ABSTRACT

TOWARDS IMPROVING THE SECURITY OF THE SOFTWARE SUPPLY CHAIN

by
Hammad Afzali

A software supply chain comprises a series of steps performed to develop and distribute a software product. History has shown that each of these steps is vulnerable to attacks that can have serious repercussions and can affect many users at once. To address the attacks against the software supply chain, end users must be provided with verifiable guarantees about the individual steps of the chain and with assurances that the steps are securely chained together.

In this dissertation, the security of several individual steps in the software supply chain is enhanced. The first step of the chain, managing the source code, usually relies on a version control system (VCS). A compromised or malicious VCS can corrupt the integrity of the source code (*e.g.*, by inserting a backdoor). Popular web-based repository hosting services such as GitHub lack strong security features that are otherwise available when using stand-alone client tools, such as the ability to sign client commits. Essentially, this means that developers who use the web UI give up the ability to sign their own commits and must fully trust the server. To address this significant issue, `le-git-imate` is proposed that incorporates the security guarantees offered by Git's standard commit signature into web-based Git hosting services.

Another crucial step in the software supply chain is the code review step, which helps to find defects in the software and to improve the readability and consistency of the project's codebase. Unfortunately, current code review systems do not have mechanisms to protect the integrity of the code review process, especially when the code review system is hosted at an untrusted server. To improve this status quo, a set of key design principles is identified that is necessary to secure the code review process.

Then, these principles are used to propose `SecureReview`, a security mechanism that can be applied on top of a Git-based code review system to ensure the integrity of the code review process and provide verifiable guarantees that the code review process followed the intended review policy.

With `SecureReview` in place, auditors have access to verifiable metadata about the code review process so that they can verify whether the code review server tampered with the code reviews. However, this verification process is not only a matter of checking the authenticity and integrity of the code reviews (*i.e.*, verifying a digital signature). It is also about ensuring that a sequence of code reviews that led to the approval of the code changes respects the intended code review policy. Depending on the code review workflow, this process can be quite complex and error prone if done manually. To address this issue, `PolicyChecker` is proposed that allows independent auditors to automatically verify the correctness of the code review process. This tool adequately interprets different code review policies on GitHub and Gerrit and enables automatic verification of a given set of code reviews against a given code review policy. `PolicyChecker` is useful in two steps of the software supply chain: (1) when a maintainer merges a branch, so she does not have to blindly rely on the code review server, (2) when someone pulls a repository and wants to check if the code was merged according to the code review policy.

TOWARDS IMPROVING THE SECURITY OF THE SOFTWARE SUPPLY CHAIN

by
Hammad Afzali

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2021

APPROVAL PAGE

TOWARDS IMPROVING THE SECURITY OF THE SOFTWARE SUPPLY CHAIN

Hammad Afzali

Reza Curtmola, Dissertation Advisor — Date
Professor of Computer Science, New Jersey Institute of Technology

---

Justin Cappos, Committee Member — Date
Associate Professor of Computer Science and Engineering, New York University, New York, NY

Cristian Borcea, Committee Member — Date
Professor of Computer Science, New Jersey Institute of Technology

---

Qiang Tang, Committee Member — Date
Associate Professor of Computer Science, The University of Sydney, Sydney, Australia

---

Abdallah Khreishah, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author:	Hammad Afzali
Degree:	Doctor of Philosophy
Date:	May 2021

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2021
- Master of Information Security,
  Malek Ashtar University of Technology, Tehran, Iran, 2012
- Bachelor of Information Technology Engineering,
  Sharif University of Technology, Tehran, Iran, 2009

Major:	Computer Science

Presentations and Publications:

Hammad Afzali, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos, "SecureReview: Towards verifiable web-based code review systems", *[Under Journal Submission]*, 2021

Hammad Afzali, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos, "Towards adding verifiability to web-based Git repositories", *Journal of Computer Security, volume 28, no. 4, pages 405-436*, 2020

Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes", *28th USENIX Security Symposium (USENIX Security 19), pages 1393-1410*, 2019

Hammad Afzali, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos, "le-git-imate: Towards verifiable web-based Git repositories", *Asia Conference on Computer and Communications Security (ASIACCS 18), pages 469-482*, 2018

To Fatima, Amir, my parents, and my siblings.

