2016 IEEE 27th International Symposium on Software Reliability Engineering

Switching to Git: the Good, the Bad, and the Ugly

Sascha Just*,Kim Herzig†,Jacek Czerwonka†,Brendan Murphy‡
*Center for IT-Security, Privacy and Accountability, Saarland University, Germany · sascha.just@cispa.saarland
†Microsoft Corporation, Redmond, USA · kimh,jacekcz@microsoft.com
‡Microsoft Research, Cambridge, UK · bmurphy@microsoft.com

***Abstract*—Since its introduction 10 years ago, GIT has taken the world of version control systems (VCS) by storm. Its success is partly due to creating opportunities for new usage patterns that empower developers to work more efficiently. However, the resulting change in both user behavior and the way GIT stores changes impacts data mining and data analytics procedures [6], [13]. While some of these unique characteristics can be managed by adjusting mining and analytical techniques, others can lead to severe data loss and the inability to audit code changes, e.g. knowing the full history of changes of code related to security and privacy functionality. Thus, switching to GIT comes with challenges to established development process analytics. This paper is based on our experience in attempting to provide continuous process analysis for Microsoft product teams who switching to GIT as their primary VCS. We illustrate how GIT's concepts and usage patterns create a need for changing well-established data analytic processes. The goal of this paper is to raise awareness how certain GIT operations may damage or even destroy information about historical code changes necessary for continuous data development process analytics. To that end, we provide a list of common GIT usage patterns with a description of how these operations impact data mining applications. Finally, we provide examples of how one may counteract the effects of such destructive operations in the future. We further provide a new algorithm to detect integration paths that is specific to distributed version control systems like GIT, which allows us to reconstruct the information that is crucial to most development process analytics.**

***Index Terms*—mining software repositories, process analysis, version control systems**

1. Introduction

A bit more than ten years ago, Linus Torvalds released a new version control system called GIT. At the time, GIT was a reaction to a disagreement on the future of *BitKeeper* [1], a distributed version control system then in use by the Linux kernel group. Linus Torvalds' goal was to develop a version control tool with an emphasis on speed and one that provided support for distributed, non-linear work flows [38] capable of handling kernel size projects. Although, initially not meant to be a general purpose tool, the concepts GIT provided were generic and enabled faster and more agile development processes. The result is that GIT has become one of the most widespread version control systems, not only in the open source community but also in industry. GIT's functionality is allowing developers to branch often, work offline more efficiently, and have a larger set of more powerful commands at their fingertips than ever before. As a result, those software developers that use it, change their behavior. And like all fundamental development process changes, switching version control requires adjustment and re-validation to data mining and development process analytic techniques. Changing the development processes without updating and adjusting the corresponding data analytics may lead to imprecise recommendation tools used to guide multi-million dollar decisions.

Mining version archives is a well-researched and active domain that can be used to provide empirical data to support development tools and development process decisions. Tools leveraging and mapping artifacts collected from software repositories provide insights into development processes and are used, for example, in predicting code quality [11], [12], [40], helping to identify code owners [16], [18], [29] and code reviewers [5], [27], [34], guiding software testing [3], [20], [22], etc. And this trend of mining development repositories has found its way into industry [2], [9]. MICROSOFT has its own mining tool, called CODEMINE [9], which constantly mines and stores various development artifacts, collecting data, of nearly all major products, among them: WINDOWS, OFFICE, and BING. Empirical data is used to provide development process insights to engineering teams to help them build or improve development tools or to support development process decisions.

With the rising popularity of GIT, data miners and data analysts have to adapt to GIT and continue supporting teams in achieving their product development goals. However, for analysts, switching to a different VCS is a challenge as it changes not only the information content of the artifacts extracted from the VCS, but more importantly their interpretation which heavily depends on the context in which these artifacts were created. Typically, data analysis must define key measurements based on the understanding of the current development process. For example, how people interact with tools or how and when they merge changes, is all encoded in the interpretation of the data into a measurement. A fundamental change to the process, such as a move to a completely new model of storing code changes, requires at least modified and often brand new data interpretation. As a consequence, after moving teams to GIT, we may need to change the definition of key development process measurements, such as code quality or development speed.

Earlier studies [6], [13] described how GIT, its usage, and its internal storage mechanisms, impact data mining and data analytics procedures. Some of GIT's unique characteristics can be managed by adjusting mining and analytical techniques. However, not all of GIT's features and usage patterns are that easy to overcome when

This study was performed during Sascha Just's internship at Microsoft Research in Cambridge, United Kingdom.

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.38

mining for its data.

In this paper, we share experiences we encountered at Microsoft while supporting teams switching from SOURCE DEPOT[1] and TEAM FOUNDATION SERVER [26] to GIT. Most of these teams use *advanced development process analytics* helping them guide their daily development process decisions with empirical data and evidence. We start this paper discussing the most common reasons for switching to GIT (Section 3), describe at a high-level how GIT stores its data internally and how teams use GIT to manage their daily development activities. Based on this knowledge we then switch perspective and look at GIT and its usage from an analytical point of view and identify challenges in Section 4 before discussing commonly used but potentially harmful GIT operations in more detail (including some hints on how to minimize data loss) in Section 5. By way of example, in Section 6 we describe challenges encountered when trying to precisely resolve code change integration paths and a new way to extract them from a distributed version control system such as GIT. We close the paper discussing related work (Section 7) and providing conclusions (Section 8).

2. Git Overview

GIT is one of the most popular version control systems (VCS) today and this is not by accident. Although not the first distributed VCS, GIT gives its users the freedom to use the VCS as they seem fit and it strongly supports agile development practices. From a technical standpoint, GIT, its way of storing data, and the usage capabilities are well designed and considered state-of-the-art. Coming from a centralized VCS such as SOURCE DEPOT or TEAM FOUNDATION SERVER, GIT is a small revolution and immediately changes how users interact with a VCS [7], [33], [35]. This is why the open source community and industry are upgrading older VCS to modern ones, like GIT, providing opportunities for teams to increase agility. Some of the benefits of distributed VCS that make GIT attractive for developers are [28]:

Easy and cheap creation of branches.
: Most development teams manage parallel development activities with branches. Large development teams using a single branch are rare as it requires stable and independent components. Like in most distributed VCS, GIT makes heavy use of branches—even local checkouts on developers' machines are referenced as local branches. Unlike older VCS, it is easy to create branches to ensure work isolation and to merge the changes later. Local branches are also very helpful for engineers working on different parallel tasks.

