, mailing list, open source, Linux kernel

1. INTRODUCTION

In November 2003, there was a suspected backdoor attempt to taint the source code of the Linux kernel project [23]. The Linux team noticed that a patch popped up in the CVS copy of the version control system that never appeared in the BitKeeper master copy. This patch pretended to just check for errors, but actually contained a back door that could render a system vulnerable. Up until today, it is still not clear whether this code was inserted by a malicious hacker, since there was (and still is) no explicit link from a commit back to all emails reviewing that patch or earlier versions of it. The recent "heartbleed" vulnerability [16] also highlights the need to have traceability from commits in the version control system to patches in the mailing list. Full traceability would have made a quicker and more accurate audit of the code possible. Apart from security audits, traceability from submitted patches to accepted commits would also benefit regular maintenance, as it will make it easy to review the email messages around that patch which comprise the reviewing and design discussions of the patches [2][5][25].

Although more and more open and closed source systems are migrating towards modern, online reviewing environments like Gerrit, which consolidate all patch versions, discussions and links to commits in one location, many open source ecosystems such as the Linux kernel and the Apache Software Foundation still swear by low-tech reviewing environments based on a mailing list [6]. Basically (Figure 1), a contributor sends an email containing a patch, asking for feedback. Other project members with the required expertise then jump in to provide design- and code-related comments. They either reject the patch up front (not an interesting feature at all), they allow the responsible kernel maintainer to commit the patch to the version control system as is, or they request a new version with modifications. In the latter case, the original patch submitter should send a new patch version, hoping that this one will be successful. This process continues until the patch is rejected, accepted or the submitter gives up.

Although modern reviewing environments require the same steps, the usage of a low-tech environment introduces three major challenges for people to trace a commit back to the original patch (and its reviews):

- **Reviewing process detached from development process.** Patches and reviews are spread across email messages in one or more mailing lists, while commits are stored in a version control system like Git (see Figure 1). Emails cannot reference the commit id of a patch that has not been committed yet, while emails do not have a universal identifier that can be referenced by a commit.
- **Accepted commits can differ significantly from original, submitted patches.** During the reviewing process, the reviewers are likely to ask the developer to revise it. Even after the maintainers integrate the patch into a Git branch, maintainers can still modify a patch by changing the order of commits ("rebasing") or filtering out uninteresting changes from the patch("cherry-picking").
- **Patches evolve across multiple versions, possibly submitted in multiple email threads.** During the reviewing process, a patch may evolve substantially, yielding a series of patch versions. Such a version can be sent in a reply to the original email thread, or sometimes in a new email thread. In the best case, such a sequel thread is announced in the subject of the new thread, for example "[V2]Patch: remove the deadlock code in driver subsystem", however this practice is not enforced and typically only used by more experienced developers.

In this paper, we propose an approach to recover the full reviewing history of a patch in a low-tech reviewing environment. This approach mainly consists of two parts: (1) linking a commit to any email submitting the original patch (often the last patch version), and (2) linking all emails about the same patch together to reach the first version. We used three techniques of different strictness and granularity to link commits to emails and emails to emails: (1) a checksum-based technique (strictest, chunk level), (2) a novel plus-minus-line-based technique (medium strict, line level), and (3) the CCFinder clone detection tool (least strict, token level). After tracing the commits to all emails up to their very first patch version, we are then able to quantitatively analyze (for the first time) the full reviewing history of a patch sent to a mailing list.

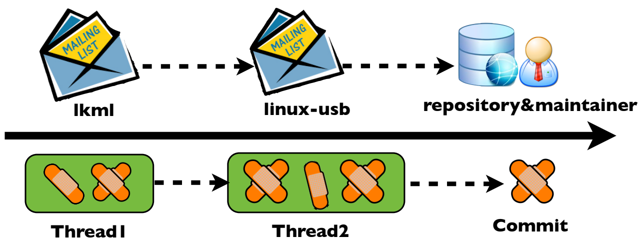

Figure 2: The evolution process of a patch with multiple versions ("super-thread").

We addressed the following three research questions:

RQ1) *Can commits be linked accurately to emails containing the corresponding patch version?*

Since emails and git commits are two different media, we first analyze which technique works best to link an email containing a patch to the accepted commit. The checksum-based technique has the highest precision (97.9%±5%), while the plus-minus-line-based technique has the highest relative recall (85.69%±5%).

RQ2) *Can emails containing different patch versions be linked accurately to each other?*

Since different versions of a patch can be submitted and reviewed across different email threads, we evaluate different techniques for linking related emails to each other. The checksum-based technique again has the highest precision (86.98%±5%), while the plus-minus-line-based technique again obtains the highest relative recall (68.68%±5%). Together, the checksum-based and plus-minus-line-based techniques can discover more than 95%±5% of the correct patch pairs.

RQ3) *What are the characteristics of the reviewing history in a low-tech reviewing environment?*

Given the means to link patch emails to each other and to commits, we can now study the characteristics of patches having multiple versions. we find that around 25% of the patches have a full reviewing history of more than four weeks. Patches that underwent multiple versions have a higher chance to be accepted and especially consist of bug fixes compared to regular patches.

In the rest of the paper, we first introduce the three techniques we applied (Section 2) and provide our case study setup (Section 3), followed by the findings of our case study (Section 4). We conclude with threats to validity (Section 5), related work (Section 6) and the conclusion (Section 7).

2. THREE LINKING TECHNIQUES

In general, to recover the evolution process of a commit, we need to (1) trace it back from its final version seen (the accepted commit) to the email submitting the committed version of the patch, then (2) link together all emails containing different versions of the same patch until we reach the first patch version. Since in both phases one tries to map one version of a patch to another one, we use and evaluate the same linking techniques for both. We do not consider the textual content of the reviewing emails themselves, nor commit metadata like the commit log message or author names, since those contain natural language vocabulary, which is known to be much more variable and hence harder to link than source code [15].

