Recovering Commit Branch of Origin from GitHub Repositories

Heather M. Michaud[1], Drew T. Guarnera[1], Michael L. Collard[2], Jonathan I. Maletic[1]

[1]Department of Computer Science
Kent State University
Kent, USA
{hmichaud, dguarner, jmaletic}@kent.edu

[2]Department of Computer Science
The University of Akron
Akron, USA
collard@uakron.edu

***Abstract*— An approach to automatically recover the name of the branch where a given commit is originally made within a GitHub repository is presented and evaluated. This is a difficult task because in Git, the commit object does not store the name of the branch when it is created. Here this is termed the commit's branch of origin. Developers typically use branches in Git to group sets of changes that are related by task or concern. The approach recovers the branch of origin only within the scope of a single repository. The recovery process first uses Git's default merge commit messages and then examines the relationships between neighboring commits. The evaluation includes a simulation, an empirical examination of 40 repositories of open-source systems, and a manual verification. The evaluations show that the average accuracy exceeds 97% of all commits and the average precision exceeds 80%.**

Keywords—mining software repositories; Git; version control; branching; merging

I. Introduction

A recurring challenge in Mining Software Repositories (MSR) regards the grouping of commits. Grouping commits into logical change sets that are related is critical for interpreting the historical information found in software repositories. However, this is difficult to do in version control systems (VCS) such as Subversion or CVS since there is no direct support for categorizing or grouping commits in the tools. Hence, researchers developed various methods to accomplish the grouping. Commits can be grouped according to such things as the author [1], a sliding time-window [2-5], the size [6] or type [7] of the change, by the files that were changed [5,8], branch patterns [9-10], or data-mining clustering methods [11]. However, all these commit groupings are an approximation of the developer's actual actions and workflow. The actual set of commits related to a particular change may not be able to be recovered correctly in an automated manner due to missing information.

The advent and near ubiquitous use of Git [12] poses new challenges and opportunities [13] for research in MSR. Git directly supports the creation of branches (i.e., independent, diverging paths from the mainline of development) which can later be merged back into the mainline. Because branching and merging operations are so flexible and efficient in Git, it has been seamlessly integrated into the developer workflow [14]. Developers use distributed version control systems, such as Git, quite differently than centralized version control systems [28]. So a typical Git (and in particular GitHub) workflow involves creating a short-lived topic branch to implement a specific feature or bug fix. This branch is then merged back into the main branch upon task completion. Long-term branches are used for continual development, such as maintaining a stable version of the software. Developers can also work on multiple branches simultaneously. This capability is a strong and intentional deviation from the previous generation of centralized version control system approaches, where branching and merging have limited support and are difficult to accomplish by comparison [28]. For purposes of repository mining, this also provides a direct view of the developer's logical commit, since branches are typically created and named by developers for specific tasks.

Thus, branches provide a natural, logical, and contextual way of grouping commits as they are typically categorized very purposefully by the developer at the time the changes are made. Unfortunately, Git does *not* maintain the record of the branch name to which a commit is originally made [13, 27, 29]. This is due to an underlying design decision of Git that allow for flexibility in naming branches. We term the name of the branch that a commit is originated in as the commit's *branch of origin*. It is the branch that the commit is originally made. An example of this is given in Fig. 1, we see that the branch of origin for commits 11 and 12 is named *hotfix*. In general, the names of branches in Git only exist in the structure of the repository as instances of the present. That is, they are only available for the most recent commit to the branch (e.g., commits 5, 10, and 12 in Fig. 1).

Thankfully, in the case of a merge commit, the names of branches are incidentally recorded by Git in the default commit message. In this situation, the branch of origin of the merge commit can be recovered. Bird et al. [13] presented this issue and a proof of concept for recovering commit branch of origin is given utilizing the default message associated with merge commits. However, as the majority of commits are not merges, there is a need to expand upon the contribution of that work and attempt to identify the branch of origin for all types of commits. Due to the fact that not all information is recorded by Git, determining a commit's branch of origin is not a trivial task. There are a number of situations where even manual inspection can not determine the origin with 100% certainty. However, recovering a commit's branch of origin has the distinct advantage that it gives us the actual grouping of commits that the developer selected (thus modeling their intentions at the time). The alternative (as is done in the case of centralized VCS such as Subversion) is to impose artificial commit groupings such as sliding time windows or file similarity. The branch of origin reconstructs the context chosen by the developer at the actual time of a commit. Thus, a commit's origin provides insight into the patterns of development that relates to a feature, issue, or any other development task as defined by the branch names assigned by a developer. The commit's branch of origin allows us to identify related changes within the same branch and clearly distinguish separately evolving concerns across different branches. Knowing the branch of origin of commits has the potential to greatly improve the historical analysis and understanding of an evolving software system.

Here, we present an approach to automatically recover the previously unknown branch of origin of all types of commits (i.e., both merge and non-merge commits). This is only in the context of a single GitHub/Git repository. To accomplish the recovery process we defined a set of rules based on the internals of how Git stores and manages the version history. The recovery process is not 100% accurate as there are situations that can not be resolved due to the lack of information. However, our evaluation shows the approach to be quite accurate in practice. The main contributions of this paper include:

- a novel approach and algorithm for recovering the branch of origin of all types of commits,
- an empirical study on 40 open-source software systems to determine the types of merges and branches that actually occur, and
- an evaluation of the approach.

The paper is organized as follows. Section 2 presents a graph-based representation of repository data as stored by Git and defines the various branching and merging operations. Section 3 explains the types of merge commits with respect to branch of origin detection, and section 4 discusses their prevalence in 40 open-source systems. Section 5 presents our approach to recovering the branch of origin. The evaluation is given in section 6. This includes a simulation experiment, an empirical study, and lastly a manual evaluation. A discussion of results, threats to validity, and conclusions follow.

II. Git

Git has become widely popularized via the repository hosting service GitHub [15]. Git functions as a miniature file system such that each commit to the repository is a snapshot of the state of the entire file system at that moment [14]. This feature allows for efficient and easy to use branching and merging operations. A branch allows for localized, parallel development, and can later be merged with other branches to form a new version with the changes from each. Again, a commit's branch of origin is defined as the name of the branch to which the commit is originally made. The relationship between commits in the history of a Git repository, as defined via branches and merges, forms a directed acyclic graph (DAG).

Fig. 1 shows an example of a Git repository in its DAG form, where each node is a commit. The commits in this figure are uniquely labeled, though Git uses a unique 40 character hexadecimal identifier for each commit, known as a SHA identifier. A Git commit contains a variety of metadata information, including the SHA identifier for the commit and the SHA of the commit or commits immediately preceding it, known as the parents of that commit. A merge commit is a special type of commit that has two or more parents, one for each branch that is merged. For example, in Fig. 1, there are three merge commits (nodes *5*, *7*, and *10*) which are identified by having more than one parent. The metadata of the commit also contains the author (who made the changes), committer

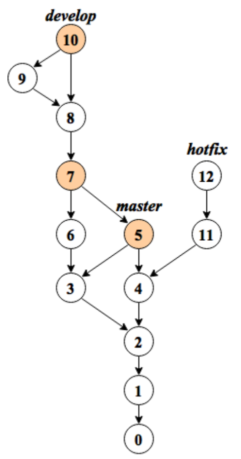

Fig 1. An example Git repository shown as a DAG, where nodes are commits, and the merge commits are shaded.

(who put the changes into the repository), timestamps, and commit message (description of changes).