Allowing to commit and revert locally.
: This is especially important for isolation. It allows developers to commit more often, to create private local restore points and combined with branches, this might be the single most powerful benefit for most developers.

Merge operations at fine level of granularity.
: Providing the ability to *cherry pick* individual *hunks* rather than entire commits. Although such operations are challenging for data miners (see Section 5.3), they provide developers the freedom to accept and reject individual changes rather than batches. This feature also allows for speedy detection of defect-inducing changes and provides the ability to precisely revert them.

These features enable developers to work the way they want rather than being forced to work a particular way due to the restrictions of a tool. Teams that use traditional version control systems such as SUBVERSION or SOURCE DEPOT are missing out on features that allow for fast, decentralized, and more agile development processes. And although there are concerns about the scalability of GIT [15], [32], more often than not, switching to GIT empowers software developers to work more flexibly and efficiently.

2.1. GIT's Artifact Store

GIT is able to support the aforementioned operations in part due to how it chooses to implement its artifact storage. Code histories contained in VCS can grow large–to millions of code changes over years of development history. Scanning the history entirely or even partially can lead to bad performance, especially for common tasks. Operations such as recording a new code change or merging code changes should require minimal effort as these are the operations that are most commonly used by VCS users. In this section, we will provide a short overview of this GIT artifact storage as later sections will rely on these concepts.

While a typical user of a VCS might not be interested in the mechanisms used by their repository to perform its internal accounting, from a data miner and data analyst perspective it is important to understand the following concepts as they may impact your ability to deliver the data sets and measurements the engineering team needs (see Section4).

GIT Objects. GIT operates on four different object types: BLOB, TREE, COMMIT, and TAG. Every object has a unique ID, which is the SHA1 hash of the object (its header and its content). The header consists of the type of the object followed by its size in bytes and a \0. The ID of an object is used as its filename for easy access. Objects are stored in the GIT directory at their corresponding file path `objects/$ID[0..1]/ID[2..39]`[2].

BLOB	contains the content of a single file.
TREE	represents a directory tree. The content consists of a series of tree entries, each representing either a BLOB or another TREE object. Every entry contains the file permission octal, the name of the file/directory, and the GIT object ID.
COMMIT	represents a snapshot in content history. A commit references a TREE object, all its parent commits, information about the author (name, email, and timestamp), and the same information about the committer. Additionally, a commit object holds arbitrary data, commonly used as the commit message.
TAG	is used as a point of reference for any GIT object. Tags contain the ID and the type of the object they refer to, the name of the tag, and information about the "tagger" (name, email, and timestamp). They also

1. SOURCE DEPOT is a Microsoft internal version control system.

2. GIT also compresses objects in pack files, see https://git-scm.com/book/en/v2/Git-Internals-Packfiles

hold arbitrary data, commonly used to describe the purpose of the tag.

GIT References. References are pointers to GIT objects or other references. They are used as aliases. The name of the file denotes a name of the reference. The content is either the ID of a GIT object or the path to another reference. GIT uses references for HEADS, REMOTES, and TAGS. Heads reference the latest commit in a branch in the local clone. Remotes refer to the heads of branches in remote repositories.

The GIT DAG. The history of the project is given by a directed, acyclic graph that can be constructed using the objects described above. When accessing the history of a branch, one starts at its head, given by the branch's reference, and traverses backwards using the parents of a commit. Adding new commits to the history is efficient as it does not require traversing the graph structure and only creates a new commit object containing the IDs of its parents. Figure 1 shows a simple example of such a GIT DAG.

3. The Good: Git and Continuous Integration

Typically, development teams work on several tasks simultaneously. To provide the necessary isolation to allow such parallel development, VCS use the concept of branches. In order to ship software, we have to converge all parallel development activities, so that we can build, verify, and ship a single code base as a product. In a perfect world, different developers and teams would work on separate parts of a perfectly componentized product so that their parallel changes never overlap or conflict with one another. In reality, however, conflicts between parallel applied changes are common and may even cause severe problems that may not be resolved automatically and thus cause build or test breaks, or even issues in the field.

Considering that conflicts will occur during software development, VCSs provide features for enabling parallel development activities (branches) and their eventual convergence into a single branch (merging) for release. At the highest-level, therefore a development process is about managing how to make changes in branches and integrating these changes into a single release.

While these concepts are not new and were supported by earlier VCSs, modern software development processes strive for agile methods and faster integration processes. However, being agile implies being flexible and thus requires better isolation between developers to guarantee independence from each other's changes. Consequently, modern, agile development processes require flexible and distributed VCSs, such as GIT. Therefore, GIT seems to be the perfect fit for teams striving for continuous integration.

3.1. Branching and integrating changes using Git

In GIT, a branch is a foundational mechanism. Every local copy of a version archive is managed through local branches. Remote branches are visible to users with access to the remote repositories. Checking out a branch, making a local change, and committing it leads to a local commit, not yet visible to others. That only happens after pushing changes to a remote branch in a shared remote repository.

Pushing a committed change from one branch to another causes *integration events* (green arrows in Figure 1). The underlying GIT operations for this are *integration operations*. At the user level, GIT provides several types of integration operations (we will discuss some of them in Section 5). From a data mining perspective, the type of integration operations used determines how much of the vital information about the development process–such as data on which teams work on the same artifacts, where do they most often conflict, and where conflicts are not auto-resolved–is preserved.

Developers typically prefer "flat" VCS histories, e.g. a DAG reduced to a simple sequence of commits. Flat histories provide excellent opportunities to find defect inducing code changes and to revert them. Thus, teams often set policies enforcing integration strategies and specific GIT commands to be used. The selection of preferred integration strategies and integration operations is thus determined by the team's development process.

However, data miners and data analysts should consider flat histories as data loss, and so should teams that depend on data analytic applications relying on version control data. Integration strategies and operations used to achieve flat histories often drop potentially important information causing tangled changes [25], [21] or removing any provenance data i.e. information on where a change came from and what its path was to the release branch. We discuss some of these operations in more detail in Section 5.

Data miners prefer integration operations that preserve as much information as possible. From a data miner's perspective, merging two or more parallel commit sequences should ideally result in an explicit merge commit.

For example, in order to merge commit 3 initially added to branch B/y into branch B/x, a merge commit 4 should be created which references commit 2 as branch parent and commit 3 as merge parent. Such integration operations reflect exactly what happened and transparently show that two changes were applied in parallel and that commit 3 has not been directly committed to branch B/x but had to be integrated.