ACKNOWLEDGMENT

It has been a long journey with lots of ups and downs, and I am grateful to have experienced every moment, bitter or sweet, of the past years. Today I would not be writing these words if it were not for my advisor, Prof. Reza Curtmola. His guidance has always made me strive to put my best foot forward. I immensely appreciate his invaluable support, encouragement, and commitment.

I am thankful to Prof. Justin Cappos not only for serving on my dissertation committee but also for his unique insight and help throughout my research. I humbly extend my sincerest thanks to other members of my dissertation committee: professors Cristian Borcea, Qiang Tang, and Abdallah Khreishah.

Some friends have been instrumental in this process. I want to thank Ammula Anilkumar for his help in the first stages of my research and Lukas Pühringer for technical supports in different projects. An exceptional thank you goes to Santiago Torres-Arias for his excellent feedback and help in my research.

When it comes to my family, there are no appropriate words to convey my deep appreciation. My parents have always supported me in exploring new directions in my life. I am always grateful for the sacrifices and encouragement they have made from the very beginning, in return for nothing but this moment. I am also quite thankful to my siblings for their endless and unparalleled love, help, and support.

There is no way to thank Amir, who has never known his dad as anything but a busy student. Every day he runs to the window to wave goodbye; he makes me happier, stronger, and more determined than ever before.

My most heartfelt thanks go to Fatima, who selflessly helped me be the person I am today. Forever, I am indebted to Fatima for all her kindness, and nothing can express my gratitude for her. It is my privilege to dedicate this work to my love, Fatima.

Finally, I must thank multiple funding agencies. My research was mainly funded by the National Science Foundation under Grants Numbers CNS 1409523, CNS 1801430, and DGE 1565478. It was also sponsored by the Defense Advanced Research Projects Agency and the Army Research Laboratory under Contract Number A8650-15-C-7521.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter Page

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1
INTRODUCTION

With the rise of defense mechanisms to improve the security posture of the computer systems, traditional attacks (*e.g.*, exploiting vulnerabilities) are becoming more difficult to be successful. In response to this, attackers have recently focused on a new avenue of threats called supply chain attacks which are replacing zero day attacks [77] and allow attackers to get into the authorized software production environment and subvert a large group of users through a single attack. According to a report by Symantec [155], there has been a 200% increase in these attacks in 2017. Moreover, a report by Sonatype [152] shows a growth of 430% in the number of supply chain attacks aimed at infiltrating open source projects.

A software supply chain is a series of steps (*i.e.*, coding, testing, building, distributing) performed to create and distribute a software product. Hijacking each step of the supply chain gives the attackers an entry point to the trusted production environment to implant malware [159]. Such attacks can happen at any step of the software development or between steps (when the software is in transit). For example, the attackers may compromise the coding step [82, 125, 73, 109, 132, 72], or the build step [162, 65, 80, 131] to insert a malicious code. They may compromise the distribution server to replace the legitimate package with a malicious one [138, 111, 107, 143, 129, 31], or even redirect users to install or update a malicious version of the software [68, 62].

To address the attacks against the software supply chain, end users must be provided with verifiable guarantees about the individual steps of the chain and with assurances that the steps are securely chained together. Indeed, the user must be able to check if a step in the supply chain was performed (*e.g.*, the code review was done), if the step was performed by the right person (*e.g.*, the authorized developer updated the codebase), and if the materials (*e.g.*, source code) were not tampered between the steps [159].

In this dissertation, we enhance the security of several individual steps in the software supply chain. The first step of the supply chain is managing the source code which is usually done by a version control system (VCS) like Git. To protect the supply chain against unauthorized changes in the source code, we need to protect both data and metadata stored in the source code repository as the version control systems are not trusted. A compromised or malicious VCS can corrupt the integrity of the source code by inserting a piece of malicious code (*i.e.*, a backdoor) or by manipulating the metadata of the repository.

The existing security features on VCS-es help to mitigate most attacks. Commit signing [88] prevents an attacker from tampering with the repository files. Torres-Arias et al. [160] mitigate attacks against the Git repository metadata. However, web-based hosting repositories lack these security features, and there is no way to verify if web UI commits in a repository have been performed by the authorized developers. That makes web-based Git hosting services an appealing target to get into the trusted supply chain and subvert the integrity of the software product without being detected.