A. Branch Operations

A critical distinction in Git is that a branch is simply a named pointer to a commit SHA identifier. In Fig. 1, the branches *develop*, *master*, and *hotfix* are just pointers to commits *10*, *5*, and *12* respectively. The creation of a Git repository creates a default *master* branch. A special pointer called the *HEAD* references the branch that the user is currently on, and when a commit is made, it is made to that branch. This is how the user switches between branches. When a commit is made to a branch, the pointer updates to the SHA of the most recent commit. While the commits made to a branch conceptually form a continuous linear path, commits are not stored within a branch this way in Git's structure. Nor is the branch name that a change is committed to stored anywhere in the commit. Branches in Git are just hooks to the present and are not recorded in the history of the repository.

B. Merge Operations

The concept of merging branches is the act of taking the union of the set of states on each branch. Merge is an overloaded term as it is also the name of one of the git commands that performs this act. However, the combination of the changes in each branch can be obtained in a variety of ways with git, including via a merge, a rebase, a pull-request, a fast-forward, or a cherry-pick. Each of these operations perform the task of conceptually merging branches, but the results differ. As merging in Git provides the key pieces of information used to recover the branch of origin for all types of commits, the variation in the types of merging adds a layer of complexity to determining this.

A git *merge* of branches combines the state of the project at each branch using a three-way merging algorithm. Provided that the branches have changes to combine, the result is an automatically created merge commit with two parents (one for each branch involved in the merge) - the commit that the source branch points to, and the commit that the destination branch (current branch) pointed to at the time of the merge. Fig. 1 contains merge commits at nodes *5*, *7*, and *10*. Git provides a default commit message that has information about which branches are involved in the merge. The user has an opportunity to edit the message before committing primarily to append more information to the default message, such as why the merge is necessary. This message comes in a variety of default formats depending on the merge type, but the simplest scenario results in "*Merge source-branch into destination-branch*", where the source-branch contains the commits that will be merged in and the destination-branch is where the commits are merged into. More details on these merge messages are discussed in section 3.

A *fast-forward* is the result of a merge operation in the event that the source branch has all of the commits that exist on the destination branch. In this instance, no merging algorithm needs to be performed, so the destination-branch pointer simply updates, or fast-forwards, to the SHA of the source branch. A fast-forward cannot be detected because no merge commit is created, unless the developer otherwise forces the merge commit to occur by a git command option. For example, commit *2* in Fig. 1 may have been the result of work performed on *develop* which was then merged via a fast-forward with *master*.

A *pull-request* is a method in which any developer can contribute to an open-source project by requesting that the maintainers of the original project pull in her changes to the main repository via a pull-request. Specifically, we are referring to GitHub pull-requests. In this scenario, a merge commit is always created because GitHub uses the git command option to force the merge commit to occur. Commit *10* in Fig. 1 could be a pull-request merge since a merge commit was created when not strictly necessary.

A *rebase* operation calculates and saves the diffs (i.e., patch of differences between two files) of the set of commits from the source branch that do not exist on the destination branch, the source branch is fast-forwarded to the destination branch, and the diffs are re-applied on the source branch as new commits. As a result, no merge commit is created and the rebase cannot be detected. For example, in Fig. 1, it is possible that the work in commit *3* was performed after commit *1* on branch *develop* but rebased onto branch *master* after commit *2*, with no way to determine this.

In a *cherry-pick*, the diffs of one or more commits on a source branch are calculated and re-applied on a destination branch, while the source branch remains unmodified. The commit is duplicated across both branches. It is possible that commit *12* in Fig. 1 was cherry-picked from branch develop as a copy of commit *8*. No merge commit is created and by default no record that the cherry-pick occurred is kept. Thus there is no accurate way to determine that the work was originally performed on a separate branch.

C. Remote Repositories

Commits on a developer's local repository can be pushed to the remote repository on GitHub (or any server hosting a Git repository), and then those changes to the remote repository can be pulled into another developer's local repository for collaboration. These sources do not always match, such as when the local repository is out of date. If changes have been made on both the local and remote repository, a merge commit is created on the local repository when a pull operation is performed. If no changes are made to the local branch on a pull or similarly to the remote branch on a push, then a fast-forward occurs. If a developer executes a pull with the rebase option, then the local branch is rebased on top of the remote branch.

TABLE 1. GIT DEFAULT MESSAGES FOR EXPLICIT MERGE COMMITS BASED ON BRANCH TYPES, WHERE S REPRESENTS THE SOURCE BRANCH AND D REPRESENTS THE DESTINATION BRANCH. MESSAGES ARE SHADED WHERE THE DESTINATION BRANCH IS A NON-MASTER BRANCH.

Merge Type	Source Type	Default Merge Message
Branch	single branch	Merge branch '*S*'
		Merge branch '*S*' into *D*
Tag	single tag	Merge tag '*S*'
		Merge tag '*S*' into *D*
Commit SHA	single commit SHA	Merge commit '*S*'
		Merge commit '*S*' into *D*
GitHub Pull-Requests	pull request	Merge pull request #N from *S*
		Merge pull request #N from *S* into *D*
Remote Branch	single remote branch	Merge remote branch '*S*' Merge remote-tracking branch '*S*'
		Merge remote branch '*S*' into *D* Merge remote-tracking branch '*S*' into *D*
Octopi	multiple branches	Merge branch 'S_1', …, and 'S_N'
		Merge branch 'S_1', …, and 'S_N' into *D*
	multiple commit SHAs	Merge commits 'S_1', …, and 'S_N'
		Merge commits 'S_1', …, and 'S_N' into *D*
	multiple tags	Merge tags '*S*', …, and 'S_N'
		Merge tags 'S_1', …, and 'S_N' into *D*
	multiple remote branches	Merge remote branches 'S_1', …, and 'S_N' Merge remote-tracking branches 'S_1', …, and 'S_N'
		Merge remote branches 'S_1', …, and 'S_N' into *D* Merge remote-tracking branches 'S_1', …, and 'S_N' into *D*

III. Recovering Explicit Merges

As previously discussed, there are many variations for combining branches in Git. Bird et al. [13] investigated two types of merge commits. Namely, a merge of branches and a merge from a remote repository. However, Git defines a wider variety of types. Let us define an *explicit merge* as any Git operation that produces a merge commit. This is also referred to as a *true merge* [14]. This includes a pull-request and a merge or pull operation that does not result in a fast-forward (e.g., merge conflicts produce a merge commit). By definition, an explicit merge is guaranteed to have more than one parent, and by default, explicit merges contain a commit message that details the names of the branches that are combined. The names of the branches in a default merge commit message are ordered with respect to the order of the parent SHA identifiers stored in the merge commit metadata.

An explicit merge cannot be created from a cherry-pick, any rebase operation, or a fast-forward. A merge that occurs and does not result in an explicit merge commit is termed an *implicit merge*. Implicit merges cannot be detected as Git does not record any information about these types of merges; they appear as a series of regular non-merge commits.

We now discuss the varying formats of default messages created by Git during an explicit merge. The default merge commit message depends on the type of merge that takes place. This is determined by the type of the source branch and the destination branch. The source-branch type can be categorized into one or more of the following criteria:

- is from a remote repository,
- is a branch, which is a pointer to a commit SHA that is updated to the most recent commit to the branch,
- is a tag, which is a constant pointer to a commit SHA that is not updated, and is usually used to tag historic events such as the release of a version of the software,
- is a commit SHA,

- is for a pull-request, or
- consists of multiple branches which also meet any of the above criteria, known as an octopi merge due to the octopus merge algorithm Git uses to merge more than two branches.

Regardless of its type, the source branch is ultimately a commit SHA or a pointer to one. The source and destination branch determine the beginning and end of the default message, respectively. For example, a merge from the source branch *hotfix* into the destination-branch *develop* will have the default commit message: *Merge branch 'hotfix' into develop*.