4. Analytical Challenges (The Bad, Part I)

Despite tremendous benefits to the user, GIT's artifact storage described in Section 2.1 has some disadvantages. Data miners often count on having access to the full history of developer activity and that is counter to GIT's focus on optimization of the store data. As a consequence, mining tasks become harder compared to other repositories and may require rethinking of at least some definitions, metrics, and mining techniques.

For example, consider a need to measure the agility of your development teams. One factor to consider is *code velocity*—the time between completing the implementation of a code change to its release to the user (a more complete definition and example is given in Section 6.1). Independent of the precise definition of code velocity, such metric requires having the ability to trace code changes shipped in a product back to the original commit(s) pushed by a developer and comparing timestamps. Being able to extract the information from your current VCS depends on the definition of branches in that VCS. As GIT's definition of branches is very different to other VCSs, it changes the velocity

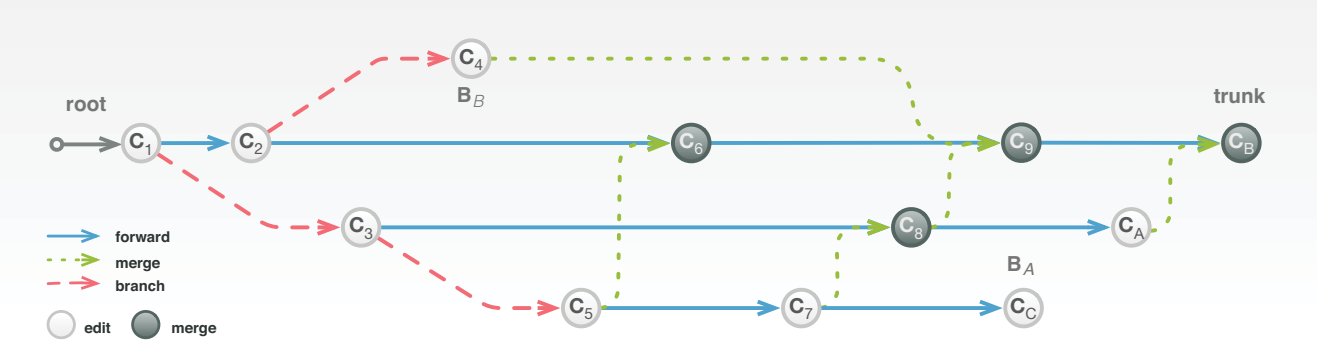

Figure 1. An example for a DAG representing a GIT repository.

calculation procedure. As a consequence, switching to GIT requires a re-validation and rethinking of, sometimes very basic, concepts we rely on when mining and interpreting software development artifacts. At the same time, we need to:

Ensure to measure the right thing.
: Since the definition of our metrics rely on information embedded by tools that created the interpreted artifacts as well as on the precise way developers use these tools, we need to ensure that the current definition is independent of the VCS. For a transition to GIT, this means a need to fully understand GIT's artifact storage and how it operates on the artifacts.

Ensure to measure the same thing as before.
: The new definition of a measurement must match the old definition with respect to the higher level objective we want to measure. Changing the measurement objective and simply re-using the old name of the metric will lead to confusion or invalid comparisons, e.g. measuring code quality before in terms of pre-release and afterwards in term of post-release defects. This aspect is especially important, as we need comparable measurements in order to determine whether switching to GIT had the desired effect on the development process.

Therefore, as data miners and data analysts, we need to understand how GIT stores its artifacts and how it operates on them. It will determine how we define measurements as well as mine and interpret development artifacts. In our experience of supporting teams switching to GIT from SOURCE DEPOT or TEAM FOUNDATION SERVER, we noticed some common issues and usage behaviors that are likely to influence many data analysis applications, especially with respect to development process analytics. The goal of the following sections, is to provide an overview of these issue themes and to elucidate how common GIT usage patterns affect data mining and analytic capabilities.

The provided list of issues is by no means complete, merely providing examples of how some GIT commands can introduce data mining challenges through rewriting past events, or even loosing historical information such as commit authors or entire branch information. How frequent such operations are performed is controlled by a teams' GIT process and development policies. For large teams performing complex changes and merges, we found that many of these operations are performed frequently enough to cause serious concerns about the accuracy of data and measurements derived from the corresponding GIT archives.

Where possible the sections will identify the GIT commands that can cause the greatest data loss if used without care. We will also try to identify mechanisms minimizing the loss of data while ideally not restricting the freedom of the developers to use the VCS as they see most fit for their work flow.

5. GIT Data Loss (The Bad, Part II)

To estimate the frequency of the operations discussed in this section, we mined 9 open source GIT repositories, using heuristics to estimate the number of operations per type. By definition, the data provided in Table 1 is incomplete and should be considered an estimation as many operation types or individual occurrences are not detectable (hence the need for this paper). We discuss the criteria used in this table in subsection 5.8.

5.1. Fast-Forward Integration

What is it and what happens?. Fast-forward integrations occur when changes on one branch are integrated into another and no merge is required, i.e. one branch is strictly ahead of another. Consider the below example in which the branch `B/y` is ahead of the branch `B/x` and thus can be fast-forwarded (appended) to the head of the latter.

To merge the two branches, GIT will reset the pointer (`B/x`) to the same object as (`B/y`). Once the merge operation is over, the two branches are synchronized—both branches have identical GIT histories. The fact that change 1 and change 2 were initially committed to different branches is now lost as GIT does not explicitly bind commits to a branch. Fast-forward integrations occur when the consuming branch (`B/x`) did not diverge from the supplying branch (`B/y`). As a result, the two branches collapse to a flat history and information about the origin of the change (branch `B/y`) is lost.

How does it impact data analytics?. Fast forward integrations can be considered data preserving as no data gets rewritten.

TABLE 1. FREQUENCY OF GIT OPERATIONS. ALL VALUES ARE APPROXIMATIONS BASED ON HEURISTICS FOR DETECTING INTEGRATION TYPES. MORE RELIABLE NUMBERS ARE NOT AVAILABLE AS THESE INTEGRATION TYPES CANNOT BE MINED OR DETECTED.

