Universität Rostock

Traditio et Innovatio

Analysis of Commit culture in open-source GitHub repositories

Name:	Sudha Vamanraj Hebsur
Matriculation number:	223100079
Submission date:	18.01.2026
Supervisor:	Prof. Dr. Regina Hebig Software Engineering University of Rostock
Reviewer:	M.Sc. Andreas Ruscheinski Computer Science Division University of Rostock

Master Thesis at Institute for Software Engineering
University of Rostock

Abstract

GitHub is one of the most widely used platforms for version control management, hosting millions of open-source software projects. The commit culture of a repository provides valuable insights into development practices and software evolution within a project. It demonstrates how teams communicate project release plans, discuss and resolve issues, design and implement features, and adhere to guidelines and best practices. It also provides patterns and insights that help teams identify shortcomings in their development workflow and formulate improved contribution guidelines. The insights are also useful in contexts where open-source repositories are forked and customised by users. Understanding commit relationships and tangling helps users keep their forks aligned with the original repository, in terms of bug fixes, features, and security updates, while still preserving their local customisations. This study highlights patterns in developer practices, such as making multiple commits for a feature, and how tangled the commits are with one another. In this thesis, I analysed commit culture in two open-source repositories of the Eclipse IDE and the Microsoft STL library. I used multiple information sources in the repositories, such as commit messages, pull requests, linked issues, comments, and commit authors, for the analysis. The gathered information was further used to identify related commits and form commit groups by appropriate reasoning. I also measured tangling degrees between code changes of commits at the file and function levels. The study found that the Eclipse repository exhibited a stronger multi-commit culture compared to the STL repository. The STL repository commits were found to be heavily tangled at the file level compared to the Eclipse repository commits. During the study, several information sources in the repositories were also identified that proved to be useful for the analysis.

Contents

List of Figures

1 Introduction

Version control is a software engineering practice that tracks and manages changes to software code [27]. Version control systems are software tools that enable teams to collaborate and coordinate work effectively in a distributed environment, and maintain easily trackable versions of source code. Version control systems have become an important component of software development. GitHub, in particular, is the largest and most popular platform for hosting source code. As of 2025, GitHub hosts 630 million repositories. 63% of these repositories are open-source or public repositories, with 1.12 billion contributions [8].

A change in a repository is called a commit. Developers save the changes into version control systems by making commits. The changes may be related to a bug fix, a new feature implementation, a documentation change, or an image update. This makes commit history a rich source of information to understand the development process in a team. Apart from code diffs, other information surrounding the commits, such as commit messages, pull requests, code review comments, discussions, labels and linked issues, collectively describes a commit culture in teams. It highlights how developers structure their work, document important information, coordinate with other developers, manage releases and follow project-specific contribution guidelines in an open-source repository.

Prior research in this area has primarily examined commits in the context of individual commits and automation of commit classification [16, 24]. This was studied for tasks such as bug prediction [2], effort estimation [1], and planning of software maintenance activities [16]. These studies rely only on commit messages and code diffs as information sources. However, modern repositories contain several other information sources that provide very useful contextual information related to commits. These information sources are not considered in the existing studies.

While existing studies focus on understanding individual commits, they provide limited insight into how commits interact with one another over time. Code changes introduced by different commits may overlap in the same parts of the source code. This overlap of changes is called tangling. Thus, analysing commits also involves understanding where and how the code changes overlap in commits.

In software engineering, tangling can refer to commit tangling [10, 11] or code tangling [17]. Tangled commits are defined as code changes to software that address multiple issues at once [10]. In the context of features, the code of features is often tangled with the base code and possibly with the code of other features [17]. This is called code tangling. In this study, I will address code tangling and refer to it simply as tangling. Tangling degree is the metric used to quantify tangling. It indicates the extent to which different features are mixed together in the same module, file, class or function.

1.1 Motivation

Understanding commit culture provides useful insights that can be used to identify challenges and shortcomings in development practices, tailor contribution guidelines to overcome shortcomings, improve feature traceability, and ease onboarding of new contributors. The insights are especially relevant in scenarios where original repositories are forked and customised by users. These forks need to be periodically updated with the original repositories without losing the customisations.

When a particular feature needs to be imported from the original repository, it is important to identify all commits related to the feature. This study provides insights into how related commits can be identified using information sources available in repositories. The number and size of commits in a commit group also indicate the complexity and size of a feature. Tangling degree can also be used as an indicator to understand the complexity of features. A higher tangling degree might increase the probability of inadvertently inducing bugs during fork updates. Based on the commit groups and tangling degrees, appropriate measures, such as testing, impact on other features, etc., can be taken for a smooth update of forks. Thus, identifying commit relationships and understanding tangling degrees can help simplify this update process.

1.2 Objectives

The main objectives of this thesis are: To investigate whether developers use multiple commits to implement a feature, bug, or enhancement and to measure file-level and function-level tangling between commits.

The corresponding research questions that guide this thesis are:

RQ1: Do developers use multiple commits for the same feature/change in the main branch in well-known open-source repositories?

Here, I will primarily study commits in the main branch of repositories. However, during analysis, I will review some commits related to pull requests that will be merged to the main branch. But the main focus is on the commits of the main branch.

RQ2: How tangled are commits with one another in terms of shared modified files and functions?

1.3 Methodological Overview

To answer these research questions, I selected two repositories: the Eclipse platform and the Microsoft STL. The Eclipse platform repository contains the source code for a well-known integrated development environment (IDE), popularly used for Java-based software development. The Microsoft STL repository provides Microsoft's implementation of the C++ Standard Library, which is distributed with the Visual Studio IDE. I selected one release for each of the two repositories. I analysed each commit within the release manually using information sources, such as commit messages, source code

diffs, pull requests and their discussions, linked issues and bug reports, labels, commit authors, and commit dates.

I then identified related commits and formed commit groups based on reasoning such as feature implementation, scenarios, infrastructure changes, or syntactic similarity. Next, I measured file-level and function-level tangling degrees to quantify overlap between commits.

1.4 Structure of the Thesis

This thesis is further structured as follows: First, I briefly discuss the related work on commits, commit classification, and code tangling in Chapter 2.

Chapter 3 describes a detailed methodology followed during this thesis. It describes the repositories examined, the information collection process, and the approach to grouping related commits, how tangling degrees were measured and visualised. This chapter also explains how some of the steps in the process were automated and verified.

In Chapter 4, I present the results of this study on commit groups, information sources used for repositories during commit analysis, and tangling degrees for both repositories.

Chapter 5 presents a discussion of the results, where I interpret the findings, compare them with existing literature, address limitations and threats to validity, and conclude by highlighting the significance of the study and future work.

In Chapter 6, I conclude the thesis report.

2 Related work

In this thesis, I analyse commit culture in open-source repositories, with a particular focus on whether developers use multiple commits to implement a feature, bug fix or change and how tangled commits are in terms of modified files and functions. Prior research has studied commits with different perspectives and objectives than those addressed in this study.

2.1 Commit analysis and classification

Commits are widely studied in the context of software maintenance and evolution for tasks such as bug prediction [2], planning of software maintenance activities [16], effort estimation [1] and so on. A common objective in this area is commit classification, where commits are categorised based on the type of maintenance activity they address.

Different categories for commit classification have been proposed in the existing literature. Commits were classified based on maintenance activities: corrective: fixing bugs, perfective: system and design improvement, and adaptive: adding new features in the study by Levin et al. [16]. Dos Santos et al. [24] further extended the categories in their studies: corrective, perfective, features, non-functional and unknown. Here, they adopted the definitions for corrective, perfective and features from the study by Levin et al. Additionally, the non-functional category was associated with documentation and non-functional requirements, and the unknown category addressed files generated by version control systems. Benett et al., in their study related to software maintenance and evolution [3], adopted the following categories for maintenance activities: adaptive: changes in the software environment, perfective: new user requirements, corrective: fixing errors, and preventive: preventing problems in the future. Sabetta et al. aimed to automatically identify commits that address security-related code changes, such as fixes for security vulnerabilities [23]. So, it can be seen that there is no consensus related to categories for classification followed in the literature [24].

Several studies also explored automation of commit classification. Levin et al. [16] studied commits from 11 GitHub repositories and automated commit classification. They considered 3 maintenance activities as categories for commit classification: corrective: fixing bugs, perfective: system and design improvement, and adaptive: adding new features. They manually created a labelled dataset, trained and evaluated predictive models using source code diffs and commit messages for commit classification. They trained 3 models: one using only commit messages, one using only source code diff, and another using both commit messages and code diff as inputs for classification. It was found that models showed significant improvement in performance when both commit messages and code diffs were used as input.

Barnett et al. [2] studied commits from 342 Java-based GitHub repositories for defect prediction. They used Just-in-time (JIT) defect predictor models to predict how likely commits are to introduce defects in the future. In general, JIT models use only source code changes for defect prediction. In this study, the relevance of commit messages was studied. The JIT models used commit message length and meaningful content as additional input. The results showed that longer or more detailed commit messages tend to carry useful indicators for defect prediction. However, the content of commit messages is an even stronger indicator for defect prediction.

Limitations of existing commits related studies While existing studies demonstrate the usefulness of commit messages and source code diffs for commit classification, bug detection, maintenance activities planning, they treat commits as independent entities [15, 16, 23, 24]. These studies rely only on commit messages and code diffs as information sources.

Modern version control systems like GitHub, however, provide other rich information sources related to commits, such as pull requests, labels, commit authors, and linked issues. These information sources are underutilised in existing studies. Moreover, existing studies do not explicitly investigate whether multiple commits contribute to a single feature, bug fix or change. They do not study commits as a group of related commits.

In this study, I will analyse individual commits and create groups of related commits that contribute to a particular feature, bug or change. I will also introduce new categories for commit groups. Understanding commit groups provides useful information related to feature traceability and types of commit groups. This information can be used during fork updates against original repositories to include new features and fixes.

2.2 Code tangling

Another important aspect of commit analysis is code tangling. In software engineering, tangling can refer to commit tangling [10, 11] or code tangling [17]. Herbold et al. defined tangled commits as code changes to software that address multiple issues at once [10]. In the context of features, Liebig et al. [17] mentioned that the code of features is often tangled with the base code and possibly with the code of other features. This is called code tangling. In this thesis, I will focus on code tangling.

Tangling degree is the metric used to quantify the extent of tangling. Krüger et al. [14] annotated the location of features in the source code using feature names. Then they measured the tangling degree as the number of other features included in a considered feature. Unlike these existing studies that measure the tangling degree across the entire source code in terms of features, I will evaluate the tangling degree within the code changes introduced by commits. This is to adhere to the focus of my study on commits.

I will assess tangling in two levels: file-level and function-level tangling. File-level tangling degree for any given commit indicates how many other commits have modified the same files as that commit. Function-level tangling degree for any given commit

indicates how many other commits have modified the same functions as the chosen commit. Function-level tangling degree offers a more granular measure than file-level tangling degree and offers deeper insights into code change interactions.

2.3 Fork maintenance and updates

In software development, it is a common practice to fork existing open-source repositories and customise the code according to one's requirements. However, over time, the original repositories receive new features, security patches and bug fixes. Updating the forks with new features from the original repositories becomes a tedious task due to merge conflicts. It is also important not to lose the fork customisations during updates.

Marcondes et al. [19] provide an approach to systematically update forks against the original project or repository. [1] The study emphasises using domain-specific language (DSL) for customising code in forks, without actually modifying the original source code. It uses DSL instructions, such as replace file, replace unit, add unit, remove string, etc., that are applied to the original source code to generate the required customisation. However, this idea provides guidelines on how to implement customisations in new forks without changing the original source code. It does not solve the problem for the forks that have existed for several years with many customisations.

My study complements the study related to fork updates [19] by providing insights into identifying commits related to features, and information sources that can be used for this identification. It also gives information about the tangling degree of source code diffs. This helps to evaluate the complexity of a feature and identify measures to take to avoid inadvertent bugs during the update process.

To summarise, prior studies have extensively studied commits for classification, defect prediction, and maintenance activities planning, basically considering commits as individual entities and relying on limited information sources. This thesis extends existing work related to commits by analysing commit culture through the concept of considering related commits as a group contributing to a feature or change, and measuring tangling degrees at file and function levels, utilising multiple information sources available in modern version control systems.

[1] This paper was originally published in Portuguese. Translation tools were used to study this paper.

3 Methodology

For this study, I will select two open-source repositories and analyse commits made in one release for both repositories. To address RQ1, I will analyse commits to identify if there are related commits. The definition of related commits can be found in section 3.2.1 below. First, I will gather all commits that belong to one release. As the focus of this study is on code changes, I will exclude commits related to documentation changes, version bumps, icon or image updates and configuration changes. After filtering out commits, I will analyse the remaining commits in two passes to identify commits that are related to each other. In the first pass, I will use commit messages and their extended description as an information source to understand if commits are related. In the second pass, I will use other information sources such as code diffs, linked issues, pull requests, labels, and commit authors to understand more about commits and further form groups of related commits. After analysing the commits, I will have the results for RQ1 in the form of a table that shows commit groups and their commits. All commits in a group contribute to the same feature or change. These results present the extent to which developers use multiple commits for the same feature/change, in terms of the number of analysed commits and commit groups formed in both repositories.

To address RQ2, I will measure tangling degrees for code changes of commits at the file and function levels. First, I will generate code diff files for all commits. To measure file-level tangling, I gather a list of modified files for each diff file by regular expression search using the file header format present in diff files. I then compare each commit pair to find the list of files that are modified by both commits of the pair. I then visualise the file-level tangling degree between commit pairs in the form of a heat map, with cell values indicating the number of common modified files for a commit pair. I will also plot a bar graph to show the number of commits tangled with a given commit that modified the same files as those of a given commit.

I will follow a similar process to measure function-level tangling between commits. I gather a list of modified functions for each diff file by regular expression search using the hunk header format present in diff files. Then, for each commit pair, I will find a list of functions modified by both commits in the pair. The function-level tangling degrees will be visualised using heat maps and bar graphs. Figure 3.1 presents an overview of the steps involved in the methodology.

3.1 Selection of Repositories

The criteria that guided my selection of repositories are described in this section. GitHub was my conscious choice of version control system for this study, as it is the largest and most popular platform for hosting open-source code, with approximately

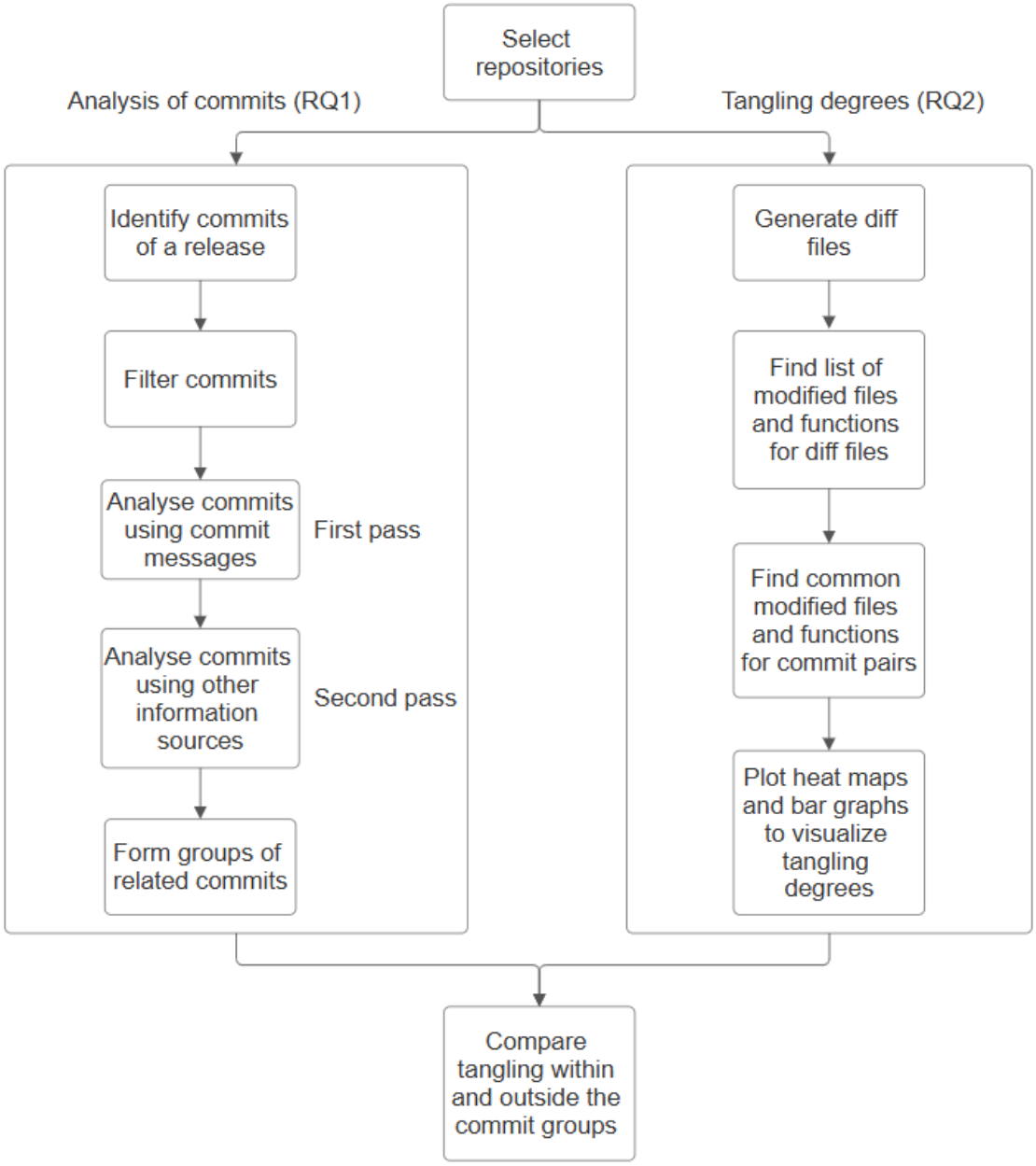

Figure 3.1: Overview of methodology

630 million open-source or public repositories as of 2025 [8].