The varying types of merge messages are based on the types of source and destination branches as found from empirical examination and git documentation, and are shown in Table 1.

Additionally, branches from remote repositories can also include the text "of *R*", where *R* represents the URL. The type of default message that git uses is based on the version of the tool at the time the commit is made.

IV. Prevalence of Explicit Merges

The previous section describes the various types of explicit-merge types along with their associated default commit message. We now investigate whether all these types of merges occur regularly in open-source projects. To address this we selected 40 well-known (based on Github's *star* popularity ranking) and active source-code repositories from GitHub, including staple projects such as Linux, Git, Mongodb, Django, and Swift. We are satisfied that the diversity of the projects in the context of programming language, size, number of commits, and domain, form a representative set of projects on GitHub. The size of each repository ranged between 421 and 588,558 commits (at the time of data collection on April 1st and 2nd of 2016). The names of the systems are listed based on how they appear on GitHub, such that the repository can be found in a web browser by appending the name of the system to "https://github.com/".

For each explicit merge found, the commit message is extracted and matched to one of the formats listed in Table 1. The total number of commits for all systems is over one million (1,285,267), where 10% (125,185) of those commits are explicit merges. A count for each type of explicit merge is obtained and a summary of the resultant merge type distributions is shown in Table 3. The merge types shown in Table 3 map to those shown in Table 1 with the exception of the Unknown category (which cannot be derived). Any merge involving multiple branches, remote branches, tags, and commit SHAs are all classified collectively as the Octopi merge type.

Of these explicit merge commits, the most frequently occurring type is a merge of two branches, which consisted of 42% of all explicit merges. This can be explained by the fact that a merge of two branches is the most simplistic form of a merge in a typical Git workflow. The Git project has one of the highest percentages of branch merges (93%), despite a large number of submitted but rejected pull-requests from GitHub. Linus Torvalds, the creator of Git, explains this as a decision not to use GitHub's pull request feature because it lacks required information in the commit message for his standards.[1] This highlights the importance of keeping the default message as supplied by Git for accurate record-keeping. The second most frequent type of explicit merge is a GitHub pull-request, consisting of 39% of all explicit merges. Less commonly occurring types of explicit merges are from tags and remote repositories, which account for less than 7% and 3%, respectively. The least frequent types by far are octopi (i.e.,

Table 2. The 40 open-source project repositories used in the empirical study.

System	Domain	Language	Total Commits
torvalds/linux	Operating system	C	588,558
mono/mono	Development framework	C#	116,622
rails/rails	Web application framework	Ruby	67,445
Homebrew/homebrew	Package manager	Ruby	63,808
mongodb/mongo	Database	C++	53,922
git/git	Content management	C, Shell, Perl	44,402
GNOME/gimp	Image editor	C	42,483
apple/swift	Programming language	C++, Swift	36,403
django/django	Web framework	Python	33,337
gitlabhq/gitlabhq	Server-side content management	Ruby, HTML	29,414
docker/docker	Application container	Go, Shell	24,709
meteor/meteor	Web framework	JavaScript	21,152
ansible/ansible	IT automation	Python	20,794
mrdoob/three.js	3D library	JavaScript, Python	15,182
twbs/bootstrap	Web framework	CSS, JavaScript	14,850
zurb/foundation	Front-end framework	CSS, JavaScript	12,103
joyent/node	Web platform	Web platform	11,176
angular/angular.js	JavaScript API	JavaScript	10,213
libgit2/libgit2	Content management API	C	9,956
facebook/react	User interface library	JavaScript	7,984
jekyll/jekyll	Static site generator	Ruby, Cucumber	7,548
jquery/jquery	JavaScript Library	JavaScript	7,275
strongloop/express	Web framework	JavaScript	5,479
RocketChat/ Rocket.Chat	Web chat server	CoffeeScript, JavaScript	5,278
Microsoft/vscode	Code editor	TypeScript, JavaScript	4,249
mbostock/d3	Visualization library	JavaScript	3,994
robbyrussell/oh-my-zsh	Configuration framework	Shell	3,907
jashkenas/backbone	JavaScript API	JavaScript, HTML	3,301
AFNetworking/ AFNetworking	Networking framework	Objective-C	3,005
moment/moment	Date manipulation API	JavaScript	2,857
nwjs/nw.js	Webkit	C++, JavaScript	2,674
github/gitignore	Templates	gitignore	2,115
hakimel/reveal.js	Presentation framework	JavaScript, CSS	1,787
fzaninotto/Faker	Data generator	PHP	1,619
h5bp/html5-boilerplate	Front-end template	JavaScript, CSS	1,592
facebook/infer	Static analyzer	OCaml, Java	1,160
FortAwesome/Font-Awesome	Font toolkit	HTML, CSS	1,111
blueimp/jQuery-File-Upload	API extension for files	JavaScript, HTML	947
google/roboto	Font family	Python	435
bpampuch/pdfmake	Client/server documentation printing	JavaScript	421

[1] https://github.com/torvalds/linux/pull/17#

Table 3. Summary of explicit merge types in studied open-source projects.

Merge Type	Average Per System	Total Of All Systems	Percent of Explicit Merges
Branch	1,328	53,137	42.4%
GitHub Pull-Requests	1,219	48,767	39.0%
Unknown	205	8,217	6.6%
Tag	222	8,881	7.1%
Remote Branch	96	3,857	3.1%
Octopi	38	1,517	1.2%
Commit SHA	20	809	0.6%
Explicit Merges	3,130	125,185	100.0%
All Commits	32,132	1,285,267	

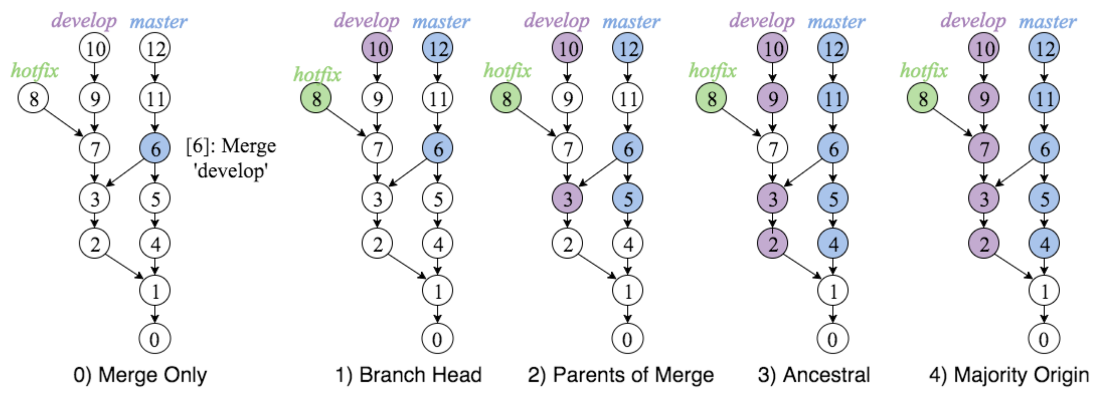

Fig. 2. An example Git repository shown at each stage that the algorithm identifies commit origins. The branches, *hotfix*, *develop*, and *master*, are given along with the commits that the algorithm identified. First, the origins of explicit merges only are identified. Each rule is applied after the previous stage to monotonically increase the accuracy of commit origins.

more than two branches) and merges of commit SHAs, which each account for around 1% of the merge commits.