	Android*	Apache†	BuildBot	CoreCLR	CoreFX	Eclipse	Git	Roslyn	Strider
Days of GIT history	2508	652	1411	346	430	1257	3931	664	1693
# Contributor aliases	16,490	13,221	648	263	312	4607	1868	246	138
Merges	$> 754k$	$> 57k$	$> 2k$	$> 1k$	$> 2k$	$> 17k$	$> 10k$	$> 3k$	> 600
Direct edits	$> 1,245k$	$> 219k$	$> 11k$	$> 1k$	$> 4k$	$> 950k$	$> 33k$	$> 8k$	$> 4k$
Rebase	$> 71k$	$> 51k$	> 700	> 250	> 550	$> 25k$	> 150	$> 1k$	> 80
Apply	$> 5k$	$> 5k$	> 300	> 20	> 20	$> 9k$	> 550	> 20	> 30
Squash	$> 5k$	$> 10k$	> 70	> 20	> 50	$> 5k$	n/a	> 40	> 20
Wrong timestamps	$> 99k$	$> 41k$	$> 1k$	> 50	> 90	≥ 0	$> 7k$	> 150	> 90

*Android Open Source Project repository (AOSP) †The entire Apache collection (incl. 630 projects)

Author, committer, timestamps, the message and the patch itself remain unchanged. Yet, using fast-forward to integrate changes between different, non-tracking branches will make it impossible to determine to which branch a change was originally committed to. Impacting all analytical applications that rely on such commit branch information, e.g. understanding which developers are working together, identifying which changes are developed in shared branches etc. Fast-forward integrations are not detectable as no metadata is found even in commit messages. Therefore, we were not able to estimate how often these operations occur. However, from our experience fast-forward integrations are common and happen on a daily basis.

How to minimize impact on data analytics?. Similar to nearly all integration types and merge operations, the only way to identify the lost branch information per commit object is to store metadata about the commit. For merge commits, many GIT clients auto-populate the commit message with the following text:

```
Merge branch 'development' [...] into 'master'.
```

GIT clients can provide that text at the time of the merge: both branches still exist and the merge operation itself needs both references in order to perform the DAG modification. However, many GIT users delete or overwrite such auto-generated parts of commit messages as it makes commit messages less readable and appear to be unnecessary from a user's perspective. Therefore, teams that use an analytical application that rely on the ability to track changes, should include metadata in commit messages, the easiest method, requiring no input from developers is to write the data using an automated procedure, for example by using a commit script or a check-in wizard.

There is also the option to explicitly turn off fast-forward merges using GIT's `--no-ff` option.

5.2. Rebase Integration

What is it and what happens?. Rebases are another way of integrating changes that would otherwise require a merge.

Updating the (local) branch `B/y` to a newer head avoids a merge and allows flat, fast-forward integration. In the background, the patch applied in commit 3 is rewritten so that it now can be applied to commit 2 rather than 1. The modified patch is then stored as a new commit object ($3'$) which replaces the original commit 3 as a new head of `B/y`. This type of integration is a common practice to keep the history as flat as possible–especially when working with a sufficient number of engineers in one branch. It keeps the DAG narrow which makes it easier to understand and track sequences of changes.

How does it impact data analytics?. Rebase integrations change the actual patch and might be considered harmful, depending on the data analytics application.

A prominent analytics example that is likely to be impacted are code quality measurements which rely on churn data to identify code entities changed in a defective commit. Modifying the patch during a rebase results in the measured churn information not necessarily reflecting the true churn caused by the developer. Thus, we may mark code entities as defective that were never changed due to fixing a bug, or vice versa. This can have consequences for models relying on this data like defect prediction models [21].

Apart from churn information, no other information is lost. In the vast majority of cases rebasing does not significantly harm process analytics, even those that depend on churn information, as long as the information about the author and event timestamps are preserved. While the topological order of changes applied to the DAG might not match the actual sequence of events, the timing information is preserved.

In addition to integrations of local changes into remote branches, rebases are used to integrate changes between remote branches, as in merging changes from the development to the release branch. Besides rewriting the actual patch, the modified change will only be reachable from the head of the target branch and thus appear as a direct edit into the target branch obscuring the fact the code change has much richer history than this.

How to minimize impact on data analytics?. As mentioned above, the impact of rebases on local branches is likely to be minimal. Yet, rebasing changes onto other branches affects patches and branch associations. Other than preventing cross-branch rebase integrations, there is little that can be done to minimize the data loss. Additional metadata that would need to be stored would be the original change(s)/branch(es) from the merged commits; enough information to restore the actual churn data—an effort likely to not be worth the return of investment. From Table 1, we know that rebase integrations are very common. For Android, we detected 71,000 rebases, that is 28 rebases per day, in Apache repositories rebases are even more popular (78 rebases/day).

```
From 99d9721 Mon Sep 17 00:00:00 2001
From: John Doe <john.doe@domain.com>
Date: Mon, 25 Jan 2016 19:51:36 -0800
Subject: [PATCH] Edits to foo.h.

---
 foo.h | 36 ++++++++++++++++++------------------
 1 file changed, 18 insertions(+), 18 deletions(-)

diff --git a/foo.h b/foo.h
index 992cc49..465abeb 100644
--- a/foo.h
+++ b/foo.h
@@ -526,25 +526,25 @@
[...]
```

Figure 2. Example patch created using `git format-patch` which preserves the original author information and the timestamp.

5.3. Apply Integration

What is it and what happens?. Apply integration simply takes a patch and applies it to the files in the repository.

This type of integration happens frequently between local branches and even across different repositories and is used to, for example, perform code reviews. In the above example commit 2 is exported as plain text diff, sent via email to another developer who simply applies the patch to her local repository. This results in no information about the original author or creation time being preserved. Committing these "applied" changes by the second developer would be the equivalent to her manually performing the edits.

How does it impact data analytics?. This form of patch submission destroys valuable data, as there is no way to detect the origin or the author of the patch. If performed frequently this may have severe consequences for analytical applications which cluster commits by the author or simply to tracking provenance of code (due to legal obligations or otherwise). A detailed view of how this type of integration impacts data mining can be found in a study by German et al. [14]. Apply integrations seem to be even more popular than rebases. Using those depends on the development process and the level of interaction between team members. Nevertheless, the data in Table 1 shows that this is a factor that should not be ignored if the analytical application depends on the precise authorship information.

How to minimize impact on data analytics?. Instead of exporting changes or applying a plain patch to a branch, one should use GIT's `format-patch` and `am` commands. GIT `format-patch` will still export a patch suitable for sending via email but with information about the original author and timestamps (see Figure 2).

To apply such changes one should use `git am` which accepts patches created by `format-patch` and applies them sequentially preserving important information about the actual author and timestamp the original changes were applied.

5.4. Squash Integration

What is it and what happens?. By themselves, squash commits are not a separate type of integration in GIT. Instead, squash commits are used to combine multiple separate commits into a single, new commit featuring a combined, descriptive commit message. There are different ways to perform squashing in GIT (e.g. using merge, rebase, reset) leading to different side effects. Developers can preserve the original commits or have them removed from history. Commit messages can be a combination of the commit messages of all the squashed commits and might even refer to the original hashes. Due to space limitations, we discuss the different ways to squash changes in our blog post [24].