Number of contributors, programming language, and size of repositories in terms of number of commits per release are the properties I considered during the selection of repositories. Since the focus of this study was the analysis of commit culture in larger communities, it was important to have at least 100 contributors for the selected repositories. To introduce variety, I chose two repositories with different programming languages, while also considering my familiarity with those programming languages, as analysing source code diffs was a major task in the study. I further selected the releases with fewer than 100 commits to keep the effort of analysing the commits to an achievable level, as it was a manual task. Thus, two repositories meeting these criteria were chosen.

Based on these criteria, two repositories that I selected for the study are: Eclipse platform [6] and Microsoft STL [20]. The Eclipse platform repository contains the source code for a well-known integrated development environment(IDE), popularly used for Java-based software development. It has 194 contributors, 148 forks and primarily consists of a Java code base(96%). Based on the branches and timeline of the commits, it can be noticed that the community follows a 3-month release cycle. For this study, I chose the commits between the branches R4_33_maintenance and R4_34_maintenance, consisting of 84 commits.

The Microsoft STL repository provides Microsoft's implementation of the C++ Standard Library that is distributed with the Visual Studio IDE. The repository has 260 contributors, 1.6K forks and primarily consists of a C++ code base (99%). All development occurs on the main branch, while releases are marked with tags. For this study, I focused on commits between the release tags vs-2022-17.13 and vs-2022-17.14, which consisted of 81 commits.

3.2 Analysis of commits

The goal of this section is to identify commits that are related. This identification forms the basis for addressing research question RQ1. In the following subsections, I will define the criteria I used to determine whether commits were related. I will then introduce how the relevant commits for the selected branches were identified and filtered. Then, I will describe what information was collected for each commit. Finally, I will explain how related commits were identified and commit groups were formed, providing the information that was captured during the process.

3.2.1 Definition of related commits

I considered two commits related if:

- the commits addressed the same feature, issue or bug.
- the commits addressed different parts of the same scenario.
- the code changes in the commits were syntactically similar.

- the commits addressed infrastructure changes for the same component.

Syntactically related commits are commits that modify the same syntactic elements in source code. These commits are not necessarily related to a common product feature, but the code changes follow a similar syntactic pattern.

3.2.2 Overview and filtering of unified diff files

After selecting the repositories, the next step was to analyse commits. For the Eclipse repository, I generated a unified diff file for the selected branches R4_33_maintenance and R4_34_maintenance to obtain an overview of files and lines of code (LOC). Since I aimed to understand code commits contributing to features and issues in this study, I primarily focused on code files (.java files). I excluded configuration, infrastructure and properties-related files such as pom.xml, manifest.mf, .gitignore, .setup, and others.

For the STL repository, I generated a unified diff file for the selected release tags vs-2022-17.13 and vs-2022-17.14 to obtain an overview of the files and lines of code (LOC). Although some files were .cpp and .hpp files, there were many files without any extensions. Each component of the STL Library has a header file with its name, but without any extension in the path stl/inc/. Since these header files contained several function definitions, I included these files for the analysis. As the focus was on source code, I excluded other configuration, documentation and test files.

3.2.3 First pass

For a systematic analysis of commits, I created a table containing commit information, an excerpt of which is shown in Figure 3.2. The analysis was conducted in two passes.

Identification of relevant commits

The goal of this step was to identify commits that belonged to the chosen release. I opened the Code tab of the repository in a browser, selected the older of the two branches chosen for the study, for example, the branch R4_33_maintenance for the Eclipse repository. I then clicked on the Commits as shown in the Figure 3.3, which displays all commits in reverse chronological order, with the most recent commit listed first. The first commit ID from this list represents the latest commit made to the R4_33_maintenance branch. I noted this commit ID, say X, for further use, because the next commit after this would correspond to the first commit for the next newer branch.

Next, I switched to the newer branch, R4_34_maintenance. I scrolled through the list of commits until I located the commit ID X. I started with the commit immediately above the commit ID X, as this would be the first commit for the R4_34_maintenance branch and continued in the reverse order of the commit list. I followed the same process to identify the relevant commits for the chosen STL repository release as well.

Commit Id	Included commits sl. No.	Commit Description	Status	Comments	File list
5672a4f		Splash Screen for 4.34 (2024-12) (#1534)	Ignored	Image file	
aadc046		Update for release 4.34 (#1535)	Ignored	version change for pom and manifest files	
1870bbe		Version bumps for 4.34 stream	Ignored	version change for pom and manifest files	
b2fbfa6		Comparator errors in I20240904-0240	Ignored	Text file	
2fb8958		Comparator errors in I20240904-1800 (#1540)	Ignored	Text file	
c0973ff	1	ElementTree#includes: relax synchronized access	Included		resources/bundles/org.eclipse.core.resources/src/org/eclipse/core/internal/watson/ElementTree.java
4032c40		version bump	Ignored	version change for pom and manifest files	
9d9bf70	2	ContentTypeCatalog#removeContentType: add missing synchronize	Included	somehow related to c0973ff ??	runtime/bundles/org.eclipse.core.contenttype/src/org/eclipse/core/internal/content/ContentTypeCatalog.java
7e1f60c	3	[Debug] Show elapsed Time in Console & Properties	Included		debug/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/preferences/DebugPreferencesMessages.java debug/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/preferences/DebugPreferencesMessages.properties debug/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/preferences/ProcessPropertyPage.java debug/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/views/console/ConsoleMessages.properties debug/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/views/console/ProcessConsole.java
a4eb5f3		version bump	Ignored	version change for pom and manifest files	
70607f3	4	Migrate resources regression tests to JUnit 5	Included	Merged with commit b432795 (JUnit)	49 files changed
8330656		org.eclipse.core.tests.resources for 4.34 stream	Ignored	version change for pom and manifest files	
ed1bcb9	5	CharsetManager.internalGetCharsetFor reuse calculated key	Included		resources/bundles/org.eclipse.core.resources/src/org/eclipse/core/internal/resources/CharsetManager.java
e2f91dc		version bump org.eclipse.core.runtime.feature	Ignored	version change for pom and manifest files	
870681		Revert tycho-packaging-plugin config in	Ignored	pom file change for maven dependency	
ccbfc48	6	IOConsoleTests: try to show all errors	Included	Merged with commit 283f954. ~~Not related features, merged with larger diff commit~~ Related to same issue as in commit 283f954	debug/org.eclipse.debug.tests/src/org/eclipse/debug/tests/console/IOConsoleTests.java
b12f857		version bump	Ignored	version change for pom and manifest files	

Figure 3.2: An excerpt of a table showing commits and their information

Information collection

For each commit, I collected information such as commit ID, commit description, modified files and entered it into the table (Figure 3.2). Some large commits consisted of several modified files. For example, 49 files were modified for one JUnit commit in the Eclipse repository. In such cases, I just mentioned the number of files that were modified, rather than listing 49 files.

Filtering of commits

Since the focus of the study was on code changes, I excluded commits related to documentation changes, version bumps, icon or image updates and configuration changes. A version bump refers to a software versioning process, where the version of a software is increased to a new unique value for every software release. There were many such version bump commits in the Eclipse repository, where versions were modified in manifest.mf files. When the commit messages included the words suggesting documentation, version bump, configuration, etc, I further looked at modified files and code changes in those commits. After verifying that the changes were not related to code functionality or features, I filtered such commits out. I also entered this filtering information for each commit in the table (Figure 3.2) under the column Status, indicating whether each commit was included or excluded in the study.

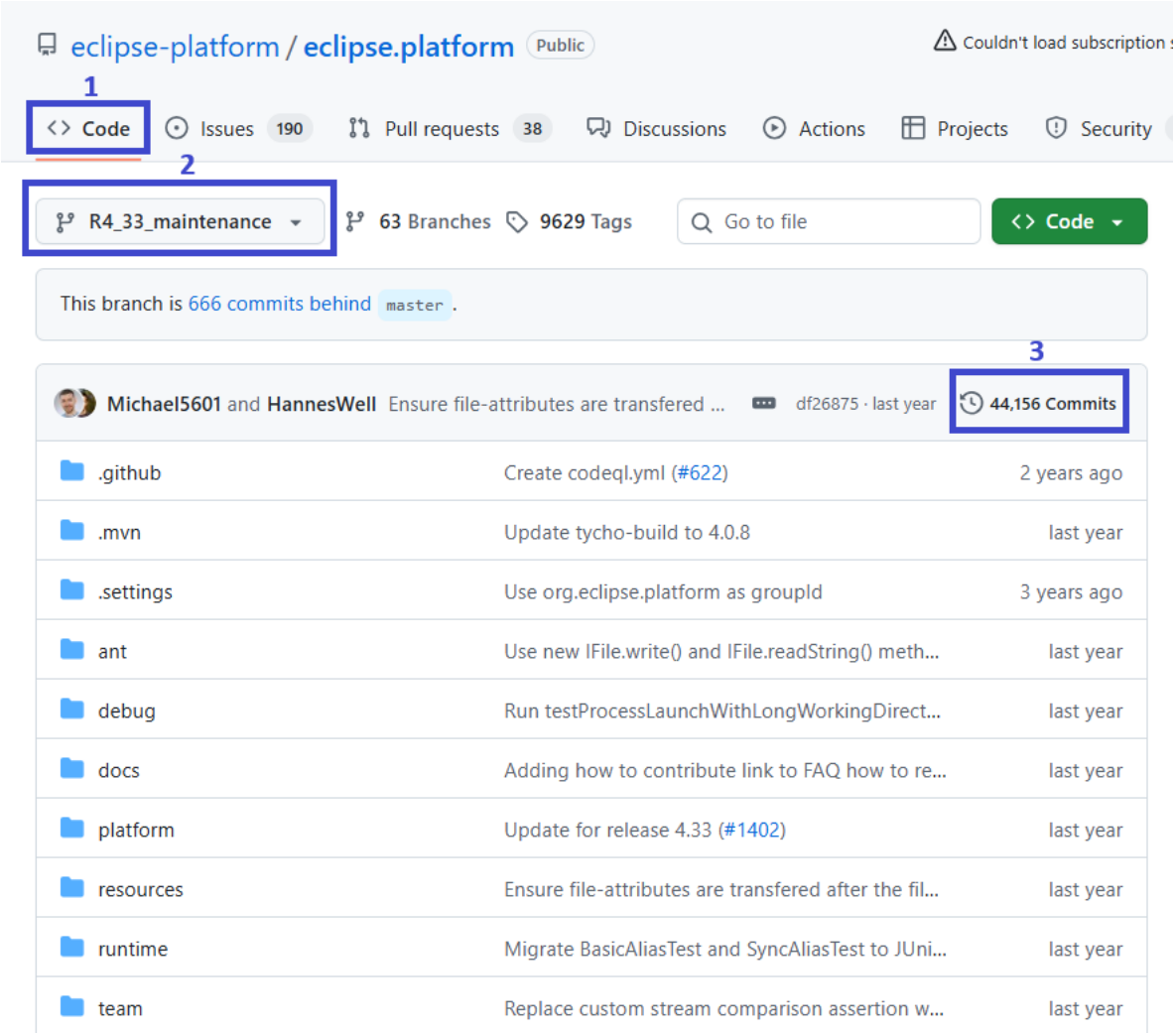

Figure 3.3: Steps to view the commit list for a branch on GitHub

Identification of related commits

During the first pass, I clicked on each commit to read the commit messages and descriptions and marked the commits that appeared to be related. Some related commits were clearly and easily identifiable in this pass. Others were less obvious and not immediately apparent from reading just the commit messages and their descriptions.

In some cases, authors mentioned the issue ID that the commit addressed in the commit message description. This information also helped me identify related commits, i.e., if the same issue ID is mentioned for different commits, it indicated that they addressed the same feature or issue.

The first pass analysis took approximately one week for each repository.

Commit group formation after the first pass

After completing the first pass, I created another table to group all related commits. An excerpt of this table is shown in Figure 3.4. In the remainder of the report, I will refer to a group of related commits that were identified as a "commit group".

The table consisted of information like commit group number, commit group ID, commit group name, LOC, number of modified files, number of individual commits within the commit group and comments. The commit group number is a serial number. I added the commit group ID and the commit group name for reference to easily identify commit groups. For 2 related commits, I considered the commit ID and commit message of the first commit within each commit group, as the commit group ID and commit group name of that commit group, respectively. LOC is the total number of LOCs of individual commits of each commit group. Similarly, the number of files is the total number of files modified in individual commits of a commit group. The number of individual commits indicates the number of related commits in a commit group. My observations, reasoning and notes regarding why I considered certain commits to be related were documented in the comments column.

Since some commits were not related to any other commits, I considered such independent commits as their own commit group. So, the table entries for such cases in the Figure 3.4 show the groups with a number of individual commits as 1. For the Eclipse repository, 21 commit groups were formed after the first pass.

3.2.4 Second pass

The goal of the second pass was to look into code diffs of commits and other information sources to identify new commit groups, if present. The information sources that I used during the second pass are code diffs of commits, pull requests, discussions, comments, labels, commit authors and linked issues in the descriptions. The second pass also helped in correcting mistakes due to oversights during the first pass.