Figure 3 shows an example committed patch in the git format used by Linux and many other open source systems. The part above the "diff −−git" line consists of commit metadata such as the commit id, author name, author data and commit log message, whereas the bottom part contains the actual patch. In this case, the patch describes the changes to a file called "src/stash.c", where on line 587 the developer has removed the line starting with a minus sign and replaced it by the lines starting with a plus sign. The code lines that do not start with a plus or minus sign correspond to the context of the patch, showing the surroundings of the changed lines.

Patches can contain multiple bursts of plus/minus lines (multiple changes to one file), and even multiple "diff −−git" sections changes to multiple files. All changes to one file are called a "chunk". The difference between a committed patch (like Figure 3) and a patch version in an email is that for the latter the commit log message occurs as email body, the author name and data correspond to the sender name and date of the email, and there is no commit ID attached to the email, since the patch is not yet committed.

2.1 Checksum-based Technique

The checksum-based technique is the strictest and most coarse-grained technique[19]. It computes MD5 hash checksums for each chunk in a commit or email patch, then links each chunk's checksum to the most recent email patch chunk with the same checksum. If an email patch has at least one chunk that was linked to a commit or email chunk, we link the patch as a whole to that commit or email. Finally, we link two email threads together if they have at least one linked email pair.

For each chunk, we first filtered out the unchanged code lines (those without plus or minus sign), remove all white space and capitalization, then concatenate all changed lines (i.e., the lines starting with a + or - in Figure 3) into one line. After prepending the relative path name of the file changed by the chunk, we calculate the chunk's MD5 checksum. We perform the white space, capitalization and concatenation processing to deal with small changes done to a patch before merging, and the path name prepending to reduce false positive matches with similar changes to other files. We perform the same chunk-level procedure to the commits in the Git repository.

Similar to Bird et al. [11], we did not directly link a whole patch to a Git commit, but used the intermediate step of chunks, because of "cherry-picking". This is a common integration activity where an integrator only picks the interesting parts of a patch and ignores the rest. Large patches risk not being merged completely, or maybe not in one Git commit. We link a whole patch to a commit, if at least one of its chunks are linked to that commit.

2.2 Plus-Minus-Line-Based Technique

The plus-minus-line-based technique looks for patches containing sufficient identical changed lines (instead of a whole chunk), which is a compromise between both strictness and granularity compared to the other two techniques. Hence, the technique mainly focuses on the lines beginning with a plus or minus sign in Figure 3.

```
commit 52c52737353a7ee7b653ab314d7b89ca6ddafe63
Author: Russell Belfer <rb@github.com>
Date:   Wed May 1 15:51:30 2013 -0700

    Clear error msg when we eat error silently

diff --git a/src/stash.c b/src/stash.c
index 355c5dc..19b29be 100644
--- a/src/stash.c
+++ b/src/stash.c
@@ -587,8 +587,10 @@ int git_stash_foreach(
        const git_reflog_entry *entry;

        error = git_reference_lookup(&stash, repo,
GIT_REFS_STASH_FILE);
-       if (error == GIT_ENOTFOUND)
+       if (error == GIT_ENOTFOUND) {
+               giterr_clear();
                return 0;
+       }
        if (error < 0)
                goto cleanup;
```

Figure 3: A commit patch example. The line with a beginning of a plus sign "+" indicates an added line while a minus "-" sign indicates a deleted line.

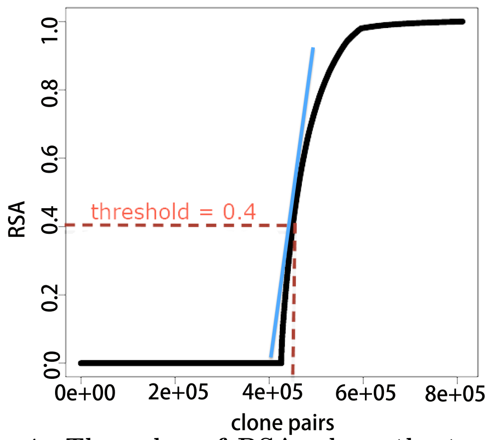

Figure 4: The value of RSA where the tangential line (blue) has the largest slope.

The linking algorithm consists of two steps. First, we parse all the Git commits and extract the changed file name, the changed line type (plus + or minus -) and the line content. We save all the information into an sqlite3 database in chronological order. Then, we analyze all emails containing a patch: for each line beginning with a plus "+" or minus "-", we query the database to find other emails or commits containing the same changed line in order to compute the proportion of matched lines with each later patch. Finally, we rank the matched patches according to the proportion of matched changed lines and output the matched patch with the highest proportion. To avoid slight changes to the patch during the reviewing process, we again remove all whitespace of each line (similar to the checksum-based technique).

2.3 Clone-Detection-based Technique

As our third technique, we use the popular CCFinderX clone detection, which is a token-based clone detection tool that can detect code clones in C, C++, Java, COBOL, and other source files. It transforms the input source text into tokens, then compares files token by token, and generates pairs of token sequences ("clones") that are found to be very similar (according to some threshold). Since such a technique allows some tokens in the matched token sequences to be different, this is the least strict and most fine-grained technique of the three [10].

We create one file for each patch version and each commit, then feed all files to CCFinderX. Afterwards (for email to commit links), we remove pairs of clones belonging to two emails from the output as well as unfeasible pairs where a commit occurred before the email was sent. Note that to enable CCFinderX to process patches instead of regular source files, we took the changed lines, removed the plus and minus characters, then put braces around the resulting code, forcing CCFinderX to consider the changed lines as a regular code block.