Using web-based repositories (such as GitHub and GitLab), users instruct the server to perform operations on their behalf and have to trust that the server will execute their requests faithfully. A malicious or a compromised server can instead execute the requested actions in an incorrect manner and change the contents of the repository. To counter this, we propose `le-git-imate` [54, 55], a defense that incorporates the security guarantees offered by Git's standard commit signature into Git repositories that are managed via web UI-based services. `le-git-imate` pioneers the ability to sign a web UI commit and create a true GPG-signed Git commit object exclusively in the browser. Subsequently, anyone who clones the repository can check whether the committed data was tampered by a malicious server or other adversaries. Our solution does not require any changes in the repository servers and can be used immediately. With `le-git-imate` in place, users can take advantage of web-based Git services without sacrificing security, thus paving the way towards verifiable web-based Git repositories. We perform a security analysis of `le-git-imate`, which shows its effectiveness in mitigating attacks that may occur when developers use the web UI of web-based Git hosting services. We also evaluate our implementation's efficiency and show that `le-git-imate` has comparable performance with Git's standard commit signature mechanism.

Another crucial step in the software supply chain is the code review step, which helps to find defects in the software and to improve the readability and consistency of the project's codebase. Having a code review system in place, new code changes will become an accepted part of the project's codebase only after being reviewed and approved by a number of developers. There are many web services (such as Gerrit [16], ReviewBoard [43], Phabricator [40], Crucible [11], and Collaborator [9]) that facilitate the code review process by providing a web UI to discuss, review, and even modify the code changes.

Unfortunately, current code review systems do not provide verifiable guarantees about the integrity of the review process. In other words, they do not allow an auditor to validate whether appropriate and authorized code review processes were performed during the software development, and whether no unauthorized change was added into the source code. For instance, a malicious or compromised Gerrit server may change the code review scores to trick the project owner into merging a vulnerable code to the project's codebase. The main security issues with current code review systems can be listed as follows. First, code review systems store the history of a code review only in their local database. That means a full history of the code review process is not accessible in the later steps of the software supply chain. Second, there is no guarantee about the integrity of the stored code review information as it can be manipulated easily by the attackers. That said, there is no way, as an independent auditor, to fully verify whether a review policy was violated during the code review process. To improve this status quo, we must provide: (1) verifiable metadata about the code review process, (2) a way to adequately verify the code review metadata.

To attain these goals, we first propose a set of design principles necessary to secure the code review process. We then use these principles to propose `SecureReview`, a security mechanism that can be applied on top of a Git-based code review system to ensure the integrity of the code review process and provide verifiable guarantees that the code review process followed the intended review policy. Our solution helps the code reviewers to easily sign and store their reviews in the source repository. In addition, it provides the ability to securely update the project during the code review process by signing the commits performed through the web UI.

We implement `SecureReview` as a Chrome browser extension for GitHub and Gerrit. Nevertheless, our design is general enough to be used on any web-based code review service that is integrable with a Git repository. We analyze the security guarantees provided by `SecureReview` and show its effectiveness in mitigating different attacks. We also evaluate the efficiency of our implementation and show that `SecureReview` adds only a slight overhead. Indeed, it can sign and store code reviews in less than half a second and merge changes between half a second (on Gerrit) and two seconds (on GitHub). Moreover, `SecureReview` adds at most 2 KB per Git commit, representing less than 0.0006 of the repository size even for small repositories.

With `SecureReview` in place, auditors have access to verifiable guarantees about the code review process so that they can attest whether the code review server tampered with the code reviews. However, this process is not only a matter of verifying the authenticity and integrity of the code reviews (*i.e.*, verifying a digital signature). It is also about ensuring that a sequence of code reviews which led to the approval of the code changes respects the intended code review policy. Depending on the code review workflow, this process can be quite complex and error prone if done manually. For instance, assume a GitHub project that enforces three approving reviews for any code change, dismissing any reviews after a new code change, ignoring code changes from specific users, and finally requiring review from certain users. Having such a code review policy in place, it is not easy to manually determine if a code change was merged correctly. This could get worse, for example when it comes to validate a software release with many code changes.