Many teams use squash commits to produce simpler GIT repository log graphs. Having a GIT history that corresponds to a flat series of commits has advantages when reverting changes or identifying code defect-inducing fixes.

How does it impact data analytics?. The disadvantages of squash integrations depend on the GIT process and your analytical needs. In cases where teams use GitFlow [10] as GIT workflow, the feature branches are deleted after an integration. If this is done by squashing, data analysts lose any chance of analyzing the individual commits, even the author and timing information are lost. If other processes are used which prevent the deletion of branches, analytical processes need to be able to handle multiple counts of squashed commits.

In general, squash commits increase the probability of tangled changes [25] and may harm analyses that require finely granular change data, such as defect prediction models [21] or which detect churn patterns such as entities frequently changing together [39].

Although only detectable if indicated through metadata in commit messages, the number of squash commits we detected in the mined repositories (Table 1) is considerable. Roughly 2 squash commits per day for Android, 15 per day for Apache repositories, and 19 per day for Eclipse. Please note that these numbers only reflect the number of final squashed commits. The actual number of commits that were squashed together is not retrievable.

How to minimize impact on data analytics?. From our experience, there is no perfect solution to data loss caused by commit squashing. Squashing a series of local fix attempts might even help analysis, as it combines partial fixes into a complete fix; in GIT, this is known as *fixup*. However, if feature implementations are being squashed, analysis relies on preservation of the original commits and metadata that links those commits to the squash commit.

5.5. Reverting changes

What is it and what happens?. Reverting is the removal of a change that has already been committed or integrated into a branch. Reverts occur very frequently and cause many mining issues. Improper revert operations impact mining and analysis across all VCS, not just GIT. Thus, we included it in this paper.

How does it impact data analytics?. Reverting changes is not bad practice in general. Issues may arise however when reverting parts of changes or single chunks. This may lead to tangled changes [21], [25]—specifically commits that combine changes serving multiple purposes—and complications in tracing changes to their origins. Similar to squash commits, tangled changes may impact defect prediction models [21] or churn pattern detection algorithms. We were not able to retrieve data for reverts as reverted changes usually disappear from remote histories.

How to minimize impact on data analytics?. For data analytic purposes, we recommend to revert entire commits and never partially revert changes. Even if only a subset of changes needs to be reverted, revert the entire commit and re-commit parts that needed to be kept. Many GIT clients use custom commit messages to indicate reverts.

```
Revert "fix for bug [...]"
This reverts commit c71cdce[...].
```

As for merge commits, this metadata should be machine readable and kept in place or stored in an external database. Miners and data analysts can leverage the data by parsing and interpreting the commit messages.

5.6. Timestamps

The third common mining issue that arises in nearly all GIT repositories we investigated so far are related to author and commit *timestamps*.

What is it and what happens?. We observed two common issues in GIT repositories that can severely affect the usefulness of GIT timestamps.

The first issue are unsynchronized clocks. The fact that GIT is a decentralized VCS means that all timestamps are local reflecting the clock time on the machine hosting the GIT repository, most often: the developer's desktop. Improper configuration of machine clocks can lead to time differences between machines ranging from a few seconds to minutes, or even involve wrongly configured time zones. For example, a case of a virtual machine in a data center in Europe using a US based time zone.

A more difficult issue is the manipulation of timestamps using GIT's configuration settings. Namely, environment variables `GIT_AUTHOR_DATE` or `GIT_COMMITTER_DATE`. Assigning constant values to these variables hides the time at which work was committed. This is not an uncommon scenario. Nearly all repositories we mined contained a substantial number of commits that contain suspicious timestamps like `2078-01-01 00:00:00`.

How does it impact data analytics?. Many mining techniques and heuristics make use of timestamps to determine the order of operations. A prominent example are code quality metrics that use the order of commits and bug reports to map code changes to defects [40] or use the time gap between changes as an indicator for the change intention [30].

Using timestamps for such purposes assumes they are correctly captured and formatted, and reflect the correct time zone. If these assumptions are violated, timestamps must be assumed unreliable, or at least noisy, sources for data analysis. Simple tasks such as separating pre- from post-release fixes may be affected when relying on raw stream of timestamps without some form of preprocessing.

To estimate the number of obviously false timestamps, we counted the number of commits with timestamps before 1980, after 2016, or which had an author timestamp larger than the commit timestamp. The values are shown in Table 1. Although this is an underestimation of the true extent of the issue, there are more than 99,000 "wrong" timestamps in the nine repositories we looked at. Most of them refer to constant date "1970-01-12 14:46:40". For Android, this means around 40 commits per day contain wrong timestamp information.

How to minimize impact on data analytics?. Encourage developers to synchronize timestamps and prevent setting timestamp configuration variables to constant values. While most local developer machines will use OS-based time synchronization mechanisms, virtual machines frequently come with wrong time configurations like time zone settings. Those are easily preventable problems. In cases in which users seem to use constant timestamp values, data miners and analysts need to ensure that commit and author timestamps are treated as entirely unreliable for any kind of analysis.

5.7. Contributor Identities

What is it and what happens?. As a distributed VCS, GIT does have no single point of authentication. As a consequence, the commit author information such as name and email are not validated and may contain false information. Developers working on different machines, often use slight variations in author information such as different email addresses.

How does it impact data analytics?. Identifying developers can be crucial for some types of analysis, like deciding code ownership. Projects which need to comply with regulations might want to track code origination very precisely. Multiple identities for one developer may break analysis or falsify reports, especially in large repositories where names and email aliases may not even be unique over time.

How to minimize impact on data analytics?. If teams use a corporate or organizational login system such as LDAP or WindowsAuth, a check-in wizard could use the centralized user data to identify the user. However, this should be implemented in a way that it requires no Internet access, e.g. caching information on disk. Otherwise we would remove an important feature from GIT; being able to commit without network access.

Another possible solution to recover such identity information after the fact is to use heuristics to compare names, email aliases, machine identifiers, or even external information such as avatars to merge multiple identities to a single virtual identity. Using corporate authentication mechanisms may also help adding additional metadata (if not sensitive).

5.8. Data Collection

We derived our estimations by scanning commit messages for patterns indicating the operation type and investigating meta data. We computed the data as follows:

Merges and direct edits. Merge commits have two or more parents whereas direct edits only have once predecessor.