Since CCFinderX is known to produce false positives, we need to filter out (email, commit) or (email, email) pairs with just a few common clones. To help with this, CCFinderX computes some metrics about the analyzed input files and clones, one of which is the RSA metric. This metric indicates the similarity percentage between each file (based on the number of common clones). If two files have an RSA value close to 100%, then one of them is very likely to be a copy of another file. To use the RSA value as a threshold to filter out false positive pairs, we rank all the (email, commit) or (email, email) pairs by their RSA value, see Figure 4, then compute the delta in RSA value between each successive pair. We then define as threshold the RSA value of the (email, commit) or (email, email) pair with the largest delta in RSA, i.e., where the slope of the tangential line changes the most in Figure 4. We filter out the clone pairs with a value lower than the RSA threshold.

3. CASE STUDY SETUP

In this section, we present the details of the case study that we performed on the Linux kernel data to evaluate the three linking techniques.

3.1 Data Extraction

This paper uses the Linux kernel as representative example of an open source project with a low-tech reviewing environment. The Linux kernel reviewing and development process is supported by the Git distributed version control system, the main kernel mailing list (LKML) and more than 130 specialized subsystem mailing lists. These mailing lists are archived online as textual mbox files. After downloading these files from 2009 to 2012, we imported their content and metadata, e.g., subject and author, into a relational database using MailMiner [9]. We ignore email attachments, since the Linux kernel developer guidelines state that all patches should be inlined into the email body.

To analyze those patches that are successfully integrated into an official Linux release, we collect the commits of Linus Torvalds' repository from 2009 (start reviews data) until 2012, which contains the commit information of accepted patches Figure 3. Note that some emails of end 2012 will not have had the chance of reaching Git, however this does not impact our findings.

3.2 Evaluation of Linking Techniques

To evaluate the performance of the three techniques for RQ1 and RQ2, we compute precision and **relative** recall for each, and compute "real" recall for RQ2.

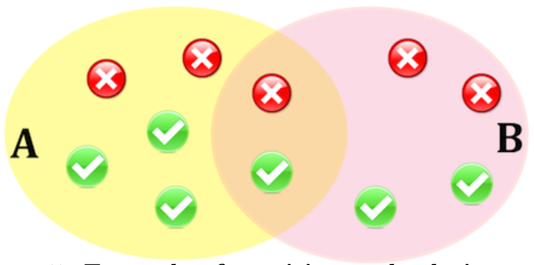

Figure 5: Example of precision and relative recall.

In both cases, we manually analyze precision. We sample 384 pairs from the results of each technique to obtain a confidence level of 95% and confidence interval of 5% [18]. Then, we run scripts to collect the Git log of a commit and the contents of the email to which a commit has been mapped in order to manually check attributes such as the title, author and review comments to determine if the matched commit and patch are really talking about the same patch.

Since pure recall is hard to compute due to the lack of ground truth, we use the concept of **relative recall** in order to measure the sensitivity of the three techniques. Such **relative recall** is defined as the number of correctly detected (email, commit) or (email, email) pairs out of the union of true positive pairs identified during our manual analysis for precision of the three techniques. Although not identical to pure recall, such a relative recall gives an indication of the amount of false negatives in the results.

For example, Figure 5 shows how technique A detects 7 candidates, 4 of them being correct, and how technique B detects 6 candidates, three of which are detected correctly. In this case, the union of all correctly detected candidates is 6, 4 of which are detected by A while 3 are detected by B. Then, the relative recall of technique A is 4 out of 6 (67%) while the relative recall of technique B is 3 out of 6 (50%).

In the case of email to email linking, there is a partial source of ground truth, which we also use to obtain a form of real (i.e., non-relative) recall. Basically, some developers reuse the same subject for different email threads, or add a qualifier like "[v2]" or "[V 3]". Using regular expressions, we recovered such links between threads, then randomly sampled 384 of them (confidence level of 95% and confidence interval of 5%) as ground truth for calculating recall.

Finally, we also compare overlap between the results of each technique to figure out if the techniques are complementary or subsume each other. The overlap is computed by counting the number of detected pairs shared with the other techniques. If two techniques are overlapping substantially, it means that they detect almost the same pairs. On the contrary, if they are not overlapping too much, they are complementary to each other. In this case, one should probably use the union of their results for further analysis to obtain the best precision and recall.

3.3 Analysis of the Reviewing Process

As a first application of the linking techniques, we are able to analyze (for the first time) the reviewing history of kernel commits back to the original submitted patch versions (depending on the performance of the linking techniques). For this analysis, we introduce the concept of "super-thread", which connects all patch versions of a commit to each other chronologically. We call the sequence of all versions of a

Table 1: Overview of the reviewing history metrics and dimensions analyzed.

	metric name	type	source	explanation
Review	`thr_volume`	numeric	thread	Number of email messages between start of current thread and current patch version.
	`nr_reviews`	numeric	thread	Number of review messages of a patch version.
	`review_time`	numeric	patch	Time in seconds from current patch version to its last review message.
	`response_time`	numeric	thread	Time in seconds from current patch version to its first review message.
	`first_response_time`	numeric	patch	Time in seconds from first patch version in thread to its first review message.
Patch	`size`	numeric	patch	Patch churn (sum of number of added and removed lines).
	`spread`	numeric	patch	Number of files changed by current patch version.
	`spread_subsys`	numeric	patch	Number of subsystems changed by current patch version.
	`bug_fix`	boolean	patch	Does this patch version contain a bug fix (as opposed to a new feature or enhancement)?
	`accepted`	boolean	commit	Has this email patch version been accepted as a commit?