For nearly 7% of explicit merges the type cannot be identified based on the the commit message alone, which is due to the ability of developers to overwrite the default message. The format of the merge commit message is not guaranteed. Some of the common non-default messages found include "*Merge from S*", and "*Automatic merge of R*", and "*git5: git5track synced with perforce at N*". While some messages, such as the git5track message, are the result of an automated tool, others however are due to the committer rewriting the message. Many of these messages are completely rewritten to the extent of removing all pertinent information, and these types cannot be used for branch name extraction.

Thus we can now answer our question about the prevalence of explicit-merge types. Merges of branches and of pull-requests are by far the most commonly recorded types of merge commits. The remaining types of merges with tags, remote repositories, multiple branches, and commit SHAs each occur the least frequently, in less than 12% of explicit merge cases. However, a notable 7% of explicit merges are of an unknown type due to developer interference. Fortunately, the remaining 93% of known explicit-merge types contain the default merge commit message, which can be used to extract the source and destination branches from the commit message. This sets the baseline for our rule-based algorithm for recovering commit branch origins.

V. Rules for Origin Recovery

Identifying which branch a commit is originally made to is no trivial task, and in some cases, an impossible one. Explicit merge commits in a repository are the only commits that are guaranteed to have originated in the destination-branch, which is specified by the default merge message. Here we direct our effort to recovering the branch of origin of the other types of commits (i.e., non-merge commits).

The order of the parent hashes with respect to the branch names as they appear in the merge commit is crucial to the commit's branch of origin recovery process. For example, a merge commit with parent SHAs 1 and 2 as identified by the git log command, and a commit message of "*Merge branch B into A*", allows us to associate SHA 1 with branch A and SHA 2 with branch B. Thus with string matching, we can extract the names of each branch and match them to commit SHAs.

Our algorithm first performs the integral task of identifying the origins of explicit merge commits. As Bird et al. [13] detects the branch names that these explicit merges are committed to, this stage is most closely related to that work. An example Git repository is presented in Fig. 2 and shows the branch of origin for the merge commits based on the merge commit message. Here, the message of commit *6* does not include the destination-branch, which indicates that the commit's branch of origin is master. As a result of stage 0, only the origins of explicit merge commits are identified.

We designed four rules to expand the set of commits with identified origins by assigning origins to the appropriate neighbors of the commits that have already been identified. We use a combination of identifying the origins of the branch heads, parents of explicit merges, ancestry, and finding the majority-origin along the branch segment to define the rules. Each rule is applied sequentially and increases the number of commits whose origin can be recovered at each stage.

The pseudocode for the full algorithm is shown in Fig. 3. The input is the entire DAG of commit objects and the output is an updated DAG where each commit is annotated with the

```
Input: G  - DAG of commit objects
Output G' - G annotated with the branch of origin for commits

Algorithm IdentifyCommitOrigins(G)
  G ← ApplyBranchHeadRule(G)

  for c ∈ G do
    // merge commits
    if |c.parents| > 1 then
      origins ← IdentifyOriginsOfMerge(c.message)

      // If the commit contains the default message and thus the
      // merge origin's detection was successful
      if |origins| > 1 then
        c.origin ← origins[0]
        sources ← origins

        destinationParent ← c.parents[0]
        G ← ApplyMergeParentRule(G, destinationParent, c.origin)
        G ← ApplyAncestralRule(G, destinationParent, c.origin)

        for i ← 1...|c.parents| do
          sourceParent ← c.parents[i]
          G ← ApplyMergeParentsRule(G,sourceParent,sources[i])
          G ← ApplyAncestralRule(G, sourceParent, sources[i])
        end for
      end if
    end if
  end for

  G ← ApplyMajorityOriginRule(G)
  return G
```

Fig. 3. Pseudocode for recovering commit origins in Git repositories.

branch of origin whenever possible. For each merge commit the algorithm identifies the origins of the merge using the default commit message. If that is successful, then the various rules are applied incrementally. The rules will now be individually presented, along with a pseudocode representation and their application to the running example of Fig. 2.

A. Branch Head Rule

This rule identifies the origins of the most recent commits. Recall that the HEAD pointer in a Git repository points to the current state, usually identified by one of the branches. A branch head references a current commit SHA for that branch. The pseudocode for this rule is given in Fig. 4.

```
Input: G  – DAG of commit objects
Output G' – G annotated with the branch of origin for commits

Algorithm ApplyBranchHeadRule(G)
  B ← Set of current branch names and pointers to commit SHAs
  for b ∈ B do
    commit ← b.pointer
    G[commit].origin ← b.name
  end for
  return G
```

Fig. 4. Pseudocode for the Branch Head Rule.

The branch-head rule is stage 1 of the algorithm and labels the commit that each branch points to as originating in that branch. Fig. 2.1 shows the additional commits whose origins can be recovered, namely, the commits pointed to by branch heads. The branches *hotfix*, *develop*, and *master* point to commits *8*, *10*, and *12*, respectively. Thus, those commits are also labeled as originating with their respective branch heads.

B. Merge Parents Rule

Explicit merges have a default commit message that specifies the source and destination branch. In stage 2, this rule identifies the parent commits of the explicit merge as belonging to the source and destination branches. This relies on the order of the parents as listed in the merge commit message, which corresponds to the order of the parent SHAs as listed in the merge commit's metadata. The commit matching the merge commit's first parent SHA is labeled as originating in the destination-branch name. The commit matching the second parent SHA is labeled as originating in the source-branch name. If the merge commit has multiple source branches, the remaining commits matching the rest of the parent SHAs are marked as belonging to the nth source branch name, as the order of the default commit message matches the order in which the parent SHAs are listed for the merge commit. The pseudocode for this rule is shown in Fig. 5.

Fig. 2.2 shows the additional commits (*3* and *5*) whose origins are identified with the merge-parents rule. As the message of commit *6* indicates that the source branch is develop, the parent commit 3 is labeled as originating in develop. As the destination branch is not specified, we can determine it is the *master* branch, so the merge-parents rule also identifies the parent commit *5* as originating from *master*.

```
Input: G  – DAG of commit objects
       C  – Commit whose branch of origin will be labeled
       N  – Name of the the branch of origin for the commit
Output G' – G annotated with the branch of origin for commits

Algorithm ApplyMergeParentRule(G, C, N)
  G[C].origin ← N
  return G
```

Fig. 5. Pseudocode for the Parents of Merge Rule.

C. Ancestral Rule

In stage 3 of the algorithm, once all of the previous rules are applied, the set of all non-merge commits whose origins have been identified are obtained. For each commit in the set, the ancestry of that commit is labeled as originating in the same branch, as shown in the pseudocode in Fig. 6. Once a commit is identified with an origin, its parent is obtained and identified with the same origin. This process continues traversing based on the parent of the current commit until it reaches a commit that has more than one parent (i.e., is a merge commit) or has more than one child (i.e., branches out). Once the ancestry of the commit is traversed, the next non-merge commit from the set of commits with identified origins is traversed. This is due to the fact that any merge commit will be identified based on its message, so it does not need to be included, and any commit with more than one child (e.g., a commit that forms a branch path) creates an ambiguity. It is not known if the commit with more than one child has the same branch of origin for its first child, second, nth, or has a branch of origin different from all of the children.

```
Input: G  – DAG of commit objects
       C – Commit whose ancestors will be identified
       N – Branch name to identify C's ancestors with
Output G' – G annotated with the branch of origin for commits

Algorithm ApplyAncestralRule(G, C, N)
  // Transpose G such that R is a DAG of commit objects
  // from child to parent nodes, and the "parents" of a
  // commit in R are the children of a commit in G
  R ← Transpose(G)

  nextParent ← G[C]
  repeat
    G[nextParent].origin ← N
    nextParent ← G[nextParent].parents[0]
  until |G[nextParent].parents| > 1 or
        |R[nextParent].parents| > 1

  return G
```

Figure 6. Pseudocode for the Ancestral Rule.