During the second pass, I did a deep dive into the commits and analysed each commit by looking into the code diff to understand the code changes. I read data from other information sources to understand more about the context and scenarios related to commits. I documented additional observations and reasons why particular pairs or

Commit group no.	Commit group ID	Commit group name	LOC	No. of files	No. of individual commits	Comments
1	c0973ff	ElementTree#includes: relax synchronized access	48	1	1	Can be indirectly related. One removes synchronize and other adds it to a function. Authored/committed by same person EcljpseB0T authored and jukzi committed. Maybe EcljpseB0T is automated bot? Also the comment says, Found by code inspection - i wondered why internalFindContentTypesSorted() is synchronized.
2	9d9bf70	ContentTypeCatalog#removeContentType: add missing synchronize	72	1	1	
3	7e1f60c	[Debug] Show elapsed Time in Console &	171	5	2	
4	ed1bcb9	CharsetManager.internalGetCharsetFor reuse calculated key	80	1	1	
5	75d509d	Add ZipEntryStorage.getArchivePath() and deprecate its getArchive()	103	2	1	
6	58dde88	TextMergeView Unhandled IndexOutOfBoundsException at lines height calc	141	1	1	
7	ea7fa2a	Breakpoint Enable All feature + Code review	355	9	3	
8	ad5c409	AntLaunchDelegate.runInSeparateVM: wakeup waiter	49	1	1	
9	b0bb6df	API: add IWorkspace.write(Map<IFile, byte[]> ...)	409	4	1	Create files in parallel. Use case: clean build, delete all and then create .class file for each .java file. Also changes made by the same author
10	283f954	ConsoleTests: do not wait for all Jobs	495	11	3	
11	b432795	Migrate Ant core tests to JUnit 5	8047	82	8	
12	6d514be	[performance] LocalFile: delete Files of Directories in parallel #1461	322	4	2	issue #1461: Project/Clean... takes much time to only delete the output
13	007ee10	Minor code simplifications in LaunchConfiguration	120	1	1	Minor code refactoring like create object and then assign is replaced by one line, etc.
14	e14565e	Add IContextFunction.ServiceContextKey OSGi component property type	55	1	2	Adding OSGi services. Changes made by same author. Properties maybe not directly related (like in case of EventTopics), but similar type of code change.
15	be89385	Use @ServiceRanking component property type annotation in e4.core.tests	171	11	1	
16	392c8ae	Use @EventTopics property type annotation in TipsStartupService	38	1	1	
17	3cef3f3	Add ISystemInformation.Section OSGi component property type	196	7	1	
18	78c13b5	Moving TabFolderLayout into SWT to have a single copy.	229	6	1	
19	c2dc063	Fix RSA encryption bit size to 2048	36	1	3	
20	578665f	Stop AntSecurityManager from impl methods removed in Java 10	366	2	1	
21	b4bff60	Revert "Make LaunchManager's IResourceDeltaVisitors more light weight"	250	1	1	

Figure 3.4: An excerpt of a table showing commit groups and their information

groups of commits appeared to be related, under the comments column in the excerpt of the table shown in figure 3.2, which would be later referred to create commit groups. I colour-coded the new commit groups found during the second pass, using the same colours for the column Commit group ID. This can be seen in the Figure 3.4.

During the second pass, I analysed 37 commits for the Eclipse repository, which took approximately one week, and 71 commits for the STL repository, which required around two weeks. The STL source code was particularly challenging to analyse. Since it is the source code for a library, it contains several template definitions, multiline function headers spanning up to 3 lines, multiple overload functions, operator-overloaded functions, nested preprocessor statements and so on. The scenarios described in the issue and commit descriptions were highly specific to particular situations, which required me to read more about concepts, issues and documentation to gain domain knowledge.

Analysing the code diff was particularly useful in gaining a better understanding of the intentions behind the commits. The information collected in the first pass was enough to identify some related commits. However, new commit groups emerged only after the detailed analysis during the second pass. For the Eclipse repository, I identified 7 and 1 commit groups during the first and second passes, respectively. During the second pass, I also identified 4 commits that belonged to older commit groups formed during the first pass. For the STL repository, I identified 4 and 5 commit groups during the first and second passes, respectively.

All the commit groups and their related commits are further discussed in the Results section 4.2

3.3 Tangling degree of commits

This section addresses research question RQ2. In this section, I will describe different levels of tangling degrees measured in this study. I will then outline the process followed to develop scripts. These scripts measure and visualise file-level and function-level tangling through heat maps. Next, I will present different approaches I took to measure function-level tangling for both repositories. Finally, I will explain the process of how I derived the tangling degrees from the cell values of the heat maps and plotted them as bar graphs.

Instead of measuring the tangling degree across the entire source code in terms of features, I focus on evaluating the tangling degree within the code changes introduced by commits. I assess the tangling degree in two levels: file-level and function-level. File-level tangling degree for any given commit indicates how many other commits have modified the same files as that commit. Function-level tangling degree for any given commit indicates how many other commits have modified the same functions as the chosen commit. Function-level tangling degree offers a more granular measure than file-level tangling degree.

3.3.1 Presentation of tangling degrees

This section describes how file-level and function-level tangling degrees for commits are presented in this study. I used heat maps to visualise file and function level tangling, where the cell values indicate the number of common modified files and functions between commit pairs. I also plotted bar graphs showing the tangling degrees for each commit.

Heat maps showing file-level tangling The heat maps show file-level tangling between commits. Each cell value represents the number of common modified files for the corresponding commit pair on the x and y axes.

Heat maps showing function-level tangling The heat maps illustrate function-level tangling between commits. Each cell value represents the number of common modified functions for the corresponding commit pair.

Bar graphs showing file-level tangling Bar graphs show, for each commit, how many other commits it is tangled with, based on common modified files. For each commit on the x-axis, its corresponding value along the y-axis indicates how many commits share at least one modified file with that commit.

Bar graphs showing function-level tangling Bar graphs show, for each commit, how many other commits it is tangled with, based on common modified functions. For each commit on the x-axis, the corresponding value along the y-axis represents the number of commits that share at least one modified function with that commit.

3.3.2 Generation of diff files for the Eclipse repository

To calculate the tangling degrees, I generated a diff file for each commit. I performed this step manually for the Eclipse repository. I used the git command, shown below, to generate a diff of a particular commit, 9d9bf70 in this example. This command gives the diff with respect to its immediate previous commit and redirects the output to a newly created file named 1.diff

```
$ git diff 9d9bf70^ 9d9bf70 > 1.diff
```

Since the focus was only on code files, I modified the git command to include only .java files for the Eclipse repository as below:

```
$ git diff 9d9bf70^ 9d9bf70 -- '*.java' > 1.diff
```

This way, I manually generated 37 diff files for the Eclipse repository, one diff file corresponding to each commit.

3.3.3 Generation of diff files for the STL repository

Since there were 71 included commits for the STL repository, I avoided the manual process of generating diff files for each commit. I automated the generation of diff files by developing a script. The script would run a git diff command for each commit ID and generate a corresponding diff file. I developed this script using a large language model(LLM), ChatGPT. I gave a prompt to generate a script that reads a specific sheet from an Excel file. For each row, the script reads the columns containing the commit serial number and commit ID. Then the script runs a git diff command using the commit ID and redirects the output to a newly created diff file, named after the commit's serial number. For example, for a row with a value of 1 as commit serial number and 2cef491 as commit ID, the script runs the following git diff command and generates a diff file named 1.diff.

```
$ git diff 2cef491^ 2cef491 > 1.diff
```

While working with the STL repository, I noticed that every code change for bug fix or feature implementation also included test file modifications to ensure test coverage for new code changes. As my focus was on code changes, I excluded the test files from diffs. The git diff option that I used for the Eclipse repository to just include .java files by specifying the file extension was not applicable for the STL repository for two reasons: the STL repository had several code files without any file extension. For example, around 130 files in the location stl/inc/ were code files without any extension. Also, the test files were with .cpp extensions. So, I used a different approach to exclude the test files by filtering based on their path (tests/). I added an exclude flag with the test files' path to the git diff command.

```
$ git diff 2cef491^ 2cef491 ':(exclude)tests/' > 1.diff
```

Script verification

During verification, I read the script. I checked the output folder to see how many diff files were generated. I noticed that file names were of the format 1.0.diff. The serial numbers from the Excel sheet were interpreted as floats, resulting in output diff files being named as 1.0.diff. To maintain consistency with the file naming format I used during manual generation for the Eclipse repository, I updated the script to convert the values to integers.

Additionally, I noticed that the number of generated diff files did not match the count of included commits. Some commit IDs, which consisted of only integers and a letter 'e', for example 0372e78, were being converted to an exponential number format by default Excel formatting to 3.72E+80. Some commit IDs, which consisted of only integers and started with a zero, for example 0190042, were being formatted by Excel, omitting the leading zero to display the value as 190042. To prevent these formatting issues, I enclosed such commit IDs in quotes. The script failed to generate the diff files for these commit IDs with quotes. So, I modified the script to handle the quotes.

I verified the results from the script by manually verifying the content of the generated diff files for 10 randomly chosen commits. I also verified that 71 diff files were generated, one for each commit. The script is available in my GitHub repository [26], which accepts paths to the Excel file, the local repository and the output directory to save the generated diff files.

The prompts used for developing the script are given in the Appendix 7.1.1

3.3.4 File-level tangling

Once the diff files were generated, I proceeded to measure the file-level tangling degree. I developed a script that calculates the file-level tangling for commits and visualises the results in the form of a heat map. Every modified file in a diff file is indicated by a file header. For example, below is the format of a file header, indicating that the stl/inc/___msvc_string_view.hpp file is modified:

```
diff --git a/stl/inc/__msvc_string_view.hpp b/stl/inc/__msvc_string_view.hpp
index aeae9bd2..468378cb 100644
--- a/stl/inc/__msvc_string_view.hpp
+++ b/stl/inc/__msvc_string_view.hpp
```

This file header format can be used to extract a list of modified files from a diff file using a regular expression search. In this way, the modified files list can be collected for all commits using their corresponding diff files. I used ChatGPT to develop the script, which is available in my GitHub repository [26]. The script takes the path to the diff files (generated in the previous step) as input, extracts a list of modified files for each diff file, finds the common modified files for each pair of commits and finally visualises the results using a heat map.

Script verification

During verification, I noticed that all the diagonal elements in the heat map had large values. This was due to comparing the commit with itself to find common modified files. I fixed the script to ignore diagonal values further using a prompt.

To verify the script, I printed the modified files list for each diff file and manually cross-checked it against the content of 5 diff files, including one large commit that modified 49 files (Commit 4 in the Eclipse repository). Additionally, I verified the generated heat map by printing the common modified files for each commit pair and manually comparing them against the contents of the diff files for 5 commit pairs, which showed non-zero values in the heat map. I also verified 2 instances of zero values from the heat map, to confirm that there were no common modified files among those commit pairs. This verification process was carried out for both repositories.

The prompts used for this script development are documented in the Appendix 7.1.2

3.3.5 Function-level tangling

A block of code change in a diff file is called a hunk [9]. Every hunk is preceded by a hunk header of a specific format, as shown below, which usually contains the function name.

To measure function-level tangling degrees, I initially attempted to extract modified function names from hunk headers in the generated diff files. However, a hunk header does not always give the actual function name that is being modified [7, 25]. Git uses a regular expression to match a line that will be used as a hunk header text. Due to regular expression matching, several hunks that were not part of any function used text that was not a true function name. For example, code changes in the public or private section of classes, as well as import statements at the beginning of files, are not part of any function. However, hunk headers provided block names such as "public" and "private". As this behaviour did not meet my requirements of function-level tangling, I discarded this approach of using hunk headers for this purpose.

```
@@ -2341,9 +2341,8 @@ _NODISCARD constexpr
auto bind_front(_Fx&& _Func, _Types&&... _Args) {
```

Function-level tangling for the Eclipse repository using JavaParser library

I used parsing libraries that parse source code files and list all functions. Since the source code languages were different for Eclipse and STL repositories, I could not use the same parsing library for both. I used JavaParser [12] for the Eclipse repository with Java source code. The parser takes a source code file and a line number as input and gives the function name that line number belongs to. I utilised one of the Maven quick-start projects, javaparser-maven-sample [13], referenced in the README documentation. I cloned this sample project to my local machine and replaced the file LogicPositivizer.java with a new file FindFunctionForLine.java. It takes a file name and a line number as input and returns the function name that the line number belongs to, using the JavaParser libraries. I modified the pom.xml to replace the mainClass parameter from LogicPositivizer to FindFunctionForLine. After compiling and packaging the project, a JAR file named javaparser-maven-sample-1.0-SNAPSHOT-shaded.jar was generated in the target folder of the Maven project.

I wrote a small Python wrapper to use the generated JavaParser JAR file. This wrapper accepts a file and a line number as input and returns the name of the function that contains the specified line. I then developed a Python script using ChatGPT that automates the process of identifying modified functions from the diff files for all commits. The script takes the paths to an Excel sheet (containing commit IDs), the local Eclipse repository and the folder containing diff files as input.

For each row in the Excel sheet, the script reads the serial number and commit ID, finds the corresponding diff file from the folder using the serial number. It then parses the diff file to find all hunks, using the hunk header format regular expression search and collects the line numbers of the first line of all hunks. Using the git show command, the script retrieves the version of the file that existed at that specific commit ID and creates a temporary file with that content. This temporary file and the line numbers of the hunks are passed to the JavaParser wrapper function to obtain the modified function names. This process is repeated for all hunks across all commits. The script then calculates the number of common modified functions in common modified files for each pair of commits and visualises the results in the form of a heat map. The script is available in my GitHub repository [26].

Script verification During verification, I noticed that the script failed to find diff files from the local folder. This was because the script constructed filenames using serial numbers from an Excel file, which resulted in a wrong format. I fixed the filename format by converting the serial numbers to int and adding a .diff extension to the filename.

I verified the non-zero values in the heat map by manually counting the number of common functions in common files for 4 pairs of commits. Additionally, I verified 2 instances of zero values from the heat map to ensure that there were no common functions between those pairs of commits.

The prompts used for this script development are mentioned in the Appendix 7.1.3

Function-level tangling for the STL repository

Since the STL repository consists of C++ source code, I used the Clang library, which is a part of the LLVM Compiler Infrastructure [18], to parse files and extract modified function names. However, the results were not accurate when I verified. This inaccuracy was primarily due to extensive use of #ifdef and #ifndef preprocessor directives in the STL codebase. The Clang parser considered the code changes in true conditional blocks, while it ignored the code changes in false blocks. Thus, I manually identified common modified functions for the STL repository.

First, I determined the total number of hunks in the STL repository across all 71 commits by performing a regular expression search in 71 diff files. There were a total of 797 hunks. However, there were 3 large commits (commits 49, 55, 60) which accounted for 398 hunks (half the total number of hunks). To make the manual process feasible, I excluded these three commits from the process. Additionally, I used the file-level tangling heat map for the STL repository as a reference. I focused only on the non-zero entries in the heat map. For each commit A, I noted a list of commits B,C,D, and so on, that had non-zero entries with respect to commit A in the heat map. I then manually compared commit A with each commit from the list, looking through code changes to identify the common functions that were modified. I noted the common function names down in a table, an excerpt of which is shown in Figure 3.5. This way, I exhaustively compared all commits with non-zero cell values in the file-level tangling heat map. Some commits took around 1 hour to compare against all file-tangled commits. Overall, this manual process took approximately 1 week to complete.

I developed a Python script to visualise this function-level tangling in the form of a heat map using ChatGPT. The script reads commit IDs, tangled commits, and tangled functions count from respective columns. It then generates a heat map by marking each commit ID against its tangled commits, using the tangled functions count as the cell values. The script is available in my GitHub repository [26]. The prompts used for developing the script are documented in the Appendix 7.1.4

I manually verified all cell values of the generated heat map against the columns showing tangled_commits and tangled_functions_count in the table(Figure 3.5)

commit_no	commit_id	common_files	common_functions	Comments	Tangled_commits	Tangled_functions_count
1	2cef491	stl/inc/xcall_once.h	#if defined at line 43	tangled with 29, as it is revert commit.	29	1
2	126f4eb		_Traits_find_ch			
3	059a1b0					
4	b60bb78		_Atomic_storage at 556 _Atomic_storage at 687 _Atomic_storage at 794 _Atomic_storage at 900 _Atomic_storage at 1006 _Atomic_storage at 1134 atomic at 2147 atomic at 2149	check with 44		
5	d40e8c0			shares two common files with different commits.		
6	0053a14					
7	a8fea5b					
8	abefd5e		seed at 833			
9	649a4f2		xmemory file: _Construct_in_place_by_deref _Emplace_back_deref _Emplace_back_deref_move Memory file: uninitialized_copy_n at 136 uninitialized_move_n at 288	22 has same 3 functions modified in file xmemory	22	3
10	e0b8a11					
11	a4ae6c3					
12	ac971a3			changes start from line 5200+, all other commits are above it. No common function		
13	83be0b7					
14	79137b6			63 has changes in destructor.		
15	e3e65be		_ClassRanges at 4126	tangled with 32 for _ClassRanges	32	1
16	b932cf8		_CharacterClassEscape at 4051 _ClassEscape2 at 4075 _ClassAtom at 4093	tangled with 33 for _ClassEscape2	33	1
17	89ca073					
18	b85dd2c					
19	eaf7355		_Find_first_of_pos_vectorized _Traits_find_first_of _Traits_find_last_of			

Figure 3.5: An excerpt of a table showing information about function-level tangling

Commit	Tangling_degree
1	0
2	0
3	1
4	2
5	0
6	1
7	0
8	0
9	0
10	1
11	1
12	0
13	0
14	1
15	2
16	2
17	0
18	1
19	0
20	0
21	0
22	1

Eclipse_file_tangling

Figure 3.6: An excerpt of a table showing Eclipse commits and their file-level tangling degree

3.3.6 Plotting of tangling degrees

After generating all 4 heat maps(file-level and function-level tangling for both repositories), I created one table for each heat map. An excerpt of one such table for Eclipse commits showing file-level tangling degree is shown in the Figure 3.6. The table consisted of 2 columns: Commit and Tangling_degree. For every commit in a heat map, I counted the number of non-zero entries in its corresponding row and noted this count as a tangling degree for that commit. I then developed a Python script to read these tables and plot a bar graph to represent the tangling degree of commits. The script is available in my GitHub repository [26].