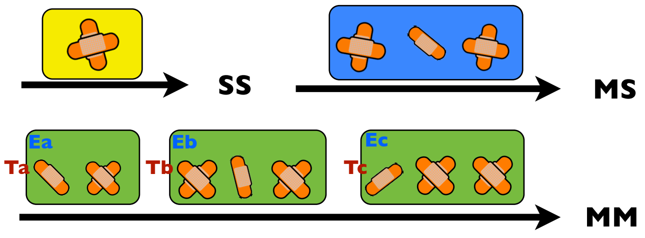

Figure 6: The three types of super-threads.

patch a "super-thread", since the email patches could be spread across one or more email threads. For example (Figure 6), if there are three emails patches Ea, Eb and Ec in three different threads Ta, Tb and Tc, and we are able to link Ea to Eb, and Eb to Ec, eventually we could link the three threads Ta, Tb and Tc together. Figure 6 illustrates the three different kinds of super-threads: SS (single patch version, single thread), MS (multiple patch versions, single thread) and MM (multiple patch versions, multiple threads).

We use the following steps to identify super-threads: (1) mapping each email to its enclosing thread (based on the mbox data), (2) lifting links between emails up to links between threads, (3) filtering out pairs within the same thread, and (4) transitively chaining (thread, thread)-pairs together if they have at least one thread in common.

We use the best of the three linking techniques to recover the (email, commit) and (email, email) links. Especially (relative) recall is important, however we also want to avoid too many false positives. For this reason, we use one or two techniques that together cover a large enough part of the relative recall, but we also take additional measures to keep precision high enough. We will discuss these for RQ3.

After recovering the super-threads, we performed quantitative analysis to analyze the reviewing history and compare important review characteristics. The characteristics that we analyzed are listed in Table 1. These characteristics cover two dimensions, i.e., "Review" and "Patch". "thr_volume" measures the reviewing activity for a patch. A higher value means that the patch is being discussed more, "nr_reviews" indicates the number of reviews on a patch version. "review_time" depicts the discussion time for a patch version. The "response_time" and "first_response_time" metrics measure the reaction of reviewers to a patch. "size" indicates how large a patch is. "spread" and "spread_subsys" metrics depict the invasiveness of a patch. Finally "bug_fix" and "accepted" indicate whether a patch version contains a bug fix (i.e., contains the words "bug", "fix" or "error" [20] in the commit message) and whether it eventually was accepted as a commit.

Table 2: Evaluation of three techniques of linking (email, commit) pair.

technique	precision	relative recall	F-score
Checksum	97.92%	47.68%	0.64
Plus-minus-line	85.16%	85.69%	0.85
Clone-detection	81.77%	37.40%	0.51

4. CASE STUDY RESULTS

In this section, we answer each research question and present its motivation, approach and findings.

RQ1: Can commits be linked accurately to emails containing the corresponding patch version?

Motivation: In a project like the Linux kernel, where the development process and reviewing process are supported by separate Git and mailing lists, there is no explicit link between the reviews of a patch and its corresponding commit. Here, we analyze whether, despite the three challenges of the introduction, the three linking techniques are still able to link a commit to at least one of its patch versions. If so, the techniques could be used to retrieve details and comments about the design rationale of a commit from corresponding emails.

Approach: We applied the three techniques to find the correct (email, commit) pairs. We then calculate precision, and **relative** recall of a representative sample of 384 (email, commit) pairs, one such sample per technique (all percentages need to be interpreted as having a 95% confidence level ± 5% confidence interval). Finally we also calculate the overlapping results between the three techniques. The results of precision, relative recall and F-score are listed in Table 2.

Findings: **The checksum-based technique has the highest precision in linking commits to emails.** The checksum-based technique detects 216,033 (email, commit) pairs, while the plus-minus-line-based technique detects 531,381 and the clone-detection-based technique (CCFinderX) detects 168,408 (after filtering with an RSA threshold of 0.4). After manually validating 384 random samples, we obtained a precision of 97.92% for the checksum-based technique (376/384), 85.16% for the plus-minus-line-based technique (327/384) and 81.77% for the clone-detection-based technique (314/384).

The checksum-based technique has the highest precision since it is the most strict, which means that it is very hard

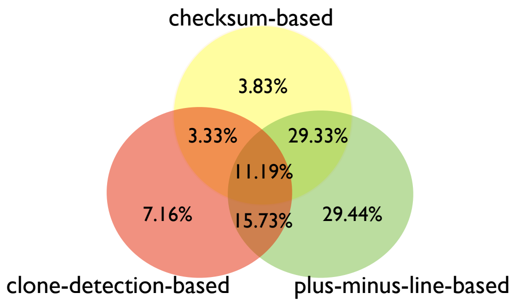

Figure 7: Overlap of (email, commit) pairs detected by the three techniques.

to accidentally find two almost identical changes (chunks) to the same file. The heuristic to match a commit to an email patch as soon as they share one common chunk seems to work well. On the other hand, the clone detection approach makes the most errors since it is the most liberal of the three.

The plus-minus-line-based technique has the highest relative recall. Out of the 992 true positive (email, commit) pairs identified across the three techniques (union of unique true positives after manual analysis), 473 are detected by the checksum-based technique, 850 by the plus-minus-line-based technique and 371 by the clone-detection-based technique. The relative recall for each technique is 47.68%, 85.69% and 37.40% respectively. Hence, relative recall-wise the plus-minus-based technique is almost twice as good as the other two techniques.

We also found that between 16.98% and 19.23% of the techniques' true positives mapped a commit to the last available version of a patch, while more than 80% mapped the commit to earlier versions (which are more different and hence harder to match). The low performance of the checksum-based technique is due to activities like cherry-picking or modifications below the chunk-level. Since checksums are calculated per chunk (i.e., per changed file), a patch that is modified before being committed just to fix a typo or some other non-whitespace related reason will yield a different checksum and hence not be linked correctly. This suggests that the checksum-based technique likely needs to be refined below the chunk level.