In addition to the previously identified commit origins, Fig. 2.3 shows newly identified commits *2*, *4*, *9*, and *11*. The ancestor of commit *8* (commit *7*) can not be identified since it has two children. The ancestor of *10* (commit *9*) is labeled as having a branch of origin of develop, the same as its child. For the same reason as before, the ancestor of commit *9* (commit 7) is not labeled. Commit *12*'s ancestors, until a merge, includes only commit *11*, which is labeled as originating from master, consistent with commit *12*'s origin. In addition, the only single child, non-merge ancestor of commit *3* is commit *2*, whose branch of origin is marked in the same manner. Lastly, commit *5*'s only single child, non-merge ancestor is commit *4*, which is labeled as originating from *master*.

D. Majority Origin Rule

In stage 4, the majority origin rule finds all of the separate linear paths in the repository, excluding merge commits. In Fig. 2, one path contains commits *4* and *5* which begin at commit *1*, and another path contains commits *11* and *12* which begin at commit *6*. These paths are extracted such that each non-merge commit belongs to exactly one path. All the commits on the path are traversed and the most commonly identified branch name is identified. Each commit on or beginning the path that has not been identified with a branch of origin is labeled as originating in the most common branch of origin within that path. This is shown in the pseudocode for this rule in Fig. 7.

```
Input: G  - DAG of commit objects
Output G' - G annotated with the branch of origin of commits

Algorithm ApplyMajorityOriginRule(G)
  S ← Set of linear paths and their starting point
  for s ∈ S do
    origins ← []

    for c ∈ s do
      origins ← append(c.origin)
    end for

    majority ← most common item in origins
    if majority is found then
      for c ∈ s do
        if G[c].origin is unidentified then
          G[c].origin ← majority
      end for
    end if
  end for

  return G
```

Fig. 7. Pseudocode for the Majority Origin Rule.

This rule is designed to take advantage of the commits along a linear path whose origins are identified in previous stages to estimate the most likely branch of origin for neighboring commits. For example, Fig. 2.4 shows that commit *7* is labeled as originating in develop as a result of the majority origin rule. Only commits *7* and *1* remained unidentified from previous stages. Commit *1* lies on a linear path by itself, whose beginning is demarcated by commit *0*. As no commits on the path or that begin the path have been labeled, commit *1* remains unidentified. Commit *7* also lies on a linear path by itself, where the beginning of the path is demarcated by commit *3*. As commit *3*'s branch of origin is *develop*, commit 7 is also marked as originating from *develop*.

VI. Evaluation

Testing the precision of the algorithm presented requires that the commit branch of origin is already known in order to compare the results. As previously stated, no other tool exists that can identify the branch of origin of all commits, nor is this information kept as a part of the history in the repository. This complicates the evaluation, so three different methods are taken to measure the performance of the algorithm as efficiently implemented in a tool. First, a simulation is performed, which includes a variety of test repositories which are generated with the commit branch of origin recorded in the history. These records are compared to the origins as obtained by the algorithm when run on the test repositories, allowing for measurements of precision and accuracy. Second, the algorithm is run on the 40 open-source systems collected from GitHub. As the algorithm's goal is to recover the origin of commits where it is unknown, only accuracy can be calculated on these systems. To account for this, the third evaluation includes a two-person manual verification of a random subset of the commits in one of the 40 systems, with measurements of precision and accuracy.

We use precision and accuracy to measure the algorithm's performance. Precision measures exactness, and is defined as:

$$precision = \frac{\#\ correctly\ identified\ commit\ origins}{\#\ identified\ commit\ origins}$$

High precision means that the algorithm returned more relevant results than irrelevant. High accuracy, which measures completeness, means that most of the relevant results are returned, and is defined as follows:

$$accuracy = \frac{\#\ commit\ origins\ identified}{\#\ commits\ in\ repository}$$

```
$ git log --graph --abbrev-commit --decorate --pretty=format:'%C(bold blue) \
%h%C(reset) - %s%C(bold green)%d %C(reset)' --all
* a520d59 - Commit 13 to A (A)
* dac5201 - Commit 12 to A
*   74118bc - Merge branch 'master' into A
|\
| * 86b33bb - Commit 9 to master (master)
| *   2c37d11 - Merge branch 'C'
| |\
| * | 8e1ff62 - Commit 6 to master
| * | 0766069 - Commit 5 to master
* | | 2ba39d9 - Commit 11 to A
| | | * 750be82 - Commit 14 to C (HEAD, C)
| | | * 755e4cb - Commit 10 to C
| | |/
| | * 70c0740 - Commit 4 to C
| |/
| * c9ea278 - Commit 3 to master
| * 452c9af - Commit 2 to master
| | * 9bb7521 - Commit 8 to D (D)
| |/
|/|
* | a85afa2 - Commit 7 to A
* | 90b474b - Commit 1 to A
|/
* 90638b6 - Commit 0 to master
```

Fig. 8. An example simulated Git repository displayed by the git log command. Each non-merge commit message is annotated by the tool with the commit number and branch.

A. Simulation

A script is created for the generation of 22 test repositories varying in number of commits and number of branches. The script initializes a Git repository with *n* branches. One of these branches is checked out at random, and a commit is made to that branch. After each commit, there is a 2/3 chance that a merge with one other branch will occur, otherwise a 1/6 chance that a merge of two other branches will occur. The odds of a merge happening are estimated to match or exceed the instances of a merge, either explicit or implicit, occurring in actual repositories. If either merge type occurred, the branch or branches to be merged in is randomly selected. This process repeats until the desired number of non-merge commits has been reached. If an explicit merge occurs, the text of the merge commit is not altered in any way, as this message will be used by the algorithm for branch name detection. Recall that implicit merges do not create merge commits. As this poses a threat to validity, we tried to recreate the issue with high merge probability. These odds produce projects similar to those with frequent collaboration and parallel development.

A small repository generated as an example of test repository generation is shown in Fig. 8. The graph is shown by git's log command and displays the first 7 characters of the commit SHA, then a hyphen, and the commit message. The branch names are encoded in parenthesis in the commit message of the current commit that they reference. For example, *master* points to commit *86b33bb*, which is the commit with message "*Commit 9 to master*". This message is generated by the test script, not by Git. However, merge commit 74118bc with the message "*Merge branch 'master' into A*" is generated by Git, along with all the other merge-commit messages. Also notice the extra branch name *HEAD*, which as previously stated, is a special pointer to indicate which branch (or commit) the developer is currently on.

The algorithm is run on 22 generated test repositories ranging from 100 to 3,000 non-merge commits and 5 to 20 branches. The full set of the test repositories can be found on https://github.com/research-data. On each repository, commit origins were identified at each stage 0-4 of the algorithm. This began with stage 0, which identified only the explicit merge, then at stage 1 with the application of the branch-head rule, at stage 2 with the added application of the merge-parents rule, at stage 3 with the added ancestral rule, and lastly at stage 4 with all of the rules applied including the majority-origin. Thus at

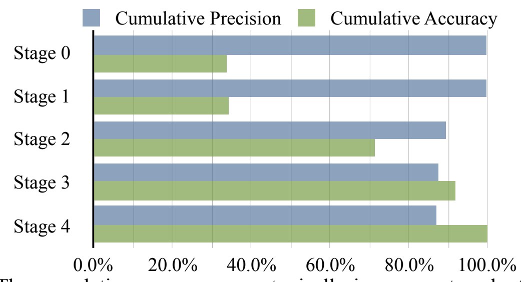

Fig 9. The cumulative accuracy monotonically increases at each stage of the algorithm, while precision decreases slightly, as measured on average of the 22 generated test repositories.

each stage, an additional rule is used for increased accuracy, and the commit messages which contain the branch name are extracted to calculate the precision. The average precision and accuracy accumulation over all test systems at each stage of the algorithm is shown in Fig. 9.