To tackle this issue, we present `PolicyChecker`, a code review policy checker that allows independent auditors to verify the correctness of the code review process. `PolicyChecker` not only allows to automatically verify if a given set of code reviews matches a code review policy but also to adequately interpret different code review policies. We implement `PolicyChecker` as a git command which can validate the code review process on GitHub and Gerrit. However, it can be easily adapted for any Git-based hosting services as long as the code review format and the code review policy are known. We analyze our implementation's efficiency and show that `PolicyChecker` can validate the code review process in a repository branch in a timely manner – on average less than 0.12 seconds to evaluate one merge request, and less than 1.50 seconds to verify a software release. Finally, `PolicyChecker` can be useful in at least two steps of the software supply chain: 1) when a maintainer merges a branch, so she does not have to blindly rely on the code review server, which happens a priori to the code merging step, 2) when someone pulls a repository and wants to check if the code was merged according to the code review policy, which happens a posteriori to the code merging step.

CHAPTER 2

ADDING VERIFIABILITY TO WEB-BASED REPOSITORY HOSTING SERVICES

2.1 Introduction

Web-based repository hosting services such as GitHub [23], GitLab [28], Bitbucket [4], Sourceforge [45], Gogs [29], Rhodecode [44], Assembla [3] and many others, have become some of the most used platforms to interact with Git repositories due to their ease of use and their rich feature-set such as bug tracking, code review, task management, feature requests, wikis, and integration with continuous integration and continuous delivery systems. As of April 2021, GitHub reports having over 224 million repositories [98, 100] which represents a growth of more than 2,100% since 2013 [89]. These platforms allow users to make changes to a remote Git repository through a web-based UI, *i.e.*, by using a web browser, and they comprise a substantial percentage of the changes made to Git repositories: 48 of the top 50 most starred GitHub projects include web UI commits and an average of 32.1% of all commits per project are done through the web UI. For some of these highly popular projects, web UI commits are actually used more often than using the traditional Git command line interface (CLI) tool (*e.g.*, 71.8% of merge commits are done via the web UI) [1].

Unfortunately, this ease of use comes at the cost of relinquishing the ability to perform Git operations using local, trusted software, including Git commit signing. Instead, a remote party (the hosting server) is instructed to perform actions for the client. Given that the server performs most of the operations on behalf of the user, it cannot cryptographically sign information without requiring users to share their private keys. Effectively, since GitHub does not support user commit signing, those

[1]These statistics refer to commits after June 1, 2016, when GitHub started to use the `noreply@github.com` committer email for web UI commits, thus providing us with the ability to differentiate between web UI commits and other commits.

who use the web UI give up the ability to sign their own commits and must rely completely on the server.

However, trusting a web-based Git hosting service to faithfully perform those actions may be unwarranted. A malicious or a compromised server can instead execute the requested actions in an incorrect manner and change the contents of the repository. Since Git repositories and other version control system repositories represent increasingly appealing targets, they have been subjected historically to such attacks [65, 67, 68, 125, 73, 169, 132, 109, 72, 162, 103, 82], with varying consequences such as the introduction of backdoors in code or the removal of security patches. Similar attacks are likely to occur again in the future, since design flaws and vulnerabilities may remain undiscovered or undisclosed for a prolonged amount of time, even years [52], and websites may not apply security patches promptly [139, 81].

For example, a user interacting with a GitHub web UI to create a file in the repository can trigger a post-commit hook that adds backdoored code on the same file on the server-side. To introduce such a backdoor, an unscrupulous server manipulates the submitted file and adds it to the newly-created commit object. As a result, from that moment on, the Git repository will contain malicious backdoor code that could propagate to future releases.

To counter this, we propose `le-git-imate` [118], a defense that incorporates the security guarantees offered by Git's standard commit signature into web UI-based Git hosting services such as GitHub or GitLab. `le-git-imate` is implemented as a browser extension and allows tools to cryptographically verify that a user's web UI actions are accurately reflected in the remote Git repository. To achieve this, we present two designs. In the first one, which we refer to as *lightweight design* [54], `le-git-imate` computes a *verification record* on the user side and then embeds it into the commit object created by the server. The *verification record* captures what the user expects to be included in the commit object. Subsequently, anyone who clones the repository can check if the server correctly performed the requested actions by comparing the user-embedded record to the actual commit object created by the server. The first design is used as a stepping stone for the second design, which we refer to as the *main design* [55]. In the *main design*, `le-git-imate` pioneers the ability to sign a web UI commit and create a standard GPG-signed Git commit object in the browser. As a result, there is no difference between signed commits created using `le-git-imate` and those created by Git client tools. This allows users to validate the integrity of web UI commits using standard Git tools. With `le-git-imate` in place, users can take advantage of GitHub/GitLab's web-based features without sacrificing security.