The plus-minus-line-based technique has the highest F-score in linking (email, commit) pair. Since the highest precision and relative recall go to different techniques, to verify which technique is the best one overall we compute the F-score for each technique, which is a combination of precision and recall [26]. As expected based on the recall results, the plus-minus-line-based technique has the highest F-score (0.85), and hence works best for linking commits to emails. It seems a good compromise between strictness and granularity. Being less strict means that it can avoid false negatives while its fine granularity still does not concede too many false positives.

The plus-minus-line-based technique can link the majority of (email, commit) pairs. Figure 7 shows the overlap of results of the three techniques. Only 3.83% and 7.16% of the pairs are identified uniquely by the checksum-based technique and the clone-detection-based technique respectively, whereas the plus-minus-line-based technique is the most effective technique for linking commits to email patches.

For linking emails to commits, the plus-minus-line-based technique has the highest relative recall and F-score, and it covers most of the correct (email, commit) pairs.

RQ2: Can emails containing different patch versions be linked accurately to each other?

Motivation: In the context of low-tech reviewing environments, the reviewing process consists of replying to emails containing a submitted patch, sent to a developer mailing list. However, sometimes the contributor may start a new thread to submit a new version of a patch, causing multiple threads to exist that talk about the same patch (we call this set of threads a super-thread), yet do not contain an explicit link to each other and to the final commit. Hence, people may regard one thread of a super-thread as the sole reviewing thread of a certain patch and ignore other related reviewing threads, which will cause underestimation of the actual reviewing process and miss important design decisions.

Approach: In order to recover the reviewing super-threads, we apply the three linking techniques to search the emails containing similar patches across different threads. We then compute the precision and relative recall of the results of each technique, as well as the overlap between techniques (just like for RQ1). Contrary to email-to-commit linking, we can also provide real recall based on threads containing annotations in their subject. The results are listed in Table 3. For CCFinderX we obtained a different RSA threshold (0.6) than for RQ1.

Findings: **The checksum-based technique has the highest precision.** The checksum-based technique detects 3,020,582 (email, email) pairs, while the plus-minus-line-based technique detects 4,565,857 and CCFinderX detects 6,124,303, respectively. We again randomly select 384 samples from the results of each technique. After manually validating the samples, we obtained a precision of 86.98% (334/384) for the checksum-based technique, 51.82% (199/384) for the plus-minus-line-based technique, and 70.83% (272/384) for the clone-detection-based technique.

The checksum-based technique again has the highest precision, for similar reasons as for RQ1. However, the clone detection technique and (especially) the plus-minus-line-based techniques dropped in precision. The explosion in number of patches available for linking makes the plus-minus-line-based technique select incorrect patches.

The plus-minus-line-based technique has the highest relative and real recall. Out of union of 796 true positive (email, email) pairs, 424 are detected by the checksum-based technique, 547 by the plus-minus-line-based technique and 148 by the clone-detection-based technique. According to our definition, the relative recall of the checksum-based technique is 53.27%, of the plus-minus-line-based technique 68.72% and of the clone-detection-based technique 18.59%.

For email to email linking, we also have an actual ground truth based on a convention in Linux kernel reviews where the subjects of patch versions include tags like "V2" or "v3". Starting from the 463,697 subjects of the first email of each thread, we used regular expressions to remove version tags like "V2" or "v3", then counted how many subjects become identical (only difference was the tag). Only 1,912 subjects (i.e., 0.41% of the threads) could be linked this way. Although only few experienced contributors follow this convention, we randomly selected 384 samples with confidence

Table 3: Evaluation of the three techniques for linking (email, email) pairs.

technique	precision	rel. recall	F-score	real recall
Checksum	86.98%	53.27%	0.66	76.83%
Plus-minus-line	51.82%	68.72%	0.59	80.47%
Clone-detection	70.83%	18.59%	0.30	23.44%

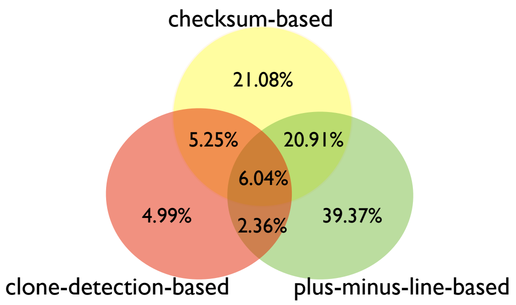

Figure 8: Overlap of (email, email) pairs detected by the three techniques.

level of 95% and confidence interval of 5% as ground truth, then checked how many of them are detected by each technique.

We found that the recall of the checksum-based technique is 76.83%, of the plus-minus-line-based technique 80.47% and the recall of the clone-detection-based technique is 23.44%. Although the values are slightly higher than the relative recall, it follows the same trend. As in RQ1, the plus-minus-line-based technique still has the highest (**relative**) recall, but with a lower performance than for (email, commit) pairs. The checksum-based technique does better than for email to commit pairs, while the clone-detection-based technique loses half of its relative recall compared to email to commit pairs. Note that, similar to RQ1, most of the true positive results were relatively hard to match: less than 8.72% of the matched pairs of each technique consisted of patch versions sent in the same thread, all others were between patch versions in different threads!

The checksum-based technique has the highest F-score. Since the highest precision and relative recall again belong to different techniques, we compute the F-score for each technique, similar to RQ1. The highest F-score goes to the checksum-based technique.