4 Results

In this chapter, I will present the findings of this study in relation to the research questions outlined earlier.

4.1 Analysis of commits

4.1.1 Overview of unified diff files

For the Eclipse repository, the unified diff file was approximately 14K LOC across 287 files. After filtering configuration and documentation-related files, there were 150 Java files with approximately 12K LOC to be analysed. The 137 excluded files contributed to only approximately 2K LOC in the unified diff file. For the STL repository, the unified diff file consisted of 263 files with approximately 19K LOC. After filtering, there were 90 files with approximately 11K LOC to be analysed. The 173 excluded files contributed to approximately 8K LOC in the unified diff file.

4.1.2 Filtering of commits

After filtering out commits that were not related to code changes in the Eclipse repository, I was left with 37 code commits, while I excluded 47 out of 84 commits. A similar filtering on the STL repository resulted in 71 out of 81 commits, while I excluded 10 commits. An overview of the number of included and excluded commits, types of excluded commits is shown in Figure 4.1

4.1.3 Examples of commit groups created during the first pass

In the Eclipse repository, there were 8 commits associated with the migration of tests from JUnit4 to JUnit5. 6 of these commits contained the keyword "JUnit5" in their commit messages. Some also included the issue ID #903. I observed that migration was performed one module after another, thus resulting in 8 commits by different authors. Some authors addressed the migration of tests for a module in two commits: Simplify and Migrate. Simplify commit removed the obsolete content, and Migrate commit addressed the actual migration by replacing JUnit annotations.

There was another scenario in the Eclipse repository, where I grouped 3 commits into one commit group. The first commit fixed an issue, and the second commit reverted that first commit. The third commit was the proper fix for the issue. These commits were easily identifiable based on the commit messages: "Fix RSA encryption bit size to 2048", "Revert "Fix RSA encryption bit size to 2048"" and "[SSH] Increase bit-size

	Eclipse repository	STL repository
Total number of commits	84	81
Number of commits excluded	47	10
Number of commits included	**37**	**71**
Excluded commit types:		
Version bumps	22	0
Configuration changes	14	1
Documentation changes	9	3
Image file changes	2	0
Test changes	0	6
Total excluded commits	**47**	**10**

Figure 4.1: Table showing the number of included and excluded commits

of generated RSA and DSA keys". Additionally, I confirmed that the same file was modified in these 3 commits.

4.1.4 Examples of commit groups created during the second pass

During the second pass, while looking into code diffs and other information sources, I identified some commits that were related based on similar underlying scenarios. For example, I grouped 3 commits in the Eclipse repository: b0bb6df, 6d514be and 9ca19a0 into a commit group. While one commit focused on performance improvement by parallel file creation, the other commit addressed performance improvements in file deletion. The commit description from b0bb6df mentioned that this scenario of parallel file creation appears when users perform a clean build of a project, during which all old build files are deleted, and new files are created in parallel. Similarly, the commit description for commit 6d514be referred to an issue, which described the same scenario of using the clean project option, where file deletion performance needed improvement. The third commit, 9ca19a0, fixed the regression caused by the second commit. I grouped these commits into one commit group, based on the scenario they addressed.

The second pass was also helpful in correcting mistakes due to oversights during the first pass. In one instance, I identified three commits 283f954, b6572c8 and ccbfc48 in the Eclipse repository, where it was not immediately clear from the commit messages that they were related. However, during the second pass, I noticed that all three commits referenced the same issue ID in the descriptions, which I had overlooked in the first pass.

4.2 RQ1: Multiple commits for the same feature

4.2.1 Eclipse repository

In the Eclipse repository, I identified 8 commit groups during the commit analysis. Of the 37 included commits, 29 were part of these commit groups. An overview of the commit groups and the number of related commits is shown in the Figure 4.2. 1 commit group consisted of 8 commits, which contributed to JUnit upgrade changes. Another commit group consisted of 5 commits that were syntactically related to OSGi properties code changes. 4 commit groups consisted of 3 commits each, while the remaining 2 commit groups consisted of 2 commits each.

Infrastructure-related commit groups I categorised commit group 1 (JUnit changes) as infrastructure-related. According to the commit messages and linked issue description, the project previously relied on JUnit3, and the team planned to upgrade to JUnit4 and JUnit5. Since there were many test cases, they decided to perform the upgrade in increments to reduce the risk of failures. Code modifications were made to test files to adapt to the JUnit5 framework. For this reason, I categorised this commit group as infrastructure-related.

Feature-related commit groups I categorised commit groups 2, 3, 4 and 7 as feature-related.

Commit group 2 consisted of 3 commits: the first commit implemented a feature to increase the RSA encryption key size, the second commit reverted the first commit, and the third commit was again an implementation of the same functionality. The first and the third commits were authored by the same developer.

Commit group 3 also consisted of 3 commits. The first commit implemented a feature to enable all breakpoints in the IDE. The second commit was a bug fix in disabling all breakpoints, which likely surfaced after implementing the enable all breakpoints feature. The third commit addressed the code review comments for the second commit. All 3 commits here were again authored by the same person.

Commit group 7 consisted of 2 commits, which were committed by the same person. The first commit introduced a feature to display elapsed time in a process in the format HH:MM:SS.mmm. The second commit modified the format to HH:MM:SS by removing the milliseconds, stating that the process was updating the time once a second anyway. This was likely due to unclear specifications at the requirement level.

Scenario related commit groups I categorised commit group 5 as scenario-related. It consisted of 3 commits.

While the first commit focused on performance improvement by parallel file creation, the second commit addressed performance improvements in the deletion of files. The first commit description noted that this scenario of parallel file creation appears when users perform a clean build of a project, where all old build files are deleted, and new files are generated in parallel. The second commit description referred to an issue that described a similar scenario of cleaning a project. Here, the performance of file deletion

Sl.No.	Commit group description	No. of commits	Commit group type	Comments
1	Junit changes	8	Infrastructure related	
2	RSA key size change	3	Feature related	1 change, 1 revert , 1 final change
3	Breakpoint enable/disable	3	Feature related	1 change for enable, 1 for disable (**handle related scenario**), 1 code review
4	Console Tests do not wait for all tests	3	Feature related	
5	Delete and create files in parallel during clean build	3	Scenario related	3rd commit fixed the **regression** caused by 2nd commit
6	Add/Remove synchronize keyword	2	Syntactically related	
7	Show elapsed time in console	2	Feature related	Error in specification level fixed later in commit 2
8	OSGi property related changes	5	Syntactically related	

Figure 4.2: Table showing commit groups for the Eclipse repository

needed improvement. The third commit fixed a regression introduced by the second commit. I grouped these commits into a commit group based on the scenario they addressed. The first and the second commits were committed by the same developer.

Syntactically related commit groups Commit groups 6 and 8 are categorised as syntactically related.

Commit group 6 consisted of 2 commits. In the first commit, synchronized access to a function was relaxed by removing the synchronized keyword from the function and restricting it to a block within the function. In the second commit, synchronized access was added to a different function by adding the synchronized keyword. These two changes are syntactically similar because both modify synchronization constructs. Additionally, both commits were authored by the same author on the same day. The author mentioned in the second commit message that the missing synchronized access to a function was found during inspection. So, I grouped these commits as syntactically related.

Commit group 8 consisted of 5 commits, which involved syntactically similar modifications. The commits addressed code changes, where different OSGi properties were defined and later used in the code. Open Service Gateway Initiative (OSGi) allows developers to define and consume services using annotations without manually managing their life cycle. All 5 commits were authored by the same developer. An example of this type of change is: A constant SERVICE_CONTEXT_KEY and an annotation to represent the service property service.context.key was defined in one commit. In the next related commit, this constant SERVICE_CONTEXT_KEY was used instead of the hard-coded string service.context.key, while using the service.

4.2.2 STL repository

In the STL repository, I identified 9 commit groups during the commit analysis. Of the 71 included commits, only 25 were part of these commit groups. An overview of commit

Sl. No.	Commit group description	No. of commits	Commit Ids	Commit group type	Comments
1	Clang with alternate name	2	2cef491 ca2b2c6	Infrastructure related	Revert commit
2	Vectorize string find function	2	126f4eb eaf7355	Feature related	Performance improvement of a feature
3	Copy elision	2	649a4f2 "0372e78"	Feature related	2nd commit is the follow-up to the 1st commit
4	Regex character classes	2	b932cf8 63fe8a2	Feature related	Partial fixes for the same bug
5	Fix ranges for ADL only chang	2	53432eb d4b57c7	Feature related	Addressing the regression from previous release commit
6	Simplify string visualization	2	56bd1fe ae9d115	Feature related	
7	STL Hardening	4	2314e1a f953153 dfdccda 0d8f517	Feature related	
8	MSVC version update	4	2378c81 "0408152" d43d49a 18cdebd	Infrastructure related	
9	Toolset update VS previews	5	b85dd2c "0190042" 2446dba a4e32ac 3804078	Infrastructure related	

Figure 4.3: Table showing commit groups for the STL repository

groups and the number of related commits within each commit group is presented in the Figure 4.3. 1 commit group consisted of 5 commits, 2 commit groups with 4 commits each, and the remaining 6 commit groups comprised 2 commits each.

Feature-related commit groups I categorised commit groups 2, 3, 4, 5, 6 and 7 as feature-related.

Commit group 2 consisted of 2 commits. Both commits addressed the performance improvement of the find function in basic_string and were authored by the same developer. While the first commit improved a general use-case of the find function in strings, the second commit improved a specific use-case of finding the first and last occurrence of a large substring (needle) in a large string (haystack).

Commit group 3 also consisted of 2 commits. In the pull request for the second commit, the author mentioned that this commit was a follow-up to the first commit.

Commit group 4 consisted of 2 commits, which provided partial fixes to the same bug by the same author. Both pull requests mentioned the same bug ID that the commits addressed.

Commit group 5 comprised 2 commits. Both commits addressed a regression introduced in the previous release. The pull request for the second commit mentioned that it was a follow-up to the first commit.

Commit group 6 also consisted of 2 commits, which were authored by the same developer on the same day. They addressed the feature of simplifying the string visualisation in the debugger window of the IDE.

Commit group 7 consisted of 4 commits that contributed to the requirement STL

Hardening. In the pull request of the third commit, the author referenced the earlier two commits that contributed to this feature. He also outlined the next step in the plan, which was the 4th commit. As a result, I grouped these 4 commits into a feature-related commit group. These were large commits that modified several files (8, 37, 53 and 21 files, respectively).

Infrastructure-related commit groups I categorised commit groups 1, 8 and 9 as infrastructure-related.

Commit group 1 consisted of 2 commits, where the second commit was a revert of the first commit. The commits addressed changes related to the Clang compiler used in the infrastructure.

Commit group 8 comprised 4 commits, where the macro value of MSVC_STL_UPDATE was updated every month. This allows users to know what version of STL is being used in their development environment. This does not affect product functionality.

Commit group 9 consisted of 5 commits. In the repository, the build compiler is upgraded to new Visual Studio releases. As a result of upgrades, certain compiler workarounds are removed from the source code. One commit in this commit group modified 89 files. Since these changes were related to compiler upgrades and their workarounds, I categorised this commit group as infrastructure-related.

4.2.3 Information sources used to identify commit groups

Eclipse repository

In the Eclipse repository, commit messages and descriptions often contained the same keywords that helped in identifying related commits. An additional helpful information source was linked issues in commit message descriptions. Authors often included links to the issue IDs that they addressed in commits, as part of commit message descriptions.

Apart from this, I looked at other information sources, such as code diffs, authors, commit dates, and modified files in commits, to further identify related commits and form commit groups.

There was one commit with the message "Code review changes". It was not possible to identify the related commits using only the commit message in this case. However, this commit occurred on the same day and immediately after the commit "Disable all breakpoints bugfix + code changes". Also, both commits were authored by the same developer. I also looked at code diffs and concluded that these two commits were related.

STL repository

In the STL repository, commit messages alone were not enough to identify related commits. This is because commit messages contained only a one-line description and a link to a pull request. Further looking at pull requests, a lot of information was obtained. It included an overview, links to features and bugs being addressed, discussions and comments from other people.

All pull requests also had labels, which carried very useful information. Some of the labels used in this repository included enhancement, bug, LWG (Library working group), infrastructure, test, performance and so on. Some pull requests even indicated the component name involved, using labels such as chrono, ranges, regex and so on. These labels were very helpful to quickly understand the nature of commits.

4.3 RQ2: Tangling degrees

As mentioned in the Methodology section 3.3.1, I presented tangling degrees using heat maps and bar graphs. Heat maps show the number of common modified files and functions between commit pairs. Bar graphs show the number of commits tangled with a commit, which share at least one modified file or function.

4.3.1 Eclipse repository

The heat map showing file-level tangling for the Eclipse repository is presented in Figure 4.4. The heat map showing function-level tangling is presented in Figure 4.5. The corresponding bar graphs are presented in Figures 4.8 and 4.9, respectively.

File-level tangling The heat map showing file-level tangling (Figure 4.4) contains 1369 cells (37*37 commits), with values ranging from 0 to 8. Of 1369, only 28 cells have non-zero values, indicating tangled files. The heat map exhibits diagonal symmetry, where the upper and lower triangles are mirrored. This is due to the symmetric nature of data: for any commits A and B, tangling-degree(A, B) is the same as tangling-degree(B, A). So, 28 cells must be treated as 14 cells to avoid duplicate commit pairs.

The majority of these 14 cells show a value of 1. That means the majority of the tangled commit pairs share only one modified file. 2 commit pairs show high cell values of 7 and 8. All these 4 commits belong to the same commit group, related to the JUnit upgrade performed by the same author. This author performed the upgrade in two steps: Simplify and Migrate. Simplify commit removed the obsolete content, and Migrate commit addressed the actual migration by replacing the JUnit annotations.

Of 14 tangled commit pairs, only one commit pair consists of commits (commits 16 and 33) that do not belong to the same commit group. Commit 16 addressed JUnit upgrade changes, and commit 33 removed functions that were not part of the old version of JDK 10, while the Eclipse projects used the latest JDK 17. The remaining 13 tangled commit pairs consist of commits that belong to the same commit groups (Figure 4.6).

The bar graph (Figure 4.8) shows that 20 out of the 37 commits are tangled with at least one other commit, which share at least one modified file. The range of tangled commits is from 0 to 2, as shown along the y-axis. Of the 20 tangled commits, 12 are tangled with only one commit. The remaining 8 are tangled with 2 commits.

Function-level tangling The heat map for function-level tangling (Figure 4.5) also contains 1369 cells (37*37 commits) with cell values ranging from 0 to 25. Of 1369 cells, only 16 have non-zero values. After considering the mirroring symmetry of the

heat map, it reduces to 8 cells with non-zero values. Of 8 cells, 6 cells show a value of 1, indicating that commit pairs share only one modified function. There are 2 commit pairs with large values, 9 and 25. These commit pairs correspond to the same pairs with high values in the file-level tangling heat map, i.e., the JUnit upgrade changes.

All 8 tangled commit pairs consist of commits that belong to the same identified commit groups (Figure 4.7).

The bar graph (Figure 4.9) shows that 13 out of the 37 commits are tangled with at least one other commit, sharing at least one modified function. Of the 13 commits, only 3 have a tangling degree of 2, while the rest have a tangling degree of 1. The commits with tangling degree 2 are 31, 32 and 34. These commits are related to RSA Key size changes. A tangling degree of 2 in the bar graph means that there are 2 other commits with which a commit has function tangling.

4.3.2 STL repository

The heat map showing file-level tangling for the Eclipse repository is presented in Figure 4.10. The heat map showing function-level tangling is presented in Figure 4.11. The corresponding bar graphs are shown in Figures 4.14 and 4.15, respectively.