The accuracy in stage 0 matches exactly with the number of explicit merges in the repositories, as these are the only ones identified at this stage; this is similar to the approach taken by Bird et al. [13]. In stage 1, which applies the branch-head rule, , the accuracy increases very slightly as it is completely dependent on the number of existing branches in the repository at that time. Even though the accuracy increase is trivial at this stage, the branch-head rule compounds the number of identifiable origins when the ancestral rule is applied. In stage 2, the addition of the merge-parents rule greatly improves the average accuracy because for each explicit merge point *m* in the repository, every parent of *m* is identified. Recall that an explicit merge is guaranteed to have at least two parents. Thus the accuracy that can be obtained in comparison to stage 0's identification of merge origins is, at a minimum, tripled. The ancestral rule is applied in stage 3, which was found to be most useful when branches diverge for a long series of commits before merging back in, as it will identify the remaining commits on the diverging path. Finally, the majority origin rule is applied last in stage 4 as it labels nearly all of the previously unlabeled commit origins based on the majority of the surrounding known origins. This results in over 2.5 times the average accuracy of 33.7% obtained in stage 0. Thus, the presented rule-based algorithm increases the accuracy by an approximate 53 percentage points when all rules are applied in stage 4, bringing the average accuracy to 100%. This gives us high confidence that the tool implementation of the algorithm works properly, so that we can proceed in the evaluation using open-source systems.

B. *Branch of Origin in Open-Source Systems*

We applied our rule-based algorithm to the same 40 systems listed in Table 2. At most, this took a minute on the cloned repository for each system. The accuracy is measured at all stages: with merge origins only, with the branch-head rule, with the addition of the merge-parents rule, with the ancestral rule also applied, and finally with all rules applied including the majority origin. As previously stated, the precision cannot be measured as there exists no other known tool to accurately identify the branches on which changes were originally made in Git repositories. However, the accuracy is still an indicator as to how well the algorithm performs on real systems, and the expected precision can be inferred based on the precision as shown in the generated test repositories, and manual verification.

TABLE 4. ACCURACY OF RULES ON OPEN-SOURCE SYSTEMS. ACCURACY IS REPORTED IN STAGES WITH THE RULES APPLIED IN ORDER.

System	Accuracy				
	Stage 0. Merge Only	Stage 1. Branch Head	Stage 2. Merge Parents	Stage 3. Ancestral	Stage 4. Majority Origin
linux	7%	7%	14%	70%	98%
mono	2%	2%	5%	16%	68%
rails	18%	18%	42%	64%	90%
homebrew	0%	0%	0%	5%	100%
mongo	9%	9%	21%	43%	90%
git	23%	23%	44%	73%	100%
gimp	0%	0%	0%	23%	100%
swift	5%	5%	11%	18%	100%
django	2%	2%	5%	37%	96%
gitlabhq	30%	30%	61%	92%	100%
docker	38%	38%	77%	93%	100%
meteor	6%	8%	18%	59%	99%
ansible	25%	25%	59%	80%	100%
three.js	17%	17%	43%	72%	99%
bootstrap	26%	26%	61%	82%	100%
foundation	26%	26%	60%	78%	99%
node	4%	5%	11%	46%	100%
angular.js	0%	0%	1%	27%	100%
libgit2	22%	23%	48%	89%	100%
react	28%	28%	61%	84%	100%
jekyll	20%	20%	58%	87%	99%
jquery	3%	3%	8%	17%	61%
express	9%	9%	23%	41%	100%
Rocket.Chat	27%	28%	61%	87%	100%
vscode	7%	8%	19%	60%	100%
d3	17%	18%	39%	81%	100%
oh-my-zsh	36%	36%	72%	94%	99%
backbone	26%	27%	62%	80%	99%
AFNetworking	24%	24%	56%	77%	100%
moment	25%	25%	54%	84%	99%
nw.js	16%	17%	41%	60%	100%
gitignore	40%	40%	80%	93%	100%
reveal.js	15%	15%	38%	58%	96%
Faker	27%	27%	59%	93%	100%
html5-boilerplate	11%	12%	30%	48%	99%
infer	1%	1%	2%	94%	100%
Font-Awesome	11%	12%	28%	52%	98%
jQuery-File-Upload	8%	9%	24%	28%	100%
roboto	12%	12%	27%	51%	100%
pdfmake	14%	15%	34%	58%	100%

The full table of the accuracy values for each system is presented in Table 4. To summarize, the total number of all commits for all repositories combined is 1,285,267. In stage 0 of only identifying the branch of origin of explicit merges, an average 16% of all commits are identified with a branch origin. With each rule applied, the accuracy monotonically increases. The addition of the branch-head rule at stage 1 increases the accuracy very slightly to 16.2%. In stage 2, the parents of the merge rule brings this value to 36.5% of all commits in all of the repositories, at stage 3 the ancestral rule raises the accuracy to 62.3%. By the addition of the final rule, the majority origin, stage 4 reveals that 97.2% of all commits in all repositories had origins that have been identified. The contribution to accuracy that the rules provide here is invaluable, from 15.9% in stage 0 to 97.2% in stage 4.

The most severe disparity between the accuracy as measured at stage 0 and at stage 4 is that of the homebrew repository, in which stage 0 results in a accuracy of 0% while the stage 4 results in a accuracy of 100%. However, this is attributed to the very low number of explicit merges with respect to the total number of commits in this repository.

The Linux repository is chosen for further examination due to its significant number of commit contributions. We recovered 8,699 different branch names. Since git does not track branch names in commits, branch names can be reused in a repository and may not be unique (e.g., deleting a branch and

TABLE 5. TOP 10 LARGEST BRANCHES IN LINUX

Branch Name	Commits
git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-2.6.25 [d10f21-85040b]	1,470
git://git.kernel.org/pub/scm/linux/kernel/git/x86/linux-2.6-x86 [213eca-afadcd]	890
master [571ecf-266918]	867
staging-next [a504de-68cf16]	864
staging-next [1407a9-a4ac0d]	837
staging-next [0f431f-9056be]	810
git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging-2.6 [874073-ba0e1e]	714
Btrfs [be0e5c-343530]	714
staging-next [9fc860-1a4b6f]	709
for_linus [e164b5-fd3a01]	707

TABLE 6. SAMPLE OF SMALL BRANCHES IN LINUX.

Branch Name	Commits
bugzilla-13751 [74b582-74b582]	1
video-error-case [e01ce7-e01ce7]	1
cpu-bindings [594f88-deeea7]	2
new-drivers [452c1c-c4e84b]	4
arm-build-fixes [fc9a57-96a301]	4
unnecessary_resource_check [5e9b4d-aaa14f]	5
next/hdmi-samsung [566cf8-0a9d5a]	6
dell-laptop [e1fbf3-8c5d30]	6
lookup-permissions-cleanup [e8e66e-18f4c6]	8
x86/setup-lzma [bc22c1-889c92]	13

making a new one later with the same name). To provide additional granularity, we uniquely identify each occurrence of these branch names with a starting and ending commit SHA identifier. Adding this identifier yields 53,275 unique branches. A list of the most active branches in terms of number of commits originating from them are shown in Table 5 and includes the abbreviated (6 character) commit SHA identifiers. A sample of other branches in which very few commits originated is shown in Table 6. Of the branches in Tables 5 & 6 for Linux, the only branch name shown in Github is the master branch. Branches with more commits tend to be long running branches used for staging areas (*for-linus*, *staging-next*, etc.) or for stability (*master*). Branches with few commits were short-lived topic branches that originally implemented a specific bug fix or feature, e.g., *x86/setup-lzma*, *next/hdmi-samsung*, etc. In some cases the bug number is used as the name of the branch (*bugzilla-13751*). This finding suggests that the purpose of a change made in a commit (e.g., maintenance task or new feature) can be determined by the branch classification (e.g., staging area or topic branch).