The checksum-based and the plus-minus-line-based techniques are complementary. Using the ground truth of 796 (email, email) pairs for relative recall, we can see (Figure 8) that 21.08% of them are detected only by the checksum-based technique, while 39.37% could only be found by the plus-minus-line-based technique compared to 4.99% by the clone-detection-based technique. The combination of the checksum-based and the plus-minus-line-based techniques together take up more than 95% of the ground truth.

To better understand the performance of each technique, we randomly picked 10 pairs from the results of each technique that are only detected by itself to analyze their characteristics: The patches only linked by the checksum-based technique, typically range from 15 LOC to 3,334 LOC, with an average size of 427.6 lines, compared to 3 LOC to 541 LOC (99.5 LOC on average) for the plus-minus-line-based technique and 45 LOC to 7,844 LOC (1,196.7 LOC on average) for the checksum-based technique. This difference

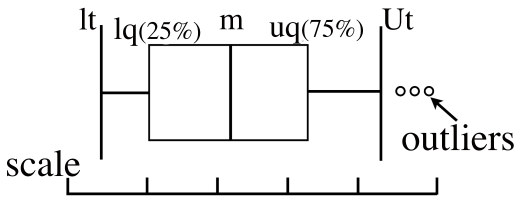

Figure 9: Boxplot indicating how to remove the outliers in our data.

stems from the fact that the checksum-based technique matches full chunks instead of fine-grained lines or groups of tokens. Furthermore, the plus-minus-based technique only matches pairs of patches who have a majority of lines in common, which makes it hard for patches with uncomparable size or large patches to obtain a match. Finally, the very large size of clone-detection-matches does not just come from the minimum size threshold used by clone detection tools, but rather from the many false positives amongst small clones. Generic patch lines consisting just of an assignment or a for-loop will be matched to many emails incorrectly, while large sets of lines or whole files containing common lines have a much higher chance of being uniquely distinguishable and hence map only to the right patch.

For linking emails to emails, the checksum-based technique overall performs the best, whereas it requires the plus-minus-line-based technique to obtain a higher recall.

RQ3: What are the characteristics of the reviewing history in a low-tech reviewing environment?

Motivation: Now that we have successfully evaluated algorithms to link commits to their earlier email patches, we can now (for the first time) study low-tech reviewing environments from a quantitative point of view. Similar to studies on modern reviewing environments [2][4][27], one can now study the following questions: How long do email-based reviews take? How many reviewers and reviews are there? Since this paper is the first to trace back from a commit to the earliest emails containing patch versions, we also try to understand why some patch versions are spread across multiple threads. Is this a bad situation for a submitter (lower acceptance rate?) and reviewers (waste of time to review later versions?), or rather a recipe for success (higher acceptance rate?)? What kind of patches (bug fixes, large vs. small, etc.) typically have multiple versions?

Approach: To answer the above questions, we combine the best and most complementary techniques for linking emails to emails (the checksum-based and the plus-minus-line-based) to obtain a linking technique capable of obtaining a high precision and recall for recovering the super-threads of commits on the full data set, not just on a sample of 384 pairs.

Manual analysis of the resulting super-threads showed several infeasible matches with emails more than ten years apart. To filter out such matching results, we only keep super-threads with a time duration below the upper tail (ut) of the data set according to the following formula [27]:

$$ut = (uq - lq) * 1.5 + uq$$

Among them, uq indicates the 75% quartile, while lq indicates the 25% quartile of the data as shown in Figure 9. The

Table 4: **Average value of characteristics in different types of threads (including rejected patches).**

	metric name	MM	MS	SS
Review	`thr_volume`	3.838	6.051	3.533
	`nr_reviews`	1.046	1.936	1.165
	`review_time (day)`	1.932	3.022	2.271
	`response_time (day)`	0.871	1.131	1.030
	`first_response_time (day)`	0.801	1.215	1.010
Patch	`size`	81.660	146.100	25.430
	`spread`	2.398	3.811	1.016
	`spread_subsys`	1.387	1.750	1.003
Other	`acceptance`	46.05%	43.97%	23.26%
	`bug-fix`	49.94%	36.72%	31.10%

Table 5: **Time duration (#days) of the super-threads of type MM.**

	time duration	# of patch versions
Min.	0	2.000
1st Qu.	2.687	2.000
Median	10.061	2.000
Mean	21.341	3.172
3rd Qu.	32.923	3.000
Max.	107.524	108.000

data with value higher than the upper tail uq are considered to be outliers and we remove them. Note that (contrary to RQ1 and RQ2) this RQ considers both patches that are accepted (linked to a commit) as well as those that are rejected (no corresponding commit).

Findings:

Qa) *How far back does the reviewing history of a patch version go?*

Around 25% of the MM patches has a previously "hidden" reviewing history of more than four weeks. We analyzed the time duration of the MM super-threads, i.e., from the very first design discussion until the last discussion of a patch (possibly across multiple threads in case of MM). Table 5 shows that most of the super-threads consist of only few patch versions (Mean. value is 3.172), but can last a long time (Mean. value is 21.341 days). More than 25% of the patches have a reviewing time that is 4.5 weeks longer than considered by researchers thus far [19].

Qb) *What kind of patches undergo multiple patch versions?*

Patches evolving across multiple versions are larger and affect more files than those with a single version. The patches from threads of type MM (especially) and MS have higher values for "size", "spread" and "spread_subsys", as shown in Table 4. This indicates that the patches undergoing multiple versions tend to be larger and more complex, and hence need more attention before being integrated. Surprisingly, such invasive patches seem to feature especially in single threads, rather than multiple ones. As we will see below, this means that they are still integrated relatively quickly, whereas the patches that need more versions tend to be slightly less invasive. A Kruskal-Wallis test with post-hoc tests verified the significant difference.