Rebases. To find potential rebases, we select all the commits from the database which are edits, not merges, i.e. the ones with only one single parent commit. Typically for rebases, we only take the ones having an author time smaller than the commit time. When rebasing, the commit was first done locally. Then, changes from remote that happened in the meantime are pulled in and the local commits are rewritten to a new head, effectively changing the commit timestamp. As this is the common scenario for rebasing, the incoming changes will have a different committer than the local ones. Hence, we also add this as a criterion. Furthermore, the commit we rebase onto has to have a commit timestamp which is smaller than the local commit that is rebased. Finally, we filter out everything that contains the following keywords: am, cherry, apply and patch. This leaves us with a lower bound for *in branch* rebases.

Applies. These changes will have equal author and committer information and equal commit and author time. We further assume the commit message to state that a patch has been applied and that it does not contain any GIT hashes. The latter is to filter out reverts and partial fixes to make sure we hit the lower bound.

Squashes. Detecting squashes per se is not possible. However, most GIT clients either explicitly state that a squash took place in the commit message or at least reference the squashed commit hashes. We exploit that fact and collect all commits mentioning *squash* in the commit message and the once referring to multiple commit hashes. We also include all individual changes that apply patches with more than 10k LOC for source code files.

Wrong Timestamps. We consider timestamps as wrong if one of the following criteria is met:

- the commit was created after it was committed
- the author or commit timestamp date prior to the start of the project
- the author or commit timestamp are in the future (after the point in time the respective repository has been synchronized)
- the commit timestamp of the commit where a change was integrated (merged) is smaller than the author time of the respective commit (the commit was integrated before the patch was created).
- the patch was created after its successor has been committed.
- two or more consecutive commits feature the same commit timestamp (from our experience, this is a result from overwriting timestamps using GIT's environment variables).

6. The Ugly: Integration Paths

As discussed in Section 5, there are GIT commands and concepts that will introduce noise and bias into data sets sourced from it. How large the problem is, depends partially on analytical needs and measurement definitions. Most of the discussed issues can be overcome either by collecting metadata about commits and applied operations or by adjusting usage behaviors and policies.

In this section, we describe–by way of example–how much extra work is sometimes needed for analysis to become viable again after moving from a centralized to a distributed repository. We use detection of *integration paths* and measurement of code velocity as our case study.

An integration path of a change C into a branch B is a path through the GIT DAG (see Figure 1) which starts with the original commit of C and ends with a commit which integrated C into B. From the example shown in Figure 1, there exist multiple integration paths of C_3 into the trunk branch: $\langle C_3 \to C_5 \to C_6 \rangle$, $\langle C_3 \to C_5 \to C_7 \to C_8 \to C_9 \rangle$, $\langle C_3 \to C_8 \to C_9 \rangle$, $\langle C_3 \to C_5 \to C_7 \to C_8 \to C_A \to C_B \rangle$, and $\langle C_3 \to C_8 \to C_A \to C_B \rangle$.

Integration paths are essential for many analytical approaches as they identify origin and paths of changes through time and branches. There has been little work on identifying correct integration paths. However, integration paths are essential for nearly all measurements related to development processes, in particular with respect to agility and speed, or to identify and avoid merge conflicts in large development environments. German et al. [14] recently published a study identifying integration paths across Linux kernel repositories and showed how much information is lost.

6.1. Example: Code Velocity

A prominent industrial example of a measurement that heavily relies on integration paths is development speed, or *code velocity*. There exist different ways to measure development speed [17], [19]. Irrespective of its precise definition, development speed is often used to estimate many important process attributes, e.g. to predict the date of the next release, better resource allocation, to identify bottlenecks in the workflow, to identify parts of the code or features at risk of delaying the release.

Here, we define code velocity of a change C and target branch B as the amount of time that was necessary to integrate C into B. Choosing the branch for the next release as B, code velocity is the amount of time between a code change completion by a developer and the change being ready to be released. Using integration paths, we can define code velocity of a change C to B as

$$\texttt{code velocity} = \min_{\forall p \in IP_{C,B}} \Delta_{time}(tail(p), head(p))$$

where $IP_{C,B}$ is the set of all integration paths from C into B, and $\Delta_{time}(tail(p), head(p))$ the amount of time between the integration of C into B, $tail(p)$, and the point in time C was first committed, $head(p)$.

Using our above example integration paths from C_3 into trunk using the DAG from Figure 1, the code velocity integration path is $\langle C_3 \to C_5 \to C_6 \rangle$.

This measurement of code velocity depends on the ability to identify all integration paths for all or the vast majority of code changes as well as on accurate timestamps (see Section 5.6). While the exact definition of code velocity may be changed, there seems to be no definition that would not rely on integration paths in any way.

6.2. GIT Integration Path Detection Algorithm

To counter the many idiosyncrasies of GIT requires an algorithm that reflects GIT's features and the common usage patterns described above. The algorithm design also attempts to avoid the issues of data loss or data imprecision discussed in Section 5. This is especially important with respect to timestamps. The presented algorithm to detect integration paths does not rely on timestamps

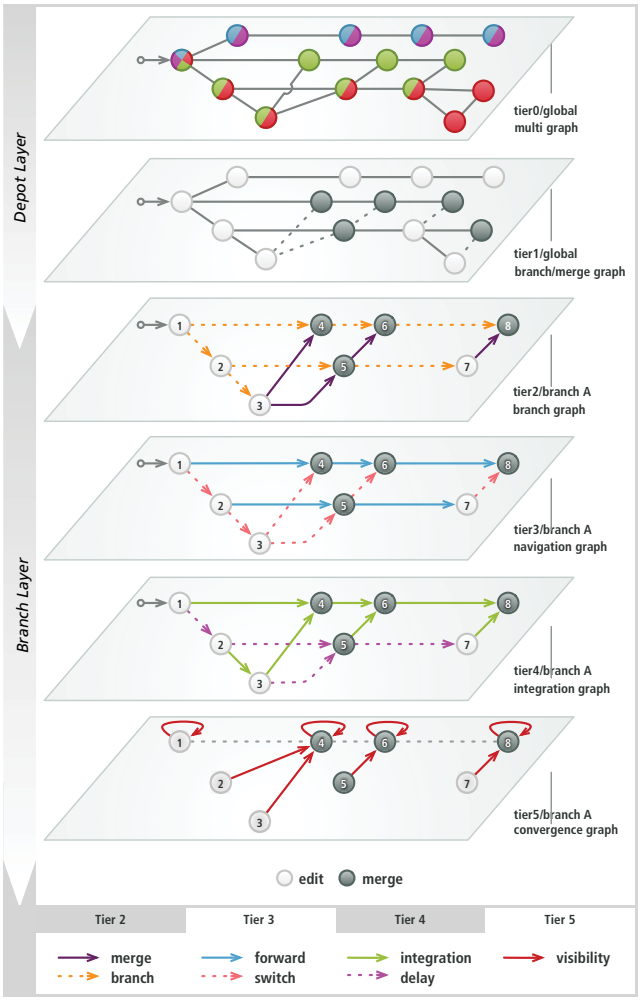

Figure 3. An example how to turn a raw GIT DAG into a convergence graph used to map the earliest point of integration in a branch.

at all. (Applications that use integration paths to measure process metrics like the aforementioned code velocity, will have to eventually use timestamps for computation of the final metric. However, timestamps are not necessary for the integration path detection algorithm described below.)