File-level tangling The heat map showing file-level tangling (Figure 4.10) contains 4624 cells (68*68 commits). As I had excluded 3 large commits (commits 49, 55, 60) to make the manual process of finding function-tangling feasible, I excluded these commits from file-level tangling as well to maintain consistency between the results. So, the heat maps include 68 commits instead of 71. Figure 4.10 shows that there are many non-zero cell values. This indicates that several commit pairs share at least one modified file. The cell values range from 0 to 6. Many cells show a value of 1, while others show values of 2, 3, 4 and 6.

Only one commit pair, commits 27 and 63, shows the highest cell value of 6. These commits do not belong to the same commit group. Commit 27 addresses code changes related to marking exception types as [[nodiscard]]. If a return value of this [[nodiscard]] type is ignored, it warns programmers to handle return values. Commit 63, which is a part of commit group 7 (Figure 4.3), addresses code changes related to destructor tombstones. It prevents access to already destroyed objects. This tangling disappears at the function level, since both address different components. Commit 27 addresses all error and exception handlers, while commit 63 addresses all destructor functions.

There are a total of 454 non-zero cells, which reduce to 227 cells after considering mirroring symmetry. Of 227 tangled commit pairs, only 12 commit pairs consist of commits that belong to the same commit groups (Figure 4.12). The remaining commit pairs consist of commits that do not belong to the same commit groups.

The bar graph (Figure 4.14) shows that 62 of the 68 commits are tangled with at least one other commit, sharing at least one modified file. File-level tangling degrees, along the y-axis, range from 0 to 35. Commit 70 is tangled with 35 other commits. This commit message is titled "Various cleanups" and consists of 22 modified files. It addresses several minor cleanup code changes, which are not related to any particular

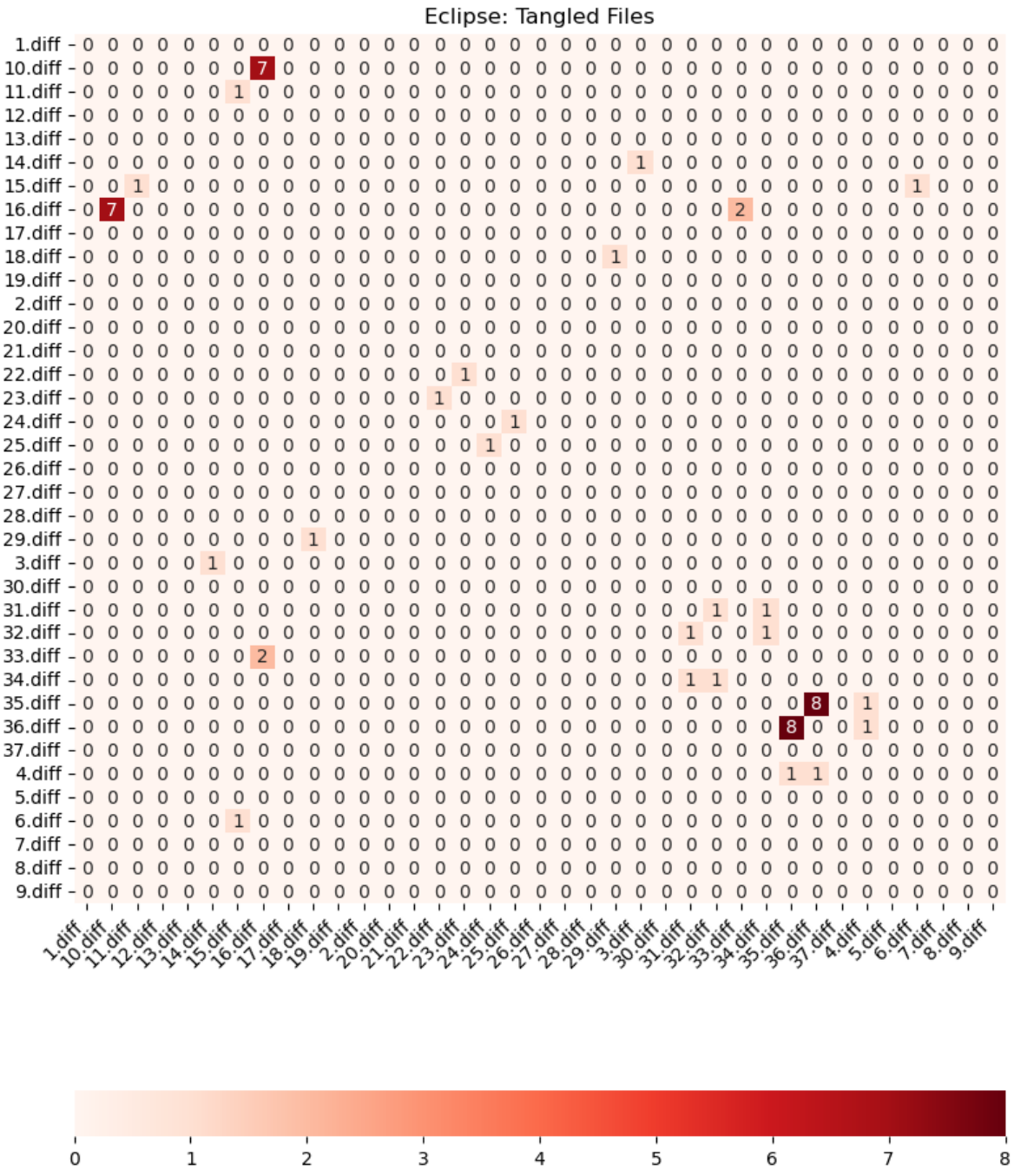

Figure 4.4: Heat map showing file-level tangling for the Eclipse repository

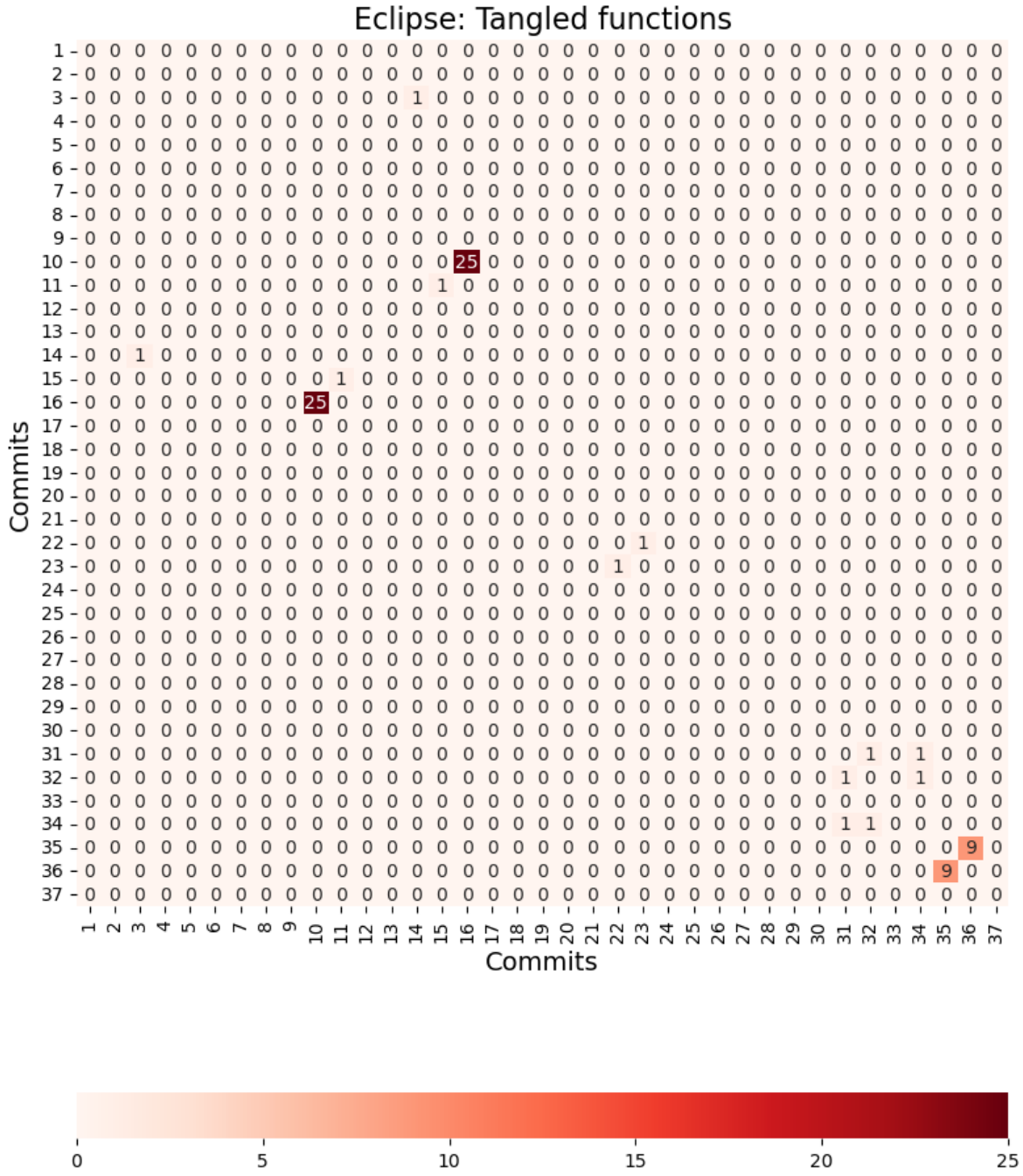

Figure 4.5: Heat map showing function-level tangling for the Eclipse repository

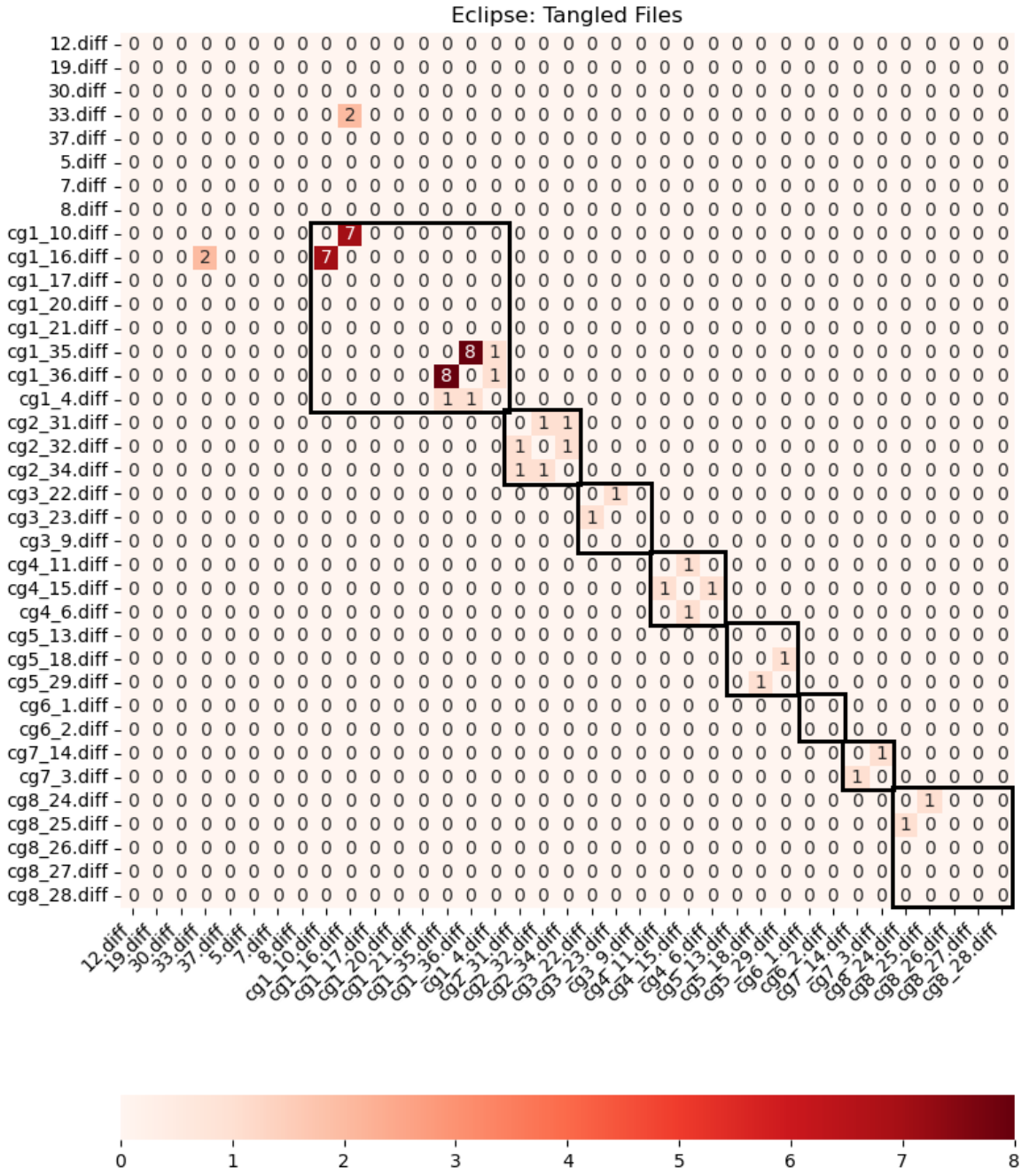

Figure 4.6: Heat map showing file-level tangling within commit groups for the Eclipse repository

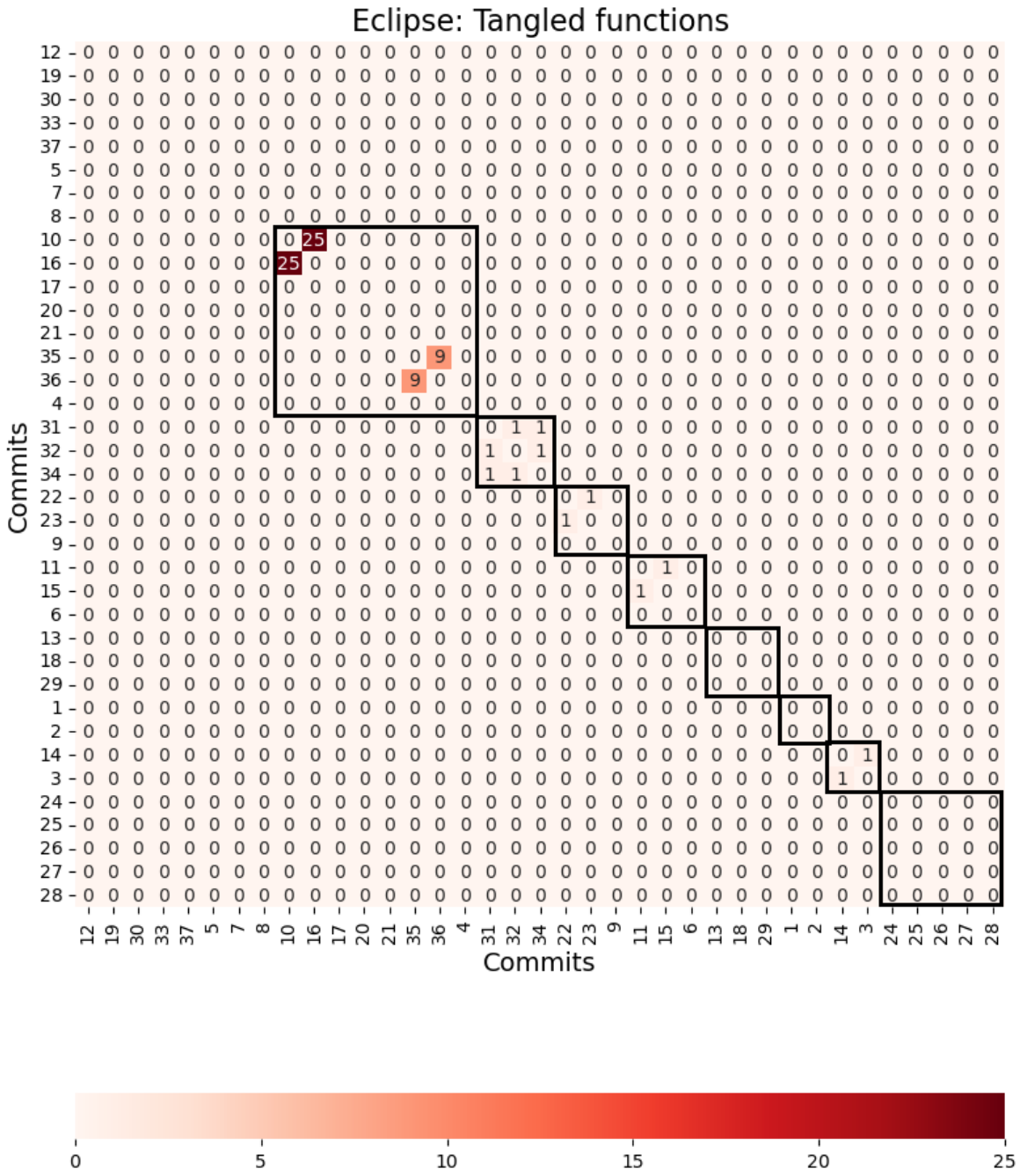

Figure 4.7: Heat map showing function-level tangling within commit groups for the Eclipse repository