Recall from Table 1 that the empirical examination found two versions of Git's default message during a merge with a remote branch. Manual analysis of the branches in the Git project's source repository revealed that the branch named *mm/phrase-remote-tracking* contained commits that were responsible for rewording the default message to use the phrase "remote-tracking branch" rather than "remote branch". This is an excellent example of how branch names can provide context of the meaning of the commits originating in that branch.

C. *Manual Verification*

As no known Git repository records a commit's branch of origin, manual verification is performed using the additional external information that GitHub provides for development workflows that use pull requests. The react system is selected as it is the largest project of the studied systems that most predominantly uses the GitHub pull-request model. It consists of 7,984 commits with over 98% of the explicit merges identified as pull-request types. A random selection of 400 commits were made (5% of the total commits). For each commit, two of the authors (both experienced developers) independently manually analyzed the commit to determine which branch they believe it originated from. This manual process took approximately 15 hours for each subject to perform. The subjects used only the data as provided by GitHub to gather pull-request and issue-tracking information, commit information, and ancestors of the commit.

TABLE 7. MANUAL VERIFICATION REVEALS A HIGH TOOL PRECISION AND AGREEMENT BETWEEN SUBJECTS, WHERE S1 AND S2 ARE THE HUMAN SUBJECTS AND T IS THE TOOL IMPLEMENTING THE ALGORITHM.

Subject Agreement	Occurrences	Percentage of Selection
S1 ∩ T	316	79%
S2 ∩ T	313	78%
S1 ∩ S2	395	99%
S1 ∩ S2 ∩ T	312	78%
(S1 ∩ T) ∪ (S2 ∩ T)	319	80%

The algorithm identified the origins of all commits, and the origins of the 400 selected commits were extracted. Subjects 1 and 2 separately labeled each of the 400 commits with the branch origin. The correct branch of origin is determined as the one chosen by at least 2 of the 3 participants. Thus, the algorithm is considered to have correctly identified a commit's branch of origin if it agrees with at least one of the subjects, and to have incorrectly identified a commit's branch of origin if it disagrees with both of the subjects.

As shown in Table 7, the tool agrees on nearly 80% of the commits with each subject. Subjects 1 and 2 agree in nearly all cases. All subjects agree in 78% of the commits. Of the 400 commits, the algorithm correctly identified 319 commits (i.e., agreed with subject 2 or 3), producing an accuracy of 100% and a precision of 80%, which is consistent with the 100% accuracy and 87% precision obtained from the generated test repositories.

VII. DISCUSSION OF RESULTS

The evaluation of the rule-based algorithm to recover branch names of commits had three separate parts including the generated test repositories, the branch of origin of commits in 40 open source systems, and manual verification.

This work produced very good values of accuracy and precision for commit branch of origin recovery. There is an average accuracy of at least 97% in all three evaluations and an average precision of at least 80% is obtained in the two evaluations where precision can be determined. During manual verification, the 80% precision of the algorithm is especially interesting as it trails the results performed by the human subjects (99%) for the open-source system by 19 percentage points. In addition, the algorithm takes seconds to identify over 7,000 commit origins in the selected repository, while it takes each human subject at least 15 hours to identify 400 of the commit origins. One important fact to keep in mind is that the automated approach is working at a disadvantage compared to the human subjects as it only bases its branch of origin recovery using the commit messages and structure, while human subjects had access to all project information contained in the project's GitHub repository. The react system is selected specifically for its high utilization of GitHub to make the manual verification more accurate, but such is not the case in all repositories. Additionally, even human participants can disagree on the branch origin, illustrating that the results can be subjective. With respect to the simulation results, the presented algorithm not only works under ideal artificial circumstances, but performs well in practice.

One of the threats to validity is that the generated repositories do not fully mimic actual projects as they do not contain pull-requests, remote branches, or merges between commits or tags; these missing types of merges are left for future work. Actual repositories cannot be automatically tested for precision, and even the recorded merge commit message used for origin identification can be re-written by developers, limiting the ability to accurately detect that commit's branch of origin and the origin of its neighboring commits. Multiple branches referencing the same commit have an ambiguous origin. Additionally, implicit merges and renamed branches cannot be detected given the repository history. This can cause the rules of the presented algorithm to incorrectly identify a commit's origin.

VIII. Related Work

The extraction of a commit message is a source of valuable research for many purposes, such as viewing it through the lenses of documentation of the source code [16], helping to determine if two files were modified in the same commit in VCS tools that do not store this metadata [17-18], ontological modeling [19], or the quality of the commit [20]. Though, to the best of the our knowledge, only one study has used this information for determining the branch of origin of a commit [13], but it is only done for explicit merge commits.

Bird et al. present the notable difficulties and exciting new data available when mining Git repositories [13]. As part of the overall work, the merge commits of 30 open-source projects are examined in order to determine the source of the merge, i.e., the branch of origin of the merge commit. An accuracy of 97.9% is obtained for the merge commits alone, though an accuracy of 2.1% is calculated with respect to all commits in the repository. The work of this paper extends the work of Bird et al. by presenting a rule-based algorithm for recovering all commit origins in the entire repository, as well as an analysis of the types of explicit merges and branch origins as found in open source projects.

Tarvo et al. [21] present an integration algorithm that allows for data collection in the presence of branches in order to re-collect the commit information that is typically lost after the merge of the branches in a centralized VCS. This is used to predict software-regression risk by identifying which bugs are fixed in different branches of the system. However, it depends on knowing the branch of origin of the commits at the time of integration. Ghezzi et al. [22] present a VCS plug-in extension architecture to expand upon the information collected by the VCS for analysis support, where an event such as a branch or merge can trigger a plug-in to calculate metrics or extract semantic changes. This infrastructure could be extended to support the branch name extraction required to enhance the history of a project by recording it at present intervals.

Germán et al. [27, 29] focus on continually mining a collection of related repositories to detect which *repository* a commit comes from as it is added to the main "blessed" repository. This complements our approach, which is centered on the retroactive analysis of a single repository and from which *branch* a commit came from when it is added. However, we do not mine any additional repositories to acquire commit origins external to the repository under analysis, and instead focus on a finer level of granularity.

In addition, a series of studies were performed using the branch structure for purposes of distribution metrics [9], merge conflict detection [23] and resolution [24], text-level authorship of a source-code document [25], pattern extraction [10], and their effect on software quality [26]. However, in all of these studies, the paths were unnamed and recognized only as diverging paths rather than textual and contextual categorization of commits.

IX. Conclusions and Future Work

The main contribution of this work is a novel and efficient algorithm for recovering the branch of origin for commits in Git within the context of a single repository. The algorithm is evaluated in three ways: with simulated test repositories, on the GitHub repositories of 40 open source systems, and by a two-person manual verification on a select system. The algorithm produces an average precision of at least 80%, and in all three evaluations produces an average accuracy of over 98% of all commits. The tool implementation takes around a minute to recover the branch origins of over 500,000 commits when executed on the Linux repository.