Qc) *What kind of patches undergo multiple threads?*

Kernel developers use multiple threads if too much time has passed since the previous patch version. We

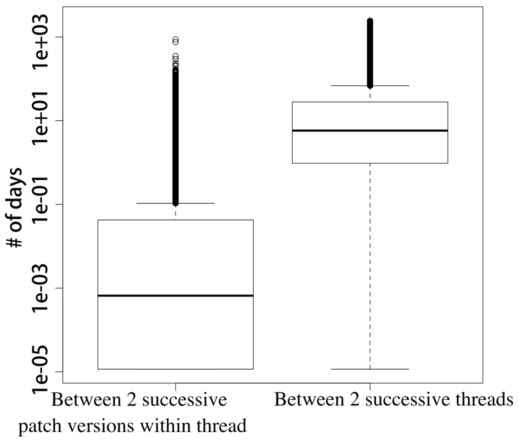

Figure 10: Boxplot of average time interval (#days) between two successive patch versions/threads of super-threads.Given the log-scale, a value of 1e-05 in fact denotes zero.

compared the time distribution of the interval between two successive threads to that of two successive patch versions within one thread. The result is shown in the boxplots of Figure 10. We can see that the time interval of threads is much longer than that of patch versions. This seems to confirm the intuition that people typically start a new thread when too much time has passed since the last review or version of a patch, whereas they would continue the same thread otherwise. A Mann-Whitney test with as null hypothesis "no difference between the average of both time distributions" obtained a p-value $<$2.2e-16, which confirms that the differences are statistically significant.

Threads of type MM especially consist of bug-fixes. Out of all MM threads, 49.94% are bug-fixes, compared to 36.7% for MS and 31.10% for SS threads. The pairwised Mann-Whitney test show that MM has no significant difference with MS, but that MM and MS are significantly different from SS.

On the one hand, this finding seems surprising, since one would expect bug-fixes to be smaller and hence require less discussion. On the other hand, bugs might be risky to fix, and hence require care and thorough reviews.

Qd) *Do reviewers lose interest in multi-version patches?*

Patches of MM and MS threads involve more discussion than SS threads. We compared the number of reviews discussing a patch for the three different types of threads. The value of "thr_volume" of MM and MS patches is higher than for the SS, which means that if a patch needs to undergo one or more additional versions reviewers seem to discuss more about it and provide more constructive comments to help improve it to be accepted. The amount of reviewing hence does not suffer from having multiple versions of a patch. A Kruskal-Wallis test with post-hoc tests showed that the three groups are different from each other.

Patches of type SS and MS have fewer number of reviews. SS and MS patches receive the most receives (nr_reviews), and hence take more review_time as well. Hence, it seems like patches evolving across multiple versions attract fewer reviews. A Kruskal-Wallis test with post-hoc tests showed that the difference is significant. One possible explanation could be that MM patches received the majority of their reviews early on, with later patch versions receiving more focused reviewing.

Reviewers are more eager to review MM threads. Although we did not find a statistically significant difference between the distributions of metrics "response_time" and "first_response_time" for SS and MS, we found that MM threads significantly take less time before the first review. Reviewers seem to consider such patches as having a higher priority than other types of superthreads.

Qe) *Do multi-version patches have a lower chance of acceptance?*

No, threads of type MM in fact have a higher acceptance rate. In Table 4, we have marked a super-thread as accepted if at least one of its patch versions could be linked to a commit. We found that the threads of type MM have the highest acceptance rate (46.05%), followed by MS patches with 43.97% and SS patches with 23.26%. A Kruskal-Wallis test for comparison of the characteristics among the three groups shows for all 3 patch-related metrics a p-value $<$2.2e-10, meaning that at least one is significantly different from the others. Subsequent post-hoc tests show that all three are significantly different from each other. In other words, being asked by reviewers to write a second or later patch version is not a guarantee for success, but at least a good sign: reviewers see potential and really want to work on the patch.

5. THREATS TO VALIDITY

Threats to external validity concern the fact that we have only studied one large open source system. The Linux kernel is a prime example of a large open source project that uses its mailing list as reviewing environment. We cannot automatically generalize our findings to other projects, potentially supported by different reviewing mechanisms. Furthermore, since Git is a flexible version control system that can be used in various setups, we also need to take care of extrapolating our findings to other projects using Git or other version control systems. Hence, more case studies on other projects are needed.

Threats to internal validity concern the relationship between treatment and outcome. Since we did not study causal relationships, but quantitatively analyzed the characteristics of the reviewing history, we do not consider threats to internal validity.

Threats to construct validity concern the relation beween theory and observation. We link the different threads belonging to the same super-thread together. However, the three techniques that we applied are not perfect (precision and relative recall below 100%). Although we use the complementary results of two techniques to improve the precision and recall, and remove the outliers in the final results, the super-threads may not be the real super-threads, due to false positive noise. However, given the amount of email and commit data that we consider, we believe that the impact of noise is tolerable and that the resulting trends are valid.

6. RELATED WORK

Our work is mainly related to previous studies on the code review process of on open source projects, traceability, mining unstructured log data and clone genealogies.

Rigby et al. [25] analyzed the code review process in the Apache open source system. They found that small, independent, complete patches are faster to be accepted. In our paper, we found that the patch versions in a super-thread are examples of patches that take longer to be accepted. Since they seem to be larger and more invasive, this confirms Rigby et al.'s findings

Bettenburg et al. [6] found that accepted contributions are on average 3 times larger than rejected contributions, however, we did not study this phenomenon here.

Rigby et al. [24] studied the peer review process of a traditional inspection of a Lucent project and six open source projects. They found that smaller changes take shorter review time and the time from the start of the review to the end of the review was on the order of weeks. Both of the findings are verified in our paper.