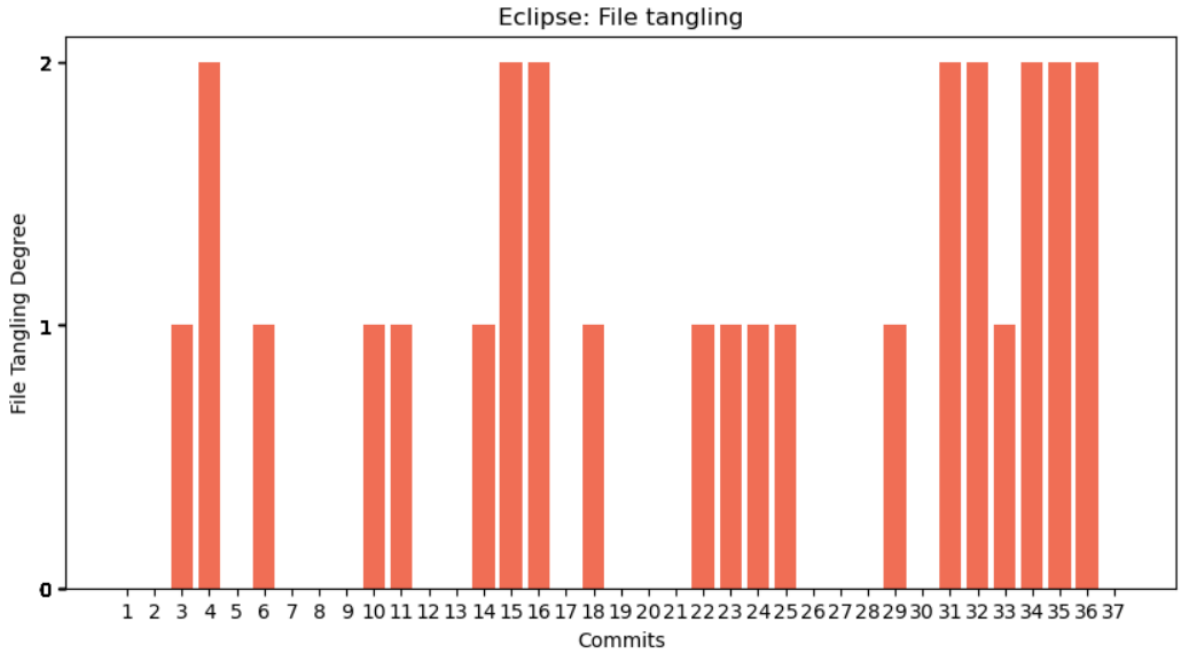

Figure 4.8: Bar graph showing file-level tangling for the Eclipse repository

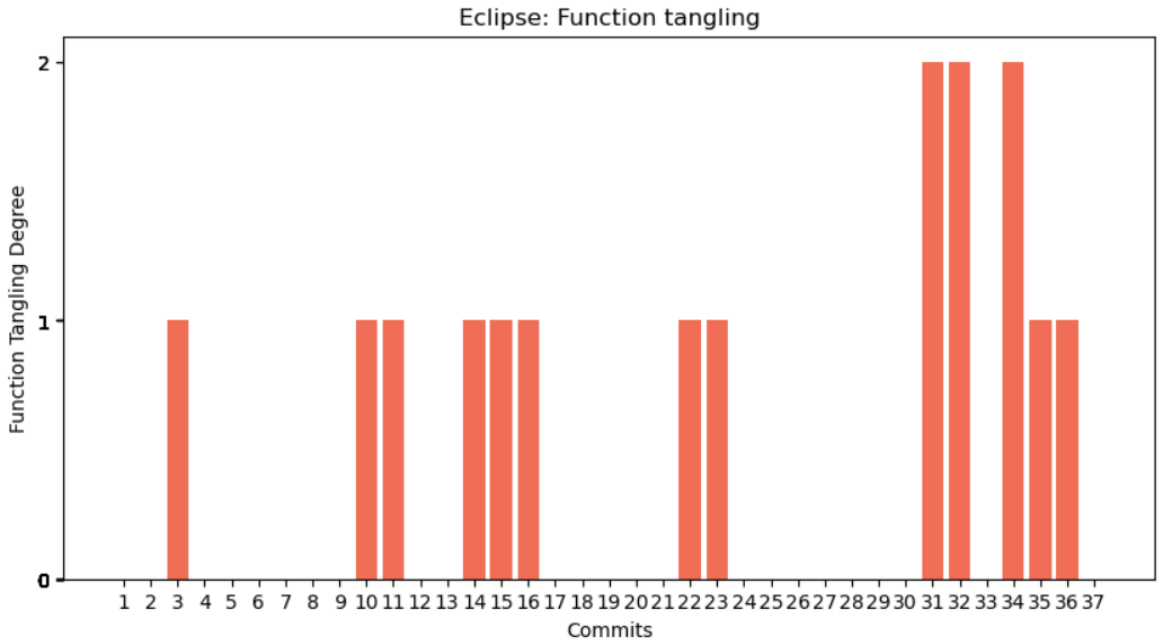

Figure 4.9: Bar graph showing function-level tangling for the Eclipse repository

feature. All minor code changes were accumulated and committed just before the release. This commit is not part of any identified commit group.

Function-level tangling The heat map for function-level tangling (Figure 4.11) contains 4624 cells for 68 commits. The cell values range from 0 to 3. Only 18 cells have non-zero values, which reduces to 9 cells after considering mirroring symmetry. Only 9 non-zero cells indicate lesser function tangling between commit pairs compared to file tangling. Of the 9 non-zero cells, only 1 cell, with commits 9 and 22, has the highest value of 3, showing that commits have 3 common modified functions. These commits belong to the same feature-related commit group 3 in Figure 4.3

Of 9 tangled commit pairs, only 2 commit pairs consist of commits that belong to the same commit groups , while the remaining 7 commit pairs consist of commits that do not belong to the same commit groups (Figure 4.13).

The bar graph (Figure 4.15) shows that only 16 out of the 68 commits are tangled with at least one other commit, sharing at least one modified function. The function tangling degree along the y-axis ranges from 0 to 2. Only 2 commits (commits 32 and 58) have a tangling degree of 2, while the remaining 14 have a tangling degree of 1.

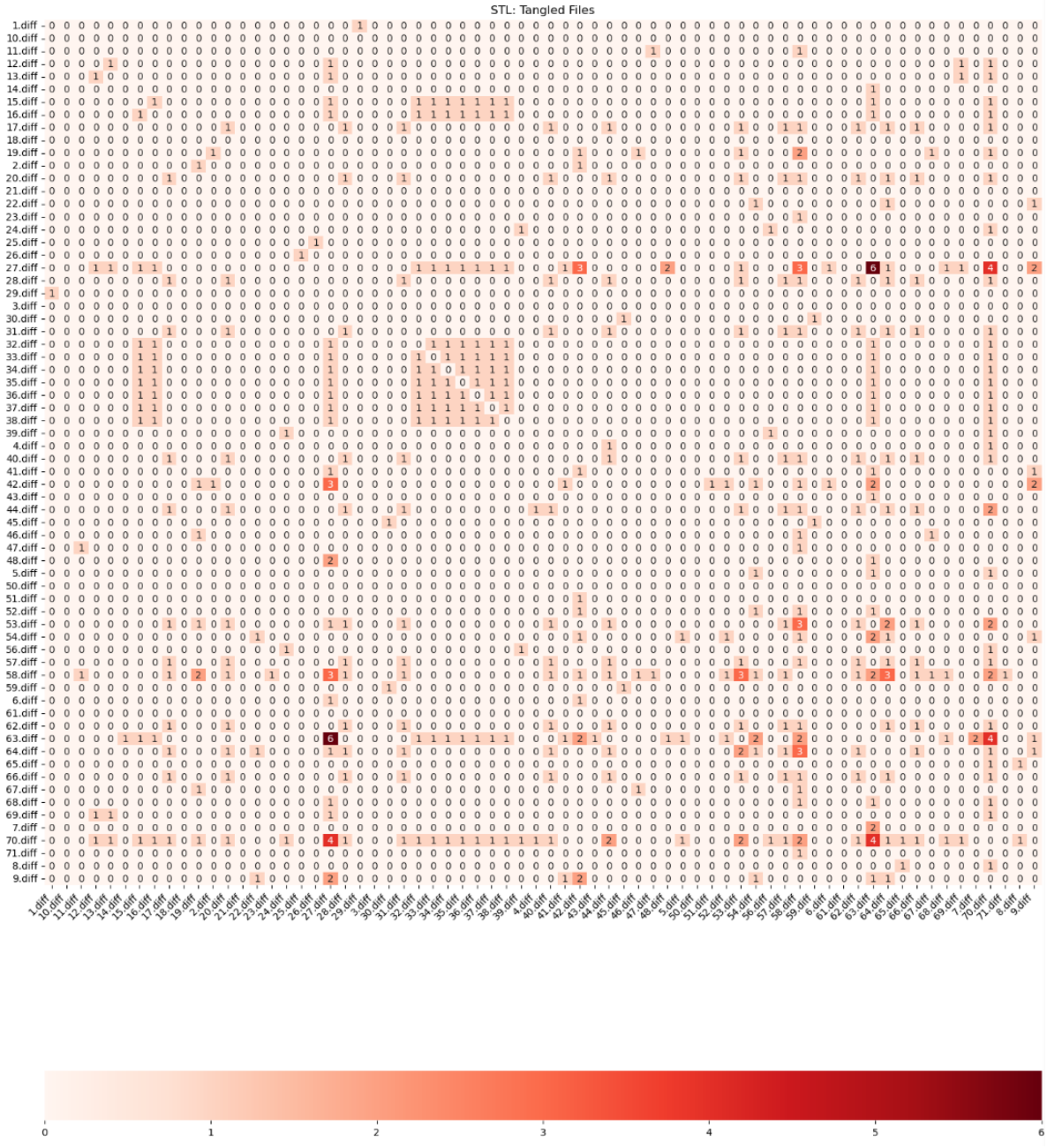

Figure 4.10: Heat map showing file-level tangling for the STL repository

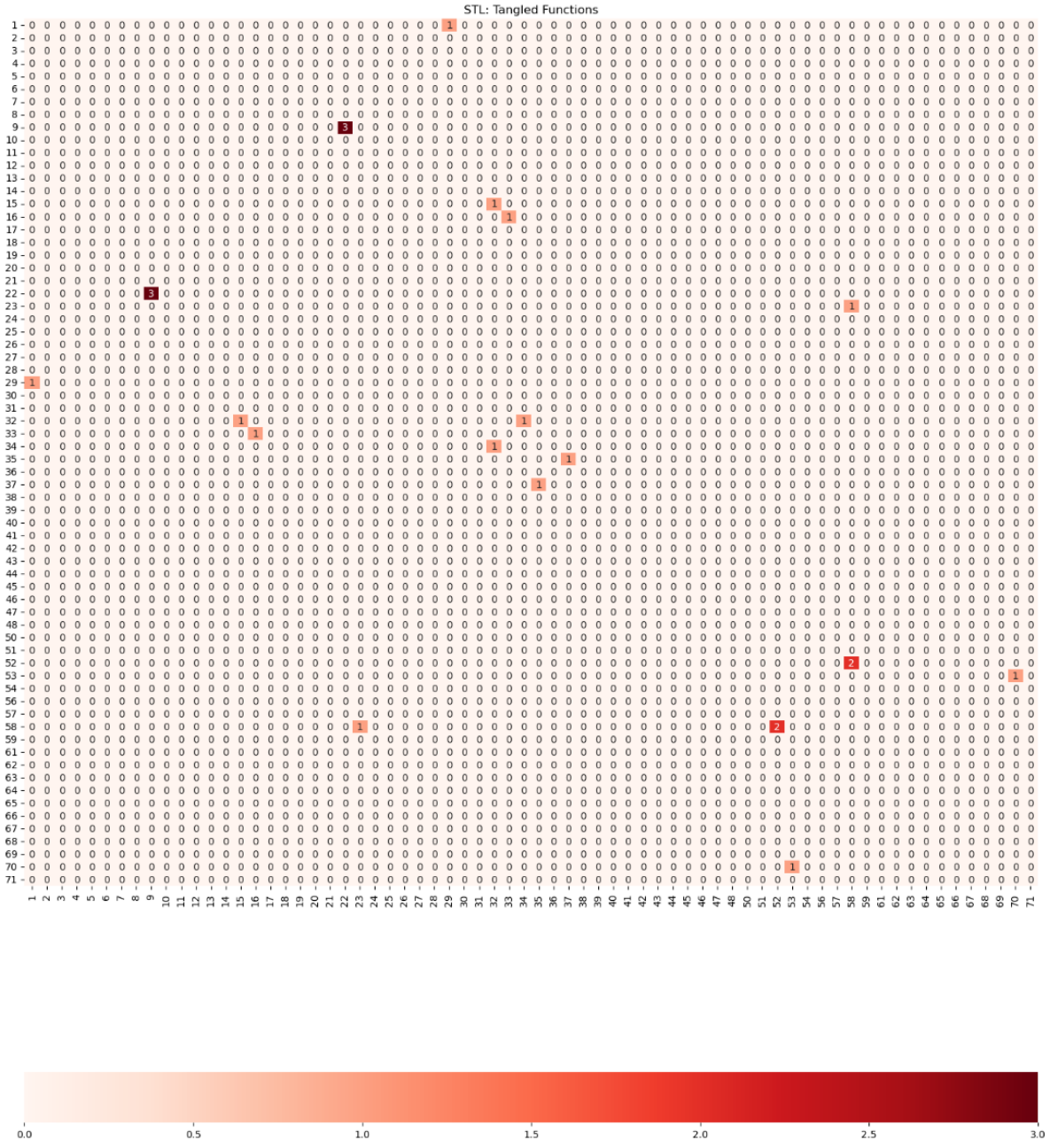

Figure 4.11: Heat map showing function-level tangling for the STL repository

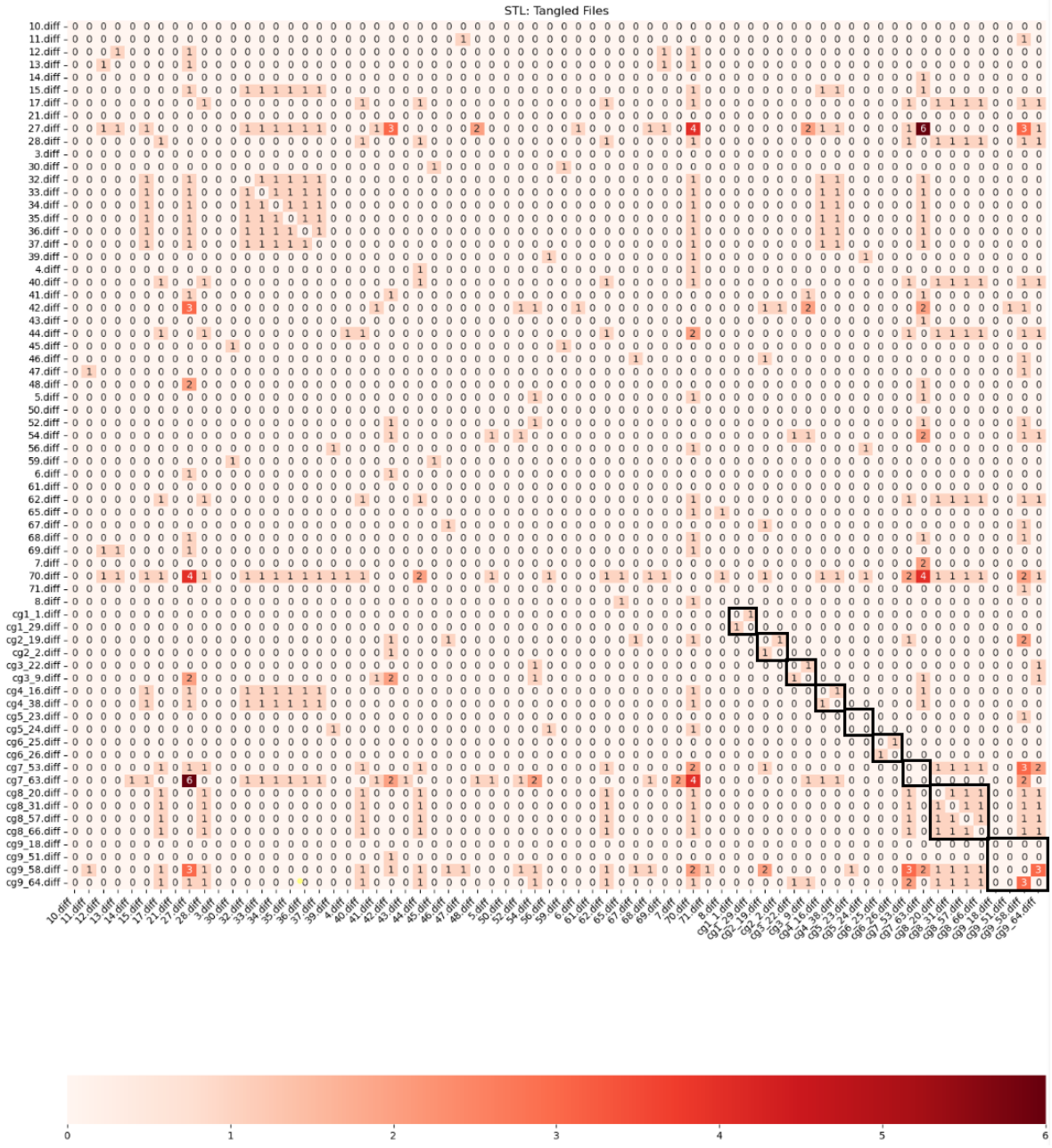

Figure 4.12: Heat map showing file-level tangling within commit groups for the STL repository