The core of our integration path algorithm is a transformation of the GIT DAG, peeling away layers of the DAG, disambiguating it to reveal the underlying information we can use for analysis. Figure 3 depicts an example of the multi-layer graph which eventually enables us to know when and where a change was integrated and how long it stayed in the individual branches.

Tier 0: the Global Multi Graph. We start with the complete DAG, i.e. the graph containing all commits reachable from any reference found in the repository. As shown in Figure 3 where every branch has its own color (purple, blue, green and red), a commit might be reachable from several references. At this level, we do not have any structural information besides nodes (commits) and the edges between them signifying a parent-child relationship.

Tier 1: the Global Branch/Merge Graph. In the next step, we label nodes as edits or merges. This information is important for analysis that treat direct edits differently from merges, as merges mostly only combine changes and fix incompatibilities between colliding patches. This information is static within the whole DAG. When a commit is created, it represents the state of the repository as the result of the changes applied to the combination of all its parents. If a commit is visible, i.e. it is a part of a branch graph, all its parents are visible as well.

Tier 2: the Specific Branch Graph. We introduce a virtual layer for each branch, labelling edges to be either *branch*-related or a *merge*. This information is required later, when computing integration paths–the algorithm requires knowledge of where merges happened and where they came from.

Tier 3: the Specific Navigation Graph. As we monitor code traveling through branches, local or remote, we require the information on when changes become visible outside their direct, fast-forward path. We start at the head of a branch and traverse backwards, following the direct parent of each commit to the root and label every edge along the way as *forward*. Every other edge leaving a node is considered a *switch*, as it is leaving the direct path. Once we reach the root, we follow the first switch and process the commits along its direct children performing the same operations as before until we've visited every node in the graph. These edge labels can be used to (partially) recover branch association, the information as to which branch a change was originally committed.

Tier 4: the Specific Integration Graph. The integration graph is another virtual branch-based layer which provides the ability to navigate the DAG to find the earliest point of integration for each commit, that is the shortest path to the topological earliest integration into the target branch. We label edges as either *integration* or *delay*. Following *integration* edges will lead to the earliest path of integration towards the corresponding branch graph.

Tier 5: the Specific Convergence Graph. When the earliest point of integration of a given commit into a branch is needed, as is when computing code velocity (see subsection 6.1), we need a mapping for each commit in each branch to its point of integration. The *visibility* edges provide a direct relation between a commit and its integration, i.e. the node in the DAG when a change first becomes visible to the target branch.

In order to allow us to determine integrations reliably, we compute a *convergence graph* layer on top of the raw GIT DAG. The process of computing the convergence graph by backwards traversing the main path of a branch, slicing it at every merge commit. We then start at the root and move forward mapping all direct and transitive parents we have not seen yet to the convergence mapping of the slicing commit. Thus, we get a mapping for the integration point into the target branch for every single commit, as depicted in Figure 3. Computing the integration graph requires performing this task on every sub-tree of every merge.

The graph layers described above work solely on the graph structure of the underlying GIT repository. Changes that are integrated, disconnected from the DAG, will still bias the analysis. For example, cherry-picking commits does not invoke any creation of edges in the DAG and requires additional metadata before it can be appropriately reflected by the graph.

7. Related Work

Bird et al. showed that GIT provides opportunities that may "lead to richer content histories" [6]. However, the authors also show that the internal concepts of distributed version control systems like GIT, may need different mining techniques and may lead to confusion when adapting mining concepts from other version control systems, such as SUBVERSION. Bird et al. explicitly state that "it is often not possible to tell what branch a commit occurred on" [6]—a problem that may have severe consequences as showed in this paper.

Rodriguez-Bustos and Aponte [35], Rigby et al. [33] as well as Brindescu [7] analyzed the impact of distributed version control systems on development practices, such as change frequency, average size of commits, etc. However, these studies did not investigate how much development teams relied on empirical data to guide development decisions nor whether the switch to version control systems like GIT threatened the ability to extract such data reliably and accurately from the new software archives.

Apel et al. [4] and later Cavalcanti et al [8] investigated how semi-structured merges of independently developed revisions can help to resolve change conflicts. Both studies focus primarily on the GIT user perspective helping developers to minimize merge conflicts. However, these and similar studies do not show whether these usage recommendations impact the ability to provide advanced data and process analytics.

Two studies show how complicated it can be to trace code changes through GIT repositories, especially when passed between different repositories. Jiang et al. [23] and German et al. [14] showed how much effort is needed to trace code changes through the Linux kernel developer repositories and how much data might be lost or be incorrect if the mining approach makes wrong assumptions or is too simplistic. Both papers are excellent examples how complicated software archive mining can be nowadays and how dangerous it is to make unverified assumptions about version archives.

Tarvo et al. presented an "integration resolution algorithm that facilitates data collection across multiple code branches" [37]. This algorithm is used at Microsoft but is based on centralized version control systems and in particular SOURCE DEPOT. One of the goals of this paper is to show how such algorithms must be adapted to different version control systems as they rely on internal storage concepts and usage behavior which changes when switching version control systems.

Negara et al. [31] investigated how incomplete and imprecise version control data is when used to study code evolution. Although the focus of this study was towards finely granular data to analyze code changes and their purpose, the authors show how confusing version archive data can be if used improperly.

Muslu et al. [28] conducted a survey and interviews with developers at MICROSOFT identifying benefits and barriers encountered by development teams when switching to a distributed VCS. The results show that the ability to commit frequently, do work and commit without network connection, and easy branching were the dominating factors in the transition to distributed VCSystems. However, the paper also investigated limitations of distributed VCS for large monolithic products and presented some guidelines for people wanting to transition. Muslu et at. [28] mainly focused on higher level benefits and limitations of distributed VCS for developer. In this paper we investigate the switch to GIT from a data mining and data analytical perspective and focused on detailed GIT usage behavior and its potential impact on mining and analytical applications.