After exploring several strategies to compute the information necessary for the two designs, we settled on solutions that we implemented exclusively in the browser using JavaScript, *i.e.*, as a Chrome browser extension. This covers the large majority of software development platforms (*i.e.*, laptops and desktops). Despite the tedious task of re-implementing significant functionality of a Git client in JavaScript, this approach achieves the best portability and does not require the presence of a local Git client. It also features optimizations that leverage the GitHub/GitLab API to download the minimum set of Git objects needed to compute the *verification record* (for the *lightweight design*) or the commit signature (for the *main design*). The browser extension based on the *lightweight design* contains 15,838 lines of JavaScript code, whereas the one based on the *main design* has 25,611 lines of code (numbers include several third-party libraries needed to create the necessary Git objects and to push these objects to the server). Excluding HTML/CSS templates, JSON manifests, and libraries, the extension consists of a total of 4,095 and 4,467 lines of Javascript code for the two designs, respectively.

In addition to the cryptographic protections suitable for automatic verification, `le-git-imate` provides UI validation to prevent an attacker from deceiving a user into performing an unintended action. To do this, the user is presented with information about their commit that makes it easy to see its impact. This limits a malicious server's ability to trick a user into performing actions they did not intend.

While we focus specifically on `le-git-imate`'s use with GitHub [23] and GitLab [28], our work is applicable to other web-based Git repository hosting services such as Bitbucket [4], RhodeCode [44], Gitea [22], SourceForge [45], and Assembla [3]. Our techniques are also general enough to be used on web-based code management tools that can be integrated with a Git repository (*e.g.*, Gerrit [16] for code reviews, Jira [34] for project management, or Phabricator [40] for web-based software development).

In this chapter, we make the following contributions:

- We identify new attacks associated with common actions when using the web UI of a web-based Git hosting service. In these attacks, the server creates a commit object that reflects a different repository state than the state intended by the user. The attacks are stealthy in nature and can have a significant practical impact, such as removing a security patch or introducing a backdoor in the code.
- We propose `le-git-imate`, a client-side mechanism for Git repositories that are managed via the web UI, to mitigate the aforementioned attacks. `le-git-imate` pioneers creating the true GPG-signed Git commits in the browser. Hence, it provides the exact security guarantees offered by Git's standard commit signing mechanism.
- We implement `le-git-imate` as a Chrome browser extension for both GitHub and GitLab, and have released it as free and open-source software [118]. Our implementation has several desirable features that are paramount for practical adoption: (1) it does not require any changes on the server side and can be used today, (2) it preserves current workflows used in GitHub/GitLab and does not require the user to leave the browser, (3) commits generated by `le-git-imate` can be checked by standard client tools (such as the Git CLI), without any modifications. `le-git-imate` also provides the first implementation of Git's merge commit functionality in JavaScript, which is of independent interest. Last, but not least, unlike other existing libraries [33], `le-git-imate` provides an implementation of Git commands (*i.e.*, "`git commit`", "`git merge`", "`git push`") without needing access to the entire repository and without creating a working directory on the client side.

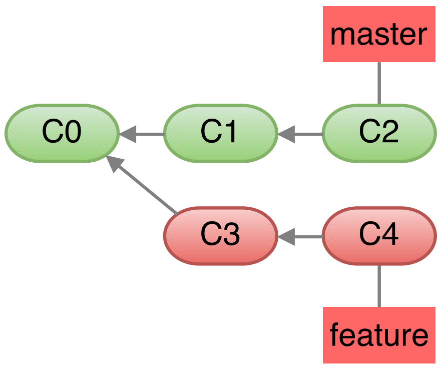


Figure 2.1 A Git repository with two branches, `master` and `feature`.

- We perform a security analysis of `le-git-imate`, which shows its effectiveness in mitigating attacks that may occur when developers use the web UI of web-based Git hosting services.
- We evaluate experimentally the efficiency of our implementation. Our findings show that, when used with a wide range of repository sizes, `le-git-imate` adds minimal overhead and has comparable performance with Git's standard commit signature mechanism.
- We perform a user study that validates the stealthiness of our attacks against a GitLab server. The study also provides insights into the usability of our `le-git-imate` defense.