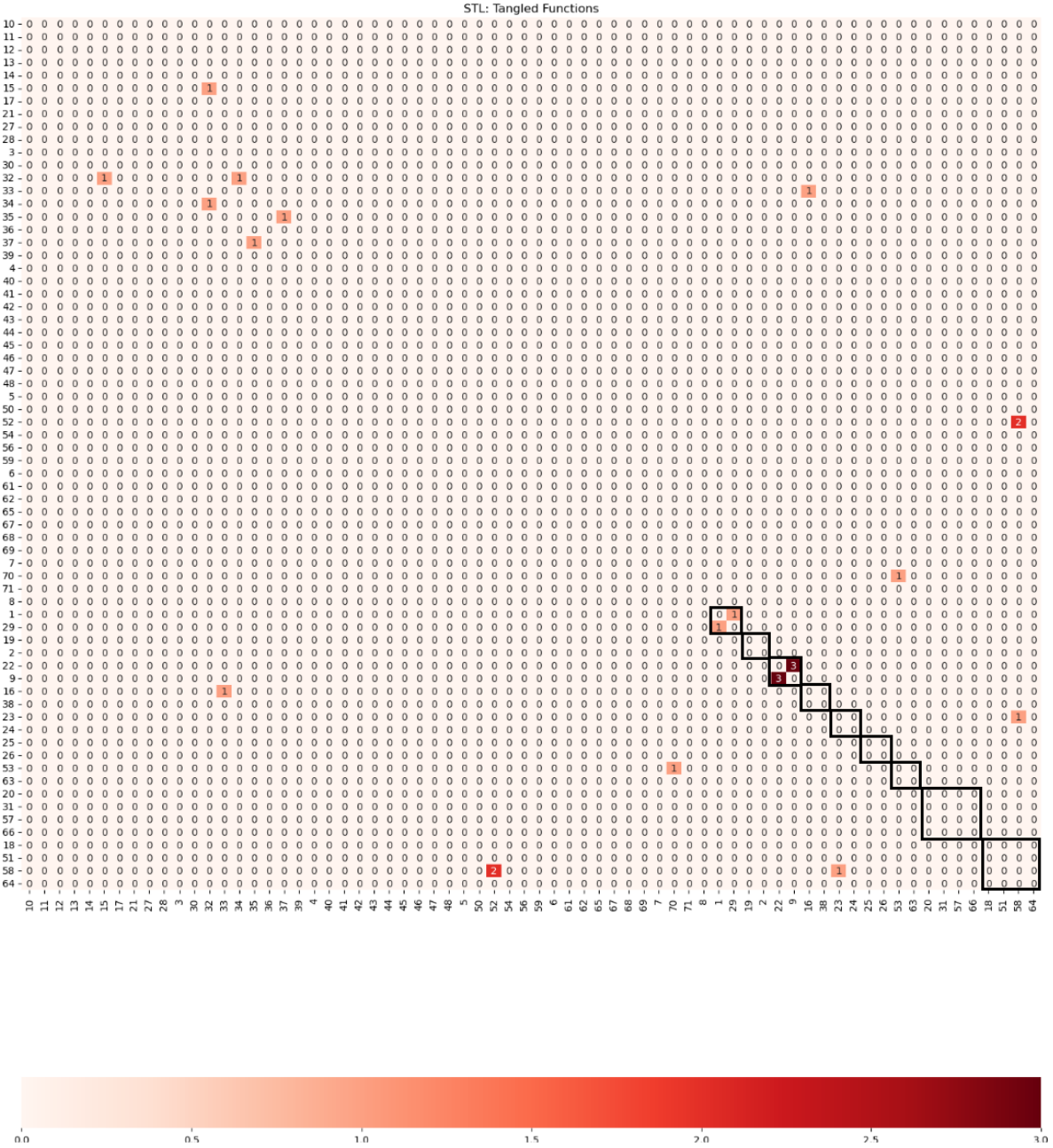

Figure 4.13: Heat map showing function-level tangling within commit groups for the STL repository

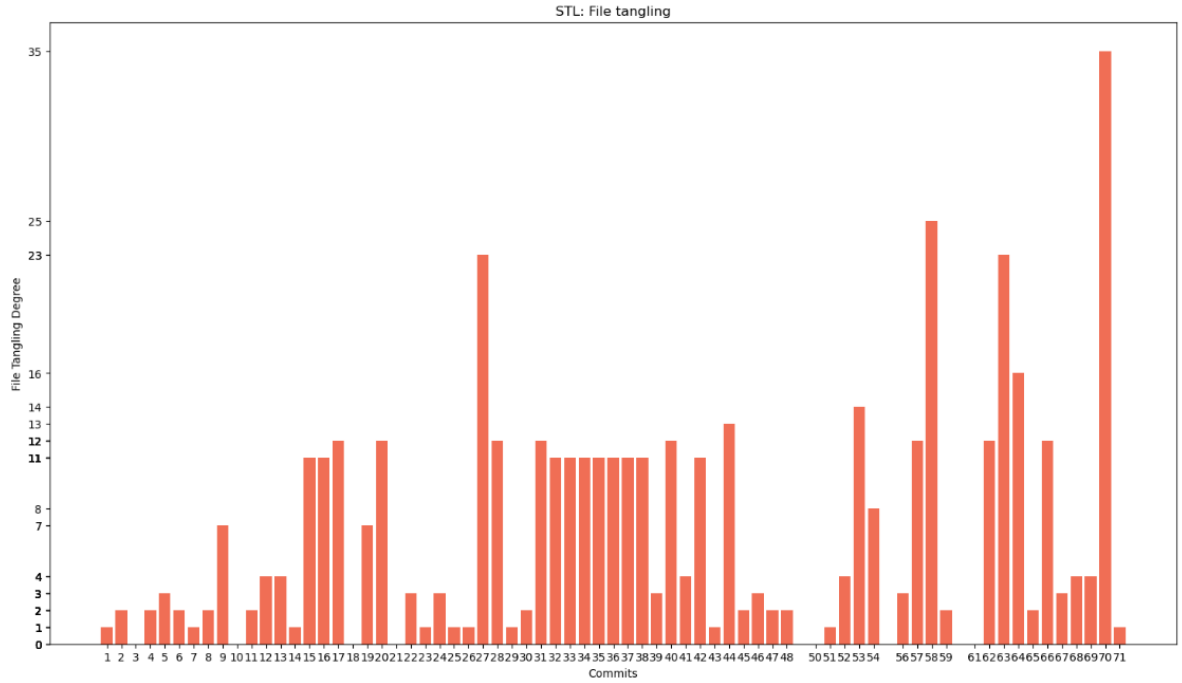

Figure 4.14: Bar graph showing file-level tangling for the STL repository

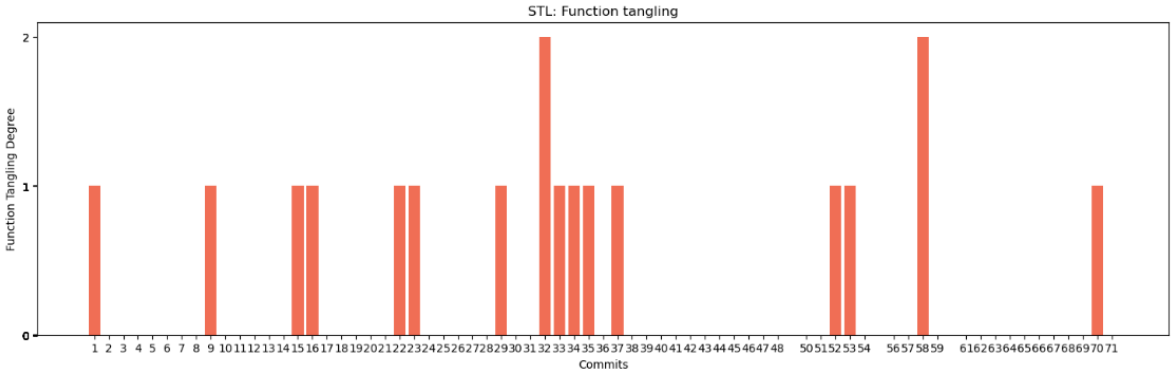

Figure 4.15: Bar graph showing function-level tangling for the STL repository

5 Discussion

5.1 Interpretation of results

5.1.1 Identified commit groups

When comparing the number of commit groups identified in Eclipse and STL repositories, it is important to consider the number of commits analysed. After filtering the commits of one release, I analysed 37 and 71 commits for the Eclipse and STL repositories, respectively. For the Eclipse repository, 37 commits resulted in 8 commit groups. In contrast, for the STL repository, 71 commits led to only 9 commit groups.

Although the number of analysed commits for the STL repository was approximately twice that of the Eclipse repository, the number of commit groups was approximately the same for both repositories (8 and 9 for Eclipse and STL, respectively). It can be inferred that a multi-commit culture is stronger in the Eclipse repository than in the STL repository.

In the STL repository, commit messages consistently referenced links to pull requests. Each pull request consisted of multiple commits related to partial feature commits, review comments, code cleanup, fixing compiler warnings, and code fixes for failed tests. Only after the review was complete, pull requests were merged to the main branch as a single commit. The checklist for merging pull requests explicitly specifies that all commits in the pull request should be squashed and merged to the main as one single commit. [22]

In the Eclipse repository, commit messages did not provide any reference to pull requests. Thus, pull requests related to commits were not easily identifiable. However, on manual search for some pull requests, I found that pull requests here, too, consisted of multiple commits addressing partial feature commits, review comments, and version bumps. After the code review was complete, pull requests were merged to the main branch. In the contribution guidelines [4], it is mentioned that pull requests can be merged to the main using either of the options: Rebase and Merge, which retains all commits of the pull request separately in the main, or Squash and Merge, where all commits in the pull request get squashed to a single commit in main. During manual search for some pull requests, I came across a few instances where pull requests were not squashed into one commit, thus retaining individual commits from pull requests in the main branch. This explained a particular commit in the main with the commit message "Code review changes".

This difference in merge options used in both repositories could be one reason explaining the number of commit groups seen in the Eclipse repository.

5.1.2 Filtered commits

In the Eclipse repository, there were 84 commits in the release. Of the 84 commits, 47 were excluded (more than half). Many of these excluded commits were related to version updates, configuration changes and documentation. However, in the STL repository, only 10 commits were excluded from 81 commits in the release.

The primary reason for excluding a large number of commits in the Eclipse repository is due to several version bump commits. These commits only addressed modifying the version numbers in manifest.MF, pom.xml and other configuration files. Manifest.MF files contain metadata about JAR files that are generated during compilation. They contain information such as version number, files, main entry class and so on. A pom.xml file in Maven projects contains information related to building a project, its configuration and dependencies. The Eclipse repository has multiple manifest and pom files. The reason for such a large number of version commit bumps ending up in the main branch is again due to the contribution guidelines [4]. It specified that if contributors made version bumps in manifest or pom files, it was advised to provide two separate commits in the same pull request: one for functional changes and the other for the version bump. This would help maintainers cherry-pick commits from pull requests during the merge.

This highlights that maintainers wanted to easily differentiate functional code changes from other infrastructure-related code changes. This further supports the necessity of categorising commit groups as infrastructure-related in my studies. Infrastructure-related commit groups indicate that there is no change in product functionality. When developers want to update their forks from the original repositories, they can prioritise the updates based on the value they add. A mere infrastructure change might not be as important as a bug fix, security update or a new feature. Commits especially related to version bumps might not even be necessary for forks if they do not maintain a release system.

5.1.3 Contributor practices

Developers in the STL repository follow strict contribution guidelines. Every commit to the main branch followed a standard format for commit messages: a one-line description followed by a reference to a pull request. The pull requests further revealed details about commits. Every pull request had at least one label, which indicated the nature of the change. A label "good first issue" was used for some issues, which indicates that the issue is intended for a new contributor to get started with the simplest possible change. Code changes for fixing such issues would be simple, low-risk and minimal to start with. This can be a very good indicator and motivator for new contributors. The conversation section in pull requests shows active coordination and discussion among developers.

In the Eclipse repository, the guidelines recommended to include issue number and title along with a short description in the commit message. However, not all commit messages followed the recommendations. One reason for this could be again due to the rebase and merge option used for some pull requests, which takes all commits with non-standard commit messages from pull requests as it is. Also, developers did not

reference pull requests in commit messages. Labels were also not seen for pull requests in the Eclipse repository. However, there is an active conversation between contributors and maintainers in pull requests.

Interactions between contributors and maintainers in pull requests provide a lot of important contextual information related to the issue or feature being addressed. Maintainers provide more information about the issue, such as the scenario where it occurs, corner cases to handle, alternatives for solutions and so on. Labels also provide a quick and good overview of the type of issue or change. These are great information sources to understand more about commits that address issues or changes. Developers who want to update their forks can look at these information sources to gain a better understanding of issues. It also enables them to come up with a better approach to update forks without affecting the customisation code.

5.1.4 Tangling degrees

Bar graphs for the function tangling degree coincidentally show a maximum value of 2 for both repositories. This could be due to the exclusion of 3 large commits in the STL repository that contributed to approximately 400 hunks of code changes. Including these large commits might increase the function tangling degree for the STL repository.

A higher tangling degree indicates that the source code is highly tangled, making it more challenging for users to pull commits to their forks without affecting other unrelated features or functionality. It is noteworthy that most of the tangled commit pairs in the Eclipse repository belong to the same commit groups, whereas this is not the case for the STL repository. Tangling between commits within the same commit group is generally less problematic when users update their forks with new features from the original repository. Users interested in a specific feature can pick all commits contributing to that feature without significant concern about tangling between commits.

However, tangling between commit pairs that do not belong to the same commit group can be challenging for users in this update scenario. Untangling and integrating such code changes into forks can be complex and may introduce bugs during the update process. Along with handling such tangled code, users must additionally ensure to retain their fork customisations during updates along with tangled code.

STL repository

In the STL repository, the heat maps indicate that file-level tangling is significantly higher compared to function-level tangling.