Mcintosh et al. [22] studied the impact of modern reviewing environments (Gerrit [17]) on software quality. The impact is assumed to be due to a large amount of freedom that reviewers have when reviewing at their moment of choice via a website. It is not clear if low-tech reviewing environments based on emails have the same issues.

Jiang et al [19] studied the integration process of the Linux kernel project. However, they only considered the reviewing history inside the current thread and did not consider the possibly "hidden" super-thread reviewing history. To our knowledge, the current paper is the first to study the whole patch reviewing process across multiple threads in a low-tech reviewing process, we also analyze the impact of different thread types on the acceptance probability and the characteristics of different thread types.

There has been a lot of research on traceability between source code and software-related documents [1][13][14]. However, our work is the first to trace the commit back to its origin to figure out its rationale. Bettenburg et al. [10] proposed an approach to link email discussion text to source code fragments based on clone detection technique. This is one of the three techniques in our work. Bettenburg et al. [7] developed a tool for mining the structural information from natural text bug reports. Bacchelli et al. [3] proposed an approach to classify email content into different source code artifacts at the line level. Since Git patches have a very rigid format, we did not need to use there techniques to identify the Git patches in emails.

Our work is also related to the research on the vast body of code clone genealogies during the software evolution process [8][12][20][21], since those also track code (changes) across time. Our work combines clone genealogies with the reviewing process in a low-tech environment to help developers and reviewers understand the design rationale of patches.

7. CONCLUSION

Tracing the reviewing process of a patch to figure out its rationale plays an important role in software maintenance. However, a low-tech reviewing environment is still commonly used in many open source systems. In our study of the Linux kernel project, the reviewing and development process are separated and furthermore, a patch may vary across multiple versions and threads. In this paper, we propose an approach to recover the full lifecycle from commits to emails and analyze the characteristics of different types of reviewing threads. We found that the plus-minus-line-based technique is the best to link (email, commit) pairs, while the combination of the plus-minus-line-based technique and the checksum-based technique works best for linking (email, email) pairs.

As a first application of the recovered email to email links, we quantitatively analyzed the reviewing process in a low-tech environment and found that around 25% of the patches have a reviewing history of more than four weeks. Patches with multiple versions seem to be larger and impact more files, while they are spread across multiple threads if they contain bug fixes and long time has passed since the previous patch. Having to submit additional patch versions is not a disaster, since relatively more such patches are accepted than for patches with just one version, and reviewers keep on being interested in subsequent versions.

Our work opens up the possibility to compare low-tech reviewing environments head-to-head with modern reviewing systems. We consider this to be our future work.

8. REFERENCES

[1] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proc. of the 31st Intl. Conf. on Software Engineering*, ICSE '09, pages 298–308, 2009.

[2] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 712–721, 2013.

[3] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *Proc. of the 34th Intl. Conf. on Software Engineering*, ICSE '12, pages 375–385, 2012.

[4] O. Baysal, R. Holmes, and M. W. Godfrey. Mining usage data and development artifacts. In *Intl. Working Conf. on Mining Software Repositories (MSR)*, pages 98–107, 2012.

[5] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In R. LŁmmel, R. Oliveto, and R. Robbes, editors, *WCRE*, pages 122–131. IEEE, 2013.

[6] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German. Management of community contributions a case study on the android and linux software ecosystems. *Empirical Software Engineering.*, 2013.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proc. of the Intl. Working Conf. on Mining Software Repositories*, MSR '08, pages 27–30, 2008.

[8] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at the release level. *Sci. Comput. Program.*, 77(6):760–776, 2012.

[9] N. Bettenburg, E. Shihab, and A. E. Hassan. An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In *Proc. of the 25th IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 539–542, 2009.

[10] N. Bettenburg, S. W. Thomas, and A. E. Hassan. Using fuzzy code search to link code fragments in discussions to source code. In T. Mens, A. Cleve, and R. Ferenc, editors, *CSMR*, pages 319–328. IEEE.

[11] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *Proc. of the 4th Intl. Workshop on Mining Software Repositories (MSR)*, page 26, 2007.

[12] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. of the 29th Intl. Conf. on Software Engineering (ICSE)*, pages 158–167, 2007.

[13] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grunbacher, and G. Antoniol. The quest for ubiquity: A roadmap for software and systems traceability research. *Intl. Requirements Engineering Conference (RE)*, pages 71–80, 2012.

[14] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic. *The Grand Challenge of Traceability (v1.0)*, pages 343–412. Springer-Verlag, 2012.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proc. of the 34th Intl. Conference on Software Engineering*, ICSE '12, pages 837–847, 2012.

[16] http://bits.blogs.nytimes.com/2014/04/09/qa-on-heartbleed-a-flaw-missed-by-the masses/. Q. and a. on heartbleed: A flaw missed by the masses.

[17] https://code.google.com/p/gerrit/. Gerrit code review.

[18] http://www.itl.nist.gov/div898/handbook/. Nist/sematech e-handbook of statistical methods.

[19] Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast? – case study on the linux kernel. In *Proc. of the 10th IEEE Working Conf. on Mining Software Repositories (MSR)*, pages 101–110, 2013.

[20] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR*, pages 1–5, 2005.

[21] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proc. of the 29th Intl. Conf. on Software Engineering (ICSE)*, pages 106–115, 2007.

[22] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proc. of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, 2014.

[23] H. Pickens. http://linux.slashdot.org/story/13/10/09/1551240/the-linux-backdoor-attempt-of-2003. The Linux Backdoor Attempt of 2003.

[24] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, 2013.

[25] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proc. of the 30th Intl. Conf. on Software Engineering (ICSE)*, pages 541–550, 2008.

[26] Wikipedia. http://en.wikipedia.org/wiki/f1_score.

[27] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.