Together, our contributions enable users to take advantage of GitHub/GitLab's web-based features without sacrificing security. For ease of exposition, throughout this chapter we will use GitHub as a representative web-based Git hosting service, but our attacks and defenses (including the `le-git-imate` browser extension) have been developed and implemented for both GitHub and GitLab.

2.2 Background

GitHub is a web-based hosting service for Git repositories, and its core functionality relies on a Git implementation. In this chapter, we describe several Git and GitHub concepts as background for the attacks introduced in Section 2.4 and the defenses proposed in Section 2.5. Readers familiar with Git/GitHub internals may skip this section.

2.2.1 Git Repository Internals

Git records a project's version history into a data structure called a repository. Git uses *branches* to provide conceptual separation of different histories. Figure 2.1 shows a repository with two branches: `master` and `feature`. As a convention, the `master` branch contains production code that has been verified and tested, whereas the `feature` branch is used to develop a new feature or fix a bug.

A branch can be merged into another branch to integrate its changes into the target branch. When a new feature is fully implemented in the `feature` branch, it may be integrated into the production code by merging the `feature` branch into the `master` branch. For GitHub, this is often achieved via the *pull request* mechanism, in which a developer sends a request to merge a code update from her branch into another branch of the project, and the appropriate party (*e.g.*, the project maintainer) does the merge.

To work as depicted above, a Git repository uses three types of objects: commit objects, tree objects, and blob objects. From the filesystem point of view, each Git object is stored in a file whose name is a SHA-1 cryptographic hash over the zlib-compressed contents of the file. This hash is also used to denote the Git object (*i.e*, it is the object's name).

A blob object is the lowest-level representation of data stored in a Git repository. At the filesystem level, each blob object corresponds to a file. A tree object is similar to a filesystem directory: It has "blob" entries that point to blob objects (similar to a filesystem directory having filesystem files) and "tree" entries that point to other tree objects (similar to a filesystem directory having subdirectories).

2.2.2 Git Signed Commits

Git provides the ability to sign commits: The user who creates a commit object can include a field that represents a GPG digital signature over the entire commit object.

commit <commit object size> **tree** <hash of tree object>

parent <hash of 1st parent commit object>

[**parent** <hash of 2nd parent commit object>]

author <author name> <author e-mail> <timestamp> <time zone>

committer <committer name> <committer e-mail> <timestamp> <time zone>

<commit message>

Figure 2.2 The format of a Git commit object. Bold font denotes pre-defined keywords, and angle brackets (*i.e.*, <>) denote actual values for those fields. Regular and squash-and-merge commits have only one parent, whereas merge commits have two (or more) parents depending on how many branches were merged – we show the case with two parents, the 2nd parent is enclosed between square brackets.

Later, upon pulling or merging, Git can be instructed to verify the signed commit objects using the signer's public key. This prevents tampering with the commit object and provides non-repudiation (*i.e.*, a user cannot claim she did not sign the commit).

However, with a service like GitHub, the server creates a commit object it cannot sign on behalf of the user, as it lacks the cryptographic key material needed for the signature.

2.2.3 Committing Through the GitHub Web UI

For every code revision, a new commit object is created reflecting the state of the repository at that time. This is achieved by including the name of the tree object that represents the project's files and directories at the moment when the commit was done. Each commit object also contains the names of one (or more) *parent* commit objects, which reflect the previous state of the repository. The exact format of a commit object is described in Figure 2.2.

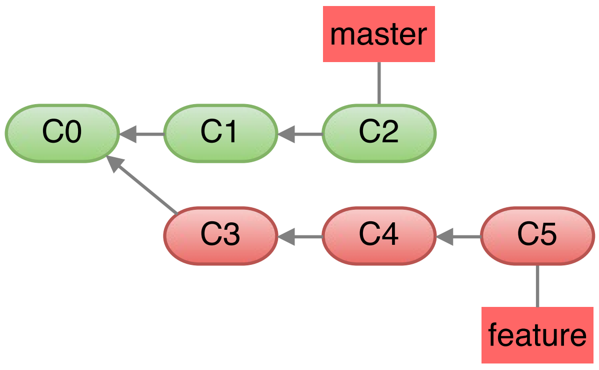


Figure 2.3 A regular commit on the `feature` branch.

Performing a code revision using GitHub's web UI will result in one of three possible types of Git commit objects: *regular commit*, *merge commit*, *squash-and-merge commit*, *rebase-and-merge commit*.