STL source code structure One explanation is based on the code structure in STL. Each component has its own header file in the repository under /stl/inc. For example, the component regex has a file /stl/inc/regex. Any code change related to regex is most likely to modify this file. During this study, I found that there were 7 consecutive commits related to regex performed on the same day. Although these commits modified

the regex file, not all commits were related to each other. Thus, it increased file-level tangling without increasing function-level tangling.

Header file with preprocessor statements A header file called yvals_core.h located at /stl/inc is approximately 2K LOC. This header file contains only preprocessor directives such as #ifdef, #undef, #define, #if defined and so on. Since this file contains several preprocessor directives, it is widely used and included in the source code. This file was modified in 13 commits, which definitely resulted in higher file tangling between commit pairs.

Large commits Although I excluded commits 49, 55, 60 that modified 89, 37, 53 files respectively, there were some other commits that modified a large number of files. 2 commits related to the Toolset update modified 21 and 23 files. A commit for Destructor tombstones related to STL Hardening modified 21 files. There was another general commit called "Various cleanups" that modified 22 files. This commit addressed several minor code cleanup changes, which were not related to any particular feature. All minor code changes were accumulated and committed just before the release.

When a commit modifies a large number of files, it is highly likely to be tangled with many commits at the file level. This pattern was visible in the heat maps and bar charts for the STL repository.

5.2 Comparison with existing literature

5.2.1 Commit group categories

Levin et al. [16] classified commits in maintenance projects into 3 categories: corrective: fixing bugs, perfective: system and design improvement and adaptive: adding new features.

Dos Santos et al. [24] also classified commits into five categories: corrective, perfective, features, non-functional and unknown. Here, they adopted the definitions for corrective, perfective and features from the study by Levin et al. Additionally, the non-functional category was associated with documentation and non-functional requirements, and the unknown category addressed files generated by version control systems.

Benett et al., in their study of software maintenance and evolution [3], adopted the following categories for maintenance activities: adaptive: changes in the software environment, perfective: new user requirements, corrective: fixing errors, and preventive: preventing problems in the future.

In my study, the reasoning I used to group related commits can be mapped to these studies as follows: My feature-related category covered corrective, features and perfective categories, infrastructure-related category aligns with the definitions of non-functional [24] and adaptive [3]. I also included two additional categories for scenario-based and syntactically related commits.

Scenario-based category addresses multiple issues/features related to the same scenario. It can be considered as a parent of the feature-related category. When users want

to update forks with new features, the scenario-based category gives them additional information about features they might be interested in. For example, if users want to improve performance during a clean build of a project, they might also be interested in performance improvement for the feature clean project.

The syntactically related category consists of code changes that are similar in nature. These commits do not address a single feature or issue. They might be corrective or adaptive. So, this category does not align with the existing categories in the literature mentioned above. Commits in this category are very good candidates for automatic code generation using LLMs through prompt engineering techniques like one-shot prompting. This helps with faster updates of forks with reduced effort.

5.3 Threats to validity

Scale of the study I performed commit culture analysis for two open-source repositories. For each repository, commits of one release were studied. So, the findings of this study might be specific to these repositories. In future, this study can be extended to more repositories with a large number of commits to get additional meaningful insights that can be generalised to all repositories.

Verification of commit group formation Commit group formation was done entirely based on my understanding of commits and source code. There is no mechanism to verify whether the grouping of related commits was entirely correct. However, I specified all the details related to the grouping of commits explicitly in the Methodology and Results chapters.

JavaParser library I used JavaParser library to find modified functions in the Eclipse repository. I noticed during initial testing that the library returns no function name for import statements at the beginning of files. This behaviour is as expected. However, there might be other scenarios that I have not come across during the study, where the library might not return expected results. This might have an influence on function-level tangling results for the Eclipse repository.

Manual process of function tangling for the STL repository I performed the task of measuring function tangling for the STL repository manually, which relied on repeated inspection of code diffs and making decisions. As a result, there might be possibility of human error. Such errors might influence the accuracy of function tangling results for the STL repository.

5.4 Significance of the study

This study provides insights into commit culture in terms of developer practices. Developers can use these insights to examine their repositories, identify shortcomings in their existing processes, and establish best practices and guidelines. This, in turn, can

simplify maintenance, onboarding of new contributors, knowledge transfer, and feature traceability.

For example, a high number of revert commits may indicate that commits unintentionally broke a feature or caused regression in software. This could motivate stricter guidelines in existing processes, such as adding test cases to new features, running an automated test set to ensure product functionality before commits, and conducting more thorough code reviews. In the Eclipse repository, 3 revert commits were found in the release chosen for this study. One commit was a configuration-related change modifying the pom.xml for the build. One commit reverted the RSA Key size change, which is a part of commit group 2. Another revert was a code change from the previous release that was reverted in this release. In the STL repository, only one revert commit was found for the release. It was related to infrastructure changes involving the Clang compiler. The commit is a part of commit group 1.

It is a common practice to fork open-source repositories and customise according to one's requirements. This analysis is helpful in scenarios where forks need to be updated with the original repository to include bug fixes, features, and security updates, while still maintaining the customisation. Commit groups help in feature traceability in these scenarios, as they represent all commits related to a feature/change. Users can select a commit group related to a feature they are interested in for fork updates. Moreover, commit group categories provide users with an overview of the value they add. For example, infrastructure-related commit groups indicate that there is no change in product functionality. A mere infrastructure change might not be as important as a bug fix, security update or a new feature. Commits especially related to version bumps might not even be necessary for forks if they do not maintain a release system.

A higher tangling degree indicates that the source code is highly tangled, making it more challenging for users to pull commits to their forks without affecting other unrelated features or functionality. Tangling between commits within the same commit group is generally less problematic when users update their forks. Users interested in a specific feature can pick all commits that contribute to the feature without significant concern about tangling between commits. However, tangling between commit pairs that do not belong to the same commit group can be challenging for users. Untangling and integrating such code changes into forks can be complex and may introduce bugs during the update process.

5.4.1 Future work

An extension of the fork update use case applies to software product lines. Some software companies have a baseline software, and they provide customised variants of the baseline software to different clients to suit their specific requirements. I will refer to customised variants of software as extensions. They maintain separate code bases for each extension of the software. When a new release is rolled out for the baseline software, all extensions must be upgraded to the newer release, while still preserving their customisation. Manually performing this upgrade across several extensions can be a time and labour-intensive task, and is prone to errors.

Automating this process would be beneficial for companies. My study of commit

analysis provides insights into how commits can be grouped to represent a feature, bug, or enhancement, what information sources can be used to identify related commits, how tangled commits are, and how challenging it is to update forks. This information can be used to automate the process of upgrading extensions. Another promising direction for automation is to refactor baseline source code using large language models, such that extensions' source code is not highly tangled with baseline source code.

If such an analysis of commit culture is performed for many open-source repositories, the findings could be used to develop a standard template of contribution guidelines for open-source repositories. For example, if the analysis reveals that there are multiple revert commits in a repository, it may indicate that the commits caused some regression, or they did not fix issues correctly. Such issues can be avoided by incorporating certain measures, such as adding more test cases, stricter code reviews, clear documentation, and clear communication of requirements. This can help solve collaborative challenges often seen in open-source development.

6 Conclusion

In this thesis, I investigated commit culture in two well-known open-source repositories: the Eclipse IDE and the Microsoft STL library. For both repositories, I analysed all commits belonging to one release. The aim was to understand how developers structure commits, i.e., whether developers use multiple commits for the same feature or change and how tangled the code changes are between commits at the file and function levels.

The results related to identified commit groups show that developers do use multiple commits to implement a feature, bug fix or enhancement, but the extent of this practice varied between the two repositories. In the Eclipse repository, a stronger multi-commit culture was seen. However, the STL repository showed fewer multi-commit instances despite the larger number of commits analysed. This contrast could be due to different contribution guidelines and practices in both repositories.

The results indicate that tangling is higher at the file level compared to the function level for both repositories. The STL repository exhibited noticeably higher file-level tangling degrees. This difference could be due to different source code structures in the repositories and extensive use of a shared header file with a large number of preprocessor directives. It is also important to note that most of the tangled commits in the STL repository do not belong to the same commit groups. This further complicates the process of updating forks with features of original repositories and increases the risk of introducing bugs.

This thesis used several information sources available in modern version control systems, such as commit messages, extended descriptions, code diffs, labels, pull requests, linked issues, and commit authors to gather information related to commits.

In conclusion, understanding commit relationships and tangling degrees provides valuable insights into software evolution, maintenance and developer practices. The insights can be helpful in updating fork repositories with original repositories for new available features and functionalities, while still preserving the fork customisations. This study also opens up, as future work, a scenario of upgrades in software product lines with customised extensions.

Bibliography

[1] Idan Amit and Dror G Feitelson. "Corrective commit probability: a measure of the effort invested in bug fixing". In: *Software Quality Journal* 29.4 (2021), pp. 817–861.

[2] Jacob G Barnett, Charles K Gathuru, Luke S Soldano, and Shane McIntosh. "The relationship between commit message detail and defect proneness in java projects on github". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016, pp. 496–499.

[3] Keith H Bennett and Václav T Rajlich. "Software maintenance and evolution: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 73–87.

[4] *Contribution guildelines for the Eclipse repository*. URL: `https://github.com/eclipse-platform/.github/blob/main/CONTRIBUTING.md#creating-a-pull-request` (Last accessed on 08.01.2026).

[5] *diff2html*. URL: `https://diff2html.xyz/#cli` (Last accessed on 09.11.2025).

[6] *Eclipse Platform GitHub*. URL: `https://github.com/eclipse-platform/eclipse.platform` (Last accessed on 09.11.2025).

[7] *Git*. URL: `https://git-scm.com/docs/gitattributes#_defining_a_custom_hunk_header` (Last accessed on 09.11.2025).

[8] *GitHub Octoverse*. URL: `https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/` (Last accessed on 05.01.2026).

[9] *GNU Diffutils*. URL: `https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html` (Last accessed on 09.11.2025).

[10] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, et al. "A fine-grained data set and analysis of tangling in bug fixing commits". In: *Empirical Software Engineering* 27.6 (2022), p. 125.

[11] Kim Herzig and Andreas Zeller. "The impact of tangled code changes". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 121–130.

[12] *JavaParser GitHub repository*. URL: `https://github.com/javaparser/javaparser` (Last accessed on 09.11.2025).

[13] *JavaParser Maven sample project*. URL: `https://github.com/javaparser/javaparser-maven-sample` (Last accessed on 09.11.2025).

[14] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. "Towards a better understanding of software features and their characteristics: A case study of marlin". In: *Proceedings of the 12th international workshop on variability modelling of software-intensive systems*. 2018, pp. 105–112.

[15] Jian Yi David Lee and Hai Leong Chieu. "Co-training for commit classification". In: *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*. 2021, pp. 389–395.

[16] Stanislav Levin and Amiram Yehudai. "Boosting automatic commit classification into maintenance activities by utilizing source code changes". In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2017, pp. 97–106.

[17] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. "An analysis of the variability in forty preprocessor-based software product lines". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 105–114.

[18] *LLVM Compiler Infrastructure*. URL: `https://llvm.org/` (Last accessed on 09.11.2025).

[19] Arthur Roberto Marcondes and Ricardo Terra. "An approach for updating forks against the original project". In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. 2020, pp. 213–222.

[20] *Microsoft STL GitHub*. URL: `https://github.com/microsoft/STL` (Last accessed on 09.11.2025).

[21] *Node.js*. URL: `https://nodejs.org/en/download` (Last accessed on 09.11.2025).

[22] *PR merge checklist for the STL repository*. URL: `https://github.com/microsoft/STL/wiki/Checklist-For-Merging-A-Pull-Request` (Last accessed on 08.01.2026).

[23] Antonino Sabetta and Michele Bezzi. "A practical approach to the automatic classification of security-relevant commits". In: *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE. 2018, pp. 579–582.

[24] Geanderson E dos Santos and Eduardo Figueiredo. "Commit Classification using Natural Language Processing: Experiments over Labeled Datasets." In: *CIbSE*. 2020, pp. 110–123.

[25] *Stack Overflow*. URL: `https://stackoverflow.com/questions/28111035/where-does-the-excerpt-in-the-git-diff-hunk-header-come-from` (Last accessed on 09.11.2025).

[26] *Thesis Source code GitHub Repository*. URL: `https://github.com/sudhahebsur/Master_Thesis` (Last accessed on 20.11.2025).

[27] *Version control systems*. URL: `https://www.atlassian.com/git/tutorials/what-is-version-control` (Last accessed on 16.01.2026).

Statutory Declaration

I hereby certify that I have written this thesis independently and without outside help. I have not used any aids or sources other than those specified by me and I have marked the passages taken from the works used in terms of content and wording as such. I confirm that the electronic version submitted is identical to the printed copies.

Rostock, 18.01.2026

Sudha Vamanraj Hebsur

7 Appendix

7.1 Prompts used for script development

7.1.1 Generation of diff files for commits

Prompt 1: I want a python a script that reads a specific sheet from an Excel file. For each row, the script reads the columns containing the commit serial number and commit ID. Then the script runs a git diff command using the commit ID and redirects the output to a newly created diff file, named after the commit's serial number. For example, for a row with value 1 as commit serial number and 2cef491 as commit ID, the script runs the below git diff command and generates a diff file named 1.diff

Prompt 2: The script is generating diff files with names like 1.0.diff, 2.0.diff. but I want to use integers like 1.diff, 2.diff

Prompt 3: I have some commit IDs in excel sheet enclosed within "" . If the first char is " , then the script should remove the quotes and then use the commit ID

7.1.2 Visualization of file-level tangling

Prompt 1: I want a python script that takes the path to the diff files from a local computer as input. It should extract a list of modified files for each diff file, using the file headers in diff files. It should then find the common modified files for each pair of commits and finally visualize the results using a heat map.

Prompt 2: The diagonal entries in the heat map should be zero, as I don't want to calculate common files with the same commit.

7.1.3 Function-level tangling for the Eclipse repository

Prompt 1: I want java code that takes a file name and a line number as input and return the function name the line number belongs to in the given file name. It should use JavaParser library

Prompt 2: I need a small Python wrapper to call this function using the built local .jar.

Prompt 3: Now, I want a Python script which takes the paths to an Excel sheet (containing commit IDs), the local Eclipse repository and the folder containing diff files as input. For each row in the Excel sheet, the script should read the serial number and commit ID columns, finds the corresponding diff file from the folder based on the serial number from the excel. It should then parses the diff file to find all hunks, using the hunk header format regular expression search and collect the line numbers of the first modified line of all hunks. Using the git show command, the script should retrieve the version of the file at that specific commit ID (read from the excel) and store that content. This temporary file and the line numbers of the hunks are passed to the JavaParser wrapper function to obtain the modified function names. This process is repeated for all hunks across all commits. The script then calculates the number of common modified functions in common modified files for each pair of commits and visualizes the results in the form of a heat map.

Prompt 4: the script is reading the serial numbers from excel as float. They need to be converted to int, since my diff files are of the format 1.diff , 2.diff and so on. The script should also handle empty values for some serial numbers.

7.1.4 Function-level tangling for the STL repository

Prompt 1: I have an excel file , where one sheet has to be processed. 3 columns from the sheet are to be considered: commit_id, Tangled_commits, Tangled_functions_count. all are integers. I want a heat map of commit_id (for 71 commit_id rows, i need 71*71 heat map). for each row of commit_id, pick Tangled_commits, Tangled_functions_count. value of a cell in heat map is commit_id*tangled_commits = tangled_functions_count. In some rows, there are more than one tangled_commits. they are comma separated values. Tangled_functions_count are comma separated in such case. ex: for row with commit_id =1 , Tangled_commits=2,3 and Tangled_functions_count 4,5. the heat map should show it as: [commit_id][tangled_commits]=tangled_functions_count. so here, [1][2]=4 and [1][3]=5

7.2 Additional tool to view diff files

In order to view large diffs, I used a command line interface(CLI) version of a tool called diff2html[5]. It takes the .diff files as an input and generates a HTML view similar to that of GitHub, where users can view the code changes file by file along with the options to minimize and maximize the files view. I downloaded node-v22.16.0-x64.msi from Node.js official website[21] and installed node.js. Later I ran the below command in command prompt to install diff2html-cli.

```
npm install -g diff2html-cli
```

Below command is an usage example that runs on command-line to generate a HTML view of an input diff file named my_diff_file.diff:

```
diff2html -i file -- my_diff_file.diff
```