There is little previous work [13] on this problem and that work only addressed a portion of the issue in the broader context of examining various issues in Git. As such it is difficult to directly compare as the approach described here subsumes the portion of that work aimed at the problem (i.e., they only examined explicit merge commits and this is the first step in our approach). Only examining explicit merge commits produces an accuracy of 16% on the systems study, and in the case of Linux 7%.

Another contribution is an analysis of explicit-merge types, where we found that the most prevalent explicit merges involve pull-requests and branches, while very little involve commits, tags, remote repositories, or an octopi merge. Of the explicit-merge types, 7% can not be identified due to the use of non-default commit messages. Additionally, we found that the systems contained approximately 10% merge commits.

An analysis of the Linux repository shows that the topmost active branches (e.g., *master*, *staging-next*) are typically used as staging areas for development. Branches with less activity are also of interest as their names reflect what the developer is working on (e.g., *new-drivers*, *arm-build-fixes*). While our method for grouping commits does not replace other grouping methods, it does provide a view that directly shows the logical task of the developer. A direct application of our approach would be to perform other measures on the branch-grouped commits, e.g., LOC, number of files, collection of changes, etc.

Recovering this information has several benefits when mining software repositories and should improve the historical analysis and understanding of an evolving software project. This motivates future work into the role, activity, and frequency of branches within and across repositories. The recovery of commit branch origins in Git allows, for the first time, studies to occur on the branches themselves. Types of branches, such as a source or destination branch, can be found based on the amount of commits performed in a particular branch versus the number of merges to that branch.

We plan to extend this work to consider the origins of commits across multiple related repositories, which will allow commit origins to be determined not only from the branch of origin, but also trace back from which external repository and branch the commit came from as well. This will provide a rich global view of the development process.

References

1. X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat, “Mining Software Repositories for Accurate Authorship,” in Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM’13), Eindhoven, The Netherlands, September 22-28 2013, pp. 250–259.
2. C. Rodriguez-Bustos and J. Aponte, “How Distributed Version Control Systems impact open source software projects.,” in Proceedings of the 9th Working Conference on Mining Software Repositories (MSR’12), Zurich, Switzerland June 2-3 2012, pp. 36–39.
3. A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, “Mining Software Repositories to Study Co-Evolution of Production & Test Code,” in Proceeding of the 1st International Conference on Software Testing, Verification, and Validation (ICST’08), Lillehammer, Norway April 9-11 2008, pp. 220–229
4. F. Jaafar, Y. Guéhéneuc, S. Hamel, and G. Antoniol, “An Exploratory Study of Macro Co-changes,” in Proceedings of the 18th Working Conference on Reverse Engineering (WCRE’11), Lero, Ireland October 17-20 2011, pp. 325–334.
5. T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, “Mining Version Histories to Guide Software Changes,” IEEE Transactions on Software Engineering, vol. 31, no. 6, pp. 429–445, 2005.
6. A. Alali, H. H. Kagdi, and J. I. Maletic, “What's a Typical Commit? A Characterization of Open Source Software Repositories,” in Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC’08), Amsterdam, The Netherlands June 10-13 2008, pp. 182–191.
7. H. C. Gall, B. Fluri, and M. Pinzger, “Change Analysis with Evolizer and ChangeDistiller,” IEEE Software, vol. 26, no. 1, pp. 26–33, 2009.
8. R. Premraj, A. Tang, N. Linssen, H. Geraats, and H. van Vliet, “To branch or not to branch?,” presented at the ICSSP '11: Proceedings of the 2011 International Conference on Software and Systems Process, 2011.
9. H. Lee, B.-K. Seo, and E. Seo, “A Git Source Repository Analysis Tool Based on a Novel Branch-Oriented Approach,” proceeding of the 4th International Conference on Information Science and Applications (ICISA’13), Pattaya, Thailand June 24-26 2013, pp. 1–4.
10. M. Biazzini, M. Monperrus, and B. Baudry, “On Analyzing the Topology of Commit Histories in Decentralized Version Control Systems,” in Proceedings of the 2014 30th IEEE International Conference on Software Maintenance and Evolution, (ICSME’14), Victoria, British Columbia September 28 - October 3 2014, pp. 261–270.
11. L. Yu and S. Ramaswamy, “Mining CVS Repositories to Understand Open-Source Project Developer Roles,” in Proceedings of the 2007 International Working Conference on Mining Software Repositories (MSR’07), Minneapolis, MN, May 19-20 2007, 8 pages.
12. I. Skerret (2014, June 23). Eclipse Community Survey 2014 Results [Online]. Available: https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/
13. C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu, “The promises and perils of mining git,” in Proceedings of the 6th International Working Conference on Mining Software Repositories, (MSR’09), Vancouver, Canada May 16-17 2009, pp. 1–10.
14. S. Chacon and B. Straub, Pro Git, 2nd ed. Apress, 2014.
15. GitHub. (2015, June 23). GitHub homepage [Online]. Available: https://github.com
16. A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail, “CVSSearch: Searching through Source Code Using CVS Comments.,” in Proceedings of the 2001 International Conference on Software Maintenance (ICSM’01), pp. 364–373, 2001.
17. D. M. Germán, “An empirical study of fine-grained software modifications,” Empirical Software Engineering, vol. 11, no. 3, pp. 369–393, 2006.
18. I. Turnu, M. Marchesi, and R. Tonelli, “Entropy of the degree distribution and object- oriented software quality,” in Proceedings of the 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM’12), 2012.
19. A. Y. Sokolov, I. R. Golovko, and E. A. Cherkashin, “Extraction of thesaurus and project structure from Linux kernel source tree,” in Proceedings of the 2012 35th IEEE International Convention Information and Communication Technology, Electronics and Microelectronics (MIPRO’12), pp. 1088–1092, 2012.
20. A. Bachmann and A. Bernstein, “When process data quality affects the number of bugs: Correlations in software engineering datasets,” in Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR’10), pp. 62–71, 2010.
21. A. Tarvo, T. Zimmermann, and J. Czerwonka, “An integration resolution algorithm for mining multiple branches in version control systems,” in Proceedings of the 2011 27th IEEE Conference on Software Maintenance (ICSM’11), pp. 402–411, 2011.
22. G. Ghezzi, M. Würsch, E. Giger, and H. C. Gall, “An architectural blueprint for a pluggable version control system for software (evolution) analysis,” in the Proceedings of the 2012 2nd IEEE Workshop on Developing Tools as Plug-ins (TOPI’12), 2012, pp. 13–18.
23. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE’11), pp. 168–178, 2011.
24. M. Ahmed-Nacer, P. Urso, and F. Charoy, “Improving textual merge result,” 2013 9th International Conference Conference on Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), pp. 390–399, 2013.
25. C. R. Prause, “Maintaining Fine-Grained Code Metadata Regardless of Moving, Copying and Merging,” in Proceedings of the 2009 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’09), pp. 109–118, 2009.
26. E. Shihab, C. Bird, and T. Zimmermann, “The effect of branching strategies on software quality,” Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on, pp. 301–310, 2012.
27. D. M. Germán, B. Adams, and A. E.Hassan, “A Dataset of the Activity of the git Super-repository of Linux in 2012,” in Proceedings of the 2015 12th ACM-IEEE Working Conference on Mining Software Repositories (MSR’15), Florence, Italy, May 16-17 2015, pp. 470-473.
28. C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, “How do centralized and distributed version control systems impact software changes?,” in Proceedings of the 2014 36th International Conference on Software Engineering (ICSE’14), pp. 322–333, Hyderabad, India, May 2014.
29. D. M. Germán, B. Adams, and A. E. Hassan, “Continuously mining distributed version control systems: an empirical study of how Linux uses Git,” Empirical Software Engineering, vol. 21, no. 1, pp. 260–299, 2016.