Regular Commit Object. GitHub's web UI provides the option to make changes directly into the repository, such as adding new files, deleting existing files, or modifying existing files. These changes can then be committed to a branch, which results in a new *regular commit* object being added to that branch of the repository. A new root tree is computed by modifying/adding/deleting the blob entries relevant to the changeset in the corresponding trees and propagating these changes up to the root tree. Then, a new commit is added with the new root tree. For example, consider the repository shown in Figure 2.1. Using GitHub's web UI in her browser, a user edits a file under the `feature` branch and then commits this change. As a result, the GitHub server will create a new *regular commit* object C5 that captures the current state of the `feature` branch, as shown in Figure 2.3. Attacks against regular commit objects are described in Section 2.4.1.

Merge Commit Object. Consider a GitHub project in which an *owner* is responsible for maintaining a branch called `master` and *contributors* work on their own branches to make updates to the code. When a contributor completes the changes she is working on, she will send a *pull request* to the project owner to merge the changes

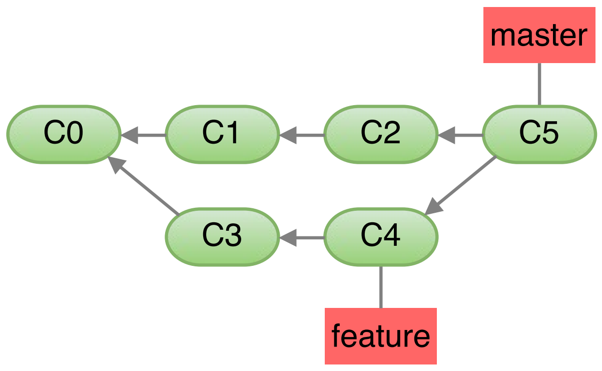


Figure 2.4 Merge commit from merging two branches.

from her branch into the `master` branch. The project owner will review the suggested changes in the *pull request* and will merge them into the `master` branch. This results in a new *merge commit* object as the new head of the `master` branch. This new *merge commit* will contain changes computed using the trees of both branches and the tree of the common ancestor(s). For example, in Figure 2.4, C5 is the merge commit object obtained by merging the `feature` branch into the `master` branch. In this case, C5 has two parents, C2 and C4 [2]. The C5 object is created by the GitHub server as a result of the project owner's action to merge the pull request via GitHub's web UI. We note that the objects C3 and C4 from the pull request branch become part of the `master` branch after the merge. Attacks against merge commit objects are described in Section 2.4.2.1

Squash-and-Merge Commit Object. When a *pull request* contains multiple commits, GitHub provides the *squash-and-merge* option: The commits from *pull request* branch are first "squashed" into a new commit object that retains all the commits but omits the individual commits from its history. This new *squash-and-merge commit* object is then added to the repository. For example, consider the repository shown in Figure 2.1, in which the project owner receives a pull request for the `feature` branch and decides to use the "squash-and-merge" option. As a

[2]We note that, in general, Git allows to merge n branches (with $n \geq 2$), and the resulting merge commit object will have n parents. However, at the moment, GitHub's web UI does not allow merging more than two branches.

result, the GitHub server first creates a new commit object by combining all the changes (commits) mentioned in the *pull request*, as shown in Figure 2.5(a). The server then adds the newly created commit object C5 on top of the current head of the `master` branch C2, as shown in Figure 2.5(b). The "squash-and-merge" option for merging a pull request is preferred when work-in-progress changes (*e.g.*, updates to address reviewer comments) that are important in the `feature` branch are not necessarily important to retain when looking at the history of the `master` branch. Indeed, objects C3 and C4 are not included in the `master` branch, and C5 will have only one parent, which is C2. The new commit object (and tree object) will be computed in the same way as the procedure for the regular commit described above. Attacks against *squash-and-merge* commit objects are described in Section 2.4.2.3.

Rebase-and-merge Commit Object. A pull request may also be merged using the *rebase-and-merge* option: all the new commits from the pull request are placed on top of all the commits in the `master` branch. However, instead of using a merge commit, for each commit in the pull request, a new commit is created in the `master` branch. This option is preferred when it is important not to pollute the history of the repository with a new merge commit object that makes it difficult to follow the evolution of the repository. For example, consider the repository shown in

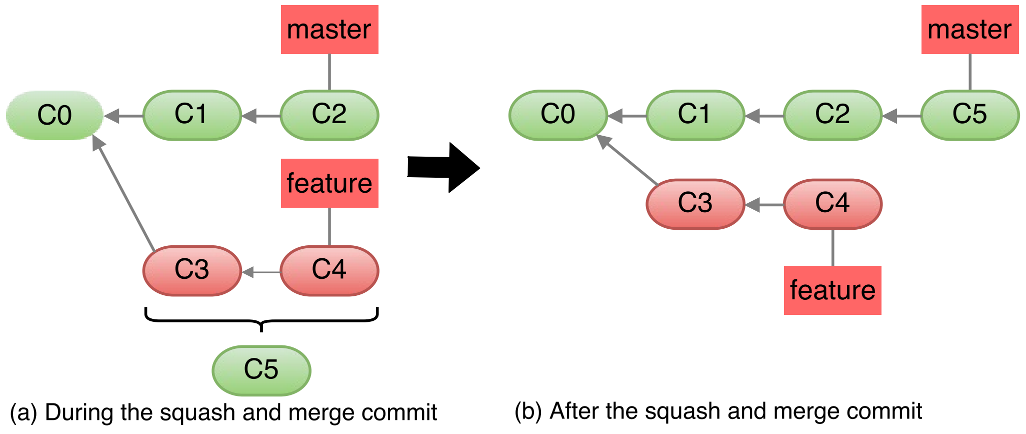

master
C0
C1
C2
feature
C3
C4
C5
(a) During the squash and merge commit
master
C0
C1
C2
C5
C3
C4
feature
(b) After the squash and merge commit

Figure 2.5 Repository state for squash-and-merge operations.

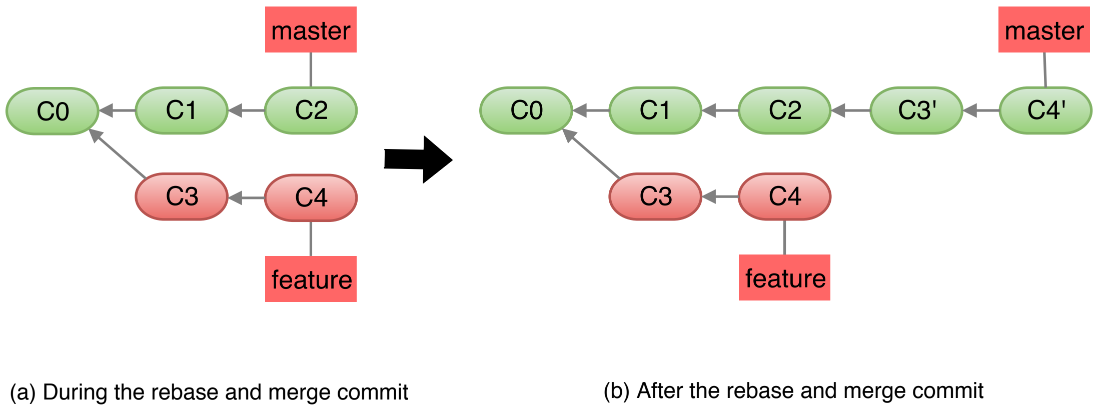

master
C0
C1
C2
C3
C4
feature
(a) During the rebase and merge commit
master
C0
C1
C2
C3'
C4'
C3
C4
feature
(b) After the rebase and merge commit

Figure 2.6 Repository state for rebase-and-merge operations.

Figure 2.6(a), in which the `feature` branch is about to be merged into the `master` branch using the *rebase-and-merge* option. The server creates objects C3' and C4' on top of C2, as shown in Figure 2.6(b). Note that objects C3' and C4' are equivalent to objects C3 and C4 in the `feature` branch (*i.e.*, they point to the same tree object). Attacks against *rebase-and-merge* commit objects are described in Section 2.4.2.4.

2.3 Threat Model

We assume a threat model in which the attacker's goal is to remove code (*e.g.*, a security patch) or introduce malicious code (*e.g.*, a backdoor) from a software repository that is managed via a web interface. We assume the attacker is able to tamper with the repository (*e.g.*, modify data stored on the Git repository), including any aspect of webpages served to clients. This scenario may happen either directly (*e.g.*, a compromised or malicious Git server