With respect to GIT usage practices and behavior models, there is no silver bullet solution on how to use GIT in general. A good example is the discussion around the *GitFlow* branching model. While some users find this model successful in their particular settings [10], other strongly argue against it [36]. As with any other development process issue, solutions are context-dependent and require process analytics to identify practices that support the current development tool infrastructure and the software development culture within the team.

8. Conclusions

GIT has become a popular VCS due to its inherent focus on speed and on the strengths of its features. It affords development teams the flexibility of usage patterns and enables individual and team behaviors not previously easy to employ. Switching to using GIT as the primary source code repository is also under way at Microsoft. Such a fundamental change in the tool forces teams to change the workflows around it. At the same time, teams which use process analytics would like to preserve that capability as much as possible. Measuring one's own process productivity, speed, and quality are important to orderly process changes.

The way GIT defines commits and branches create a difficulty in re-applying existing definitions and metrics to the new tool and the new workflow. We outline several usage patterns which may alter or destroy vital historical information pertaining to code changes such as points of origin, timing and ordering. Many of the GIT usage patterns can be adjusted in such a way as to avoid or at least minimize the information loss.

In some cases, new algorithms or heuristics are necessary to extract the meaning from the information provided by the repository. We present, by way of example, an integration resolution algorithm which needed to be completely re-designed to account for how GIT stores and exposes historical code change information and which attempts to counter the effects of the data loss from GIT's most prominent usage patterns.

Nevertheless, in case of some usage patterns, the information loss is so severe, it cannot be recovered from.

In the end, if the need for more precise historical event log and the analysis it enables are common and deemed important for the community, changing the information stored by default in a GIT repository might become necessary. Until then the teams using analytics relying on strict timing of events or having a need to prove code origin, have to weigh in benefits and costs of using the data altering patterns. If needed, the development teams need to be educated and encouraged to change the tool's configuration or adjust policy as to minimize the information loss.

9. Acknowledgments

Sascha Just is funded by Microsoft Research through its PhD Scholarship Program. We also like to thank Varun Singh and Trevor Carnahan for their support and discussions.

References

[1] BitKeeper. http://www.bitkeeper.com/.

[2] Companies & projects using git. https://git-scm.com/, December 2015.

[3] J. Anderson, S. Salem, and H. Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 142–151, New York, NY, USA, 2014. ACM.

[4] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 190–200, New York, NY, USA, 2011. ACM.

[5] M. Barnett, C. Bird, J. Brunet, and S. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 134–144, May 2015.

[6] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, May 2009.

[7] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333, New York, NY, USA, 2014. ACM.

[8] G. Cavalcanti, P. Accioly, and P. Borba. Assessing semistructured merge in version control systems: A replicated experiment. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10, Oct 2015.

[9] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy. Codemine: Building a software development data analytics platform at microsoft. *IEEE Software*, July 2013.

[10] V. Driessen. Git-Flow: A successful Git branching model, 2010.

[11] N. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, Sep 1999.

[12] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, Sept 2003.

[13] D. German, B. Adams, and A. Hassan. A dataset of the activity of the git super-repository of linux in 2012. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 470–473, May 2015.

[14] D. M. German, B. Adams, and A. E. Hassan. Continuously mining distributed version control systems: an empirical study of how Linux uses Git. *Empirical Software Engineering*, 21(1):260–299, 2015.

[15] D. Goode and S. P. Agarwal. Scaling Mercurial at Facebook, 2014.

[16] M. Greiler, K. Herzig, and J. Czerwonka. Code ownership and software quality: A replication study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 2–12, Piscataway, NJ, USA, 2015. IEEE Press.

[17] A. Griffin. Metrics for measuring product development cycle time. *Journal of Product Innovation Management*, 10(2):112–125, 1993.

[18] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 141–150, May 2009.

[19] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: Distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 81–90, Washington, DC, USA, 2001. IEEE Computer Society.

[20] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[21] K. Herzig, S. Just, and A. Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, pages 1–34, 2015.

[22] K. Herzig and N. Nagappan. The impact of test ownership and team structure on the reliability and effectiveness of quality test runs. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 2:1–2:10, New York, NY, USA, 2014. ACM.

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

[24] S. Just, K. Herzig, J. Czerwonka, and B. Murphy. Squashing Changes in Git. https://www.sascha-just.com/scig/, 2016.

[25] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 351–360, 2011.

[26] Microsoft. Team Foundation Version Control . https://www.visualstudio.com/en-us/features/version-control-vs.aspx.

[27] M. Mukadam, C. Bird, and P. Rigby. Gerrit software code review data from android. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 45–48, May 2013.

[28] K. Muslu, C. Bird, N. Nagappan, and J. Czerwonka. Transition from centralized to distributed vcs: A microsoft case study on reasons, barriers, and outcomes. In *Proceedings of the International Conference on Software Engineering*. ACM, June 2014.

[29] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[30] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 309–318, 2010.

[31] S. Negara, M. Vakilian, N. Chen, R. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In J. Noble, editor, *ECOOP 2012 Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 79–103. Springer Berlin Heidelberg, 2012.

[32] J. Redstone, E. Bjarmason, S. Vilain, N. T. N. Duy, M. Graham, E. Sazhin, C. Lee, Z. Mokhtarzada, J. Hess, C. Couder, D. Mohhs, G. Troxel, T. Carnecky, and E. Zattin. Git performance results on a large repository, 2012.

[33] P. Rigby, E. Barr, C. Bird, P. Devanbu, and D. German. What effect does distributed version control have on oss project organization? In *Release Engineering (RELENG), 2013 1st International Workshop on*, pages 29–32, May 2013.

[34] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans. Softw. Eng. Methodol.*, 23(4):35:1–35:33, Sept. 2014.

[35] C. Rodriguez-Bustos and J. Aponte. How distributed version control systems impact open source software projects. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 36–39, June 2012.

[36] A. Ruka. Gitflow considered harmful. http://endoflineblog.com/gitflow-considered-harmful/, 05 2015.

[37] A. Tarvo, T. Zimmermann, and J. Czerwonka. An integration resolution algorithm for mining multiple branches in version control systems. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 402–411, Sept 2011.

[38] L. Torvalds. Kernel SCM saga.. https://lkml.org/lkml/2005/4/6/121.

[39] T. Zimmermann, V. Dallmeier, K. Halachev, and A. Zeller. erose: Guiding programmers in eclipse (tool demonstration). In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*, pages 186–187, New York, NY, USA, Oct. 2005. ACM.

[40] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9, May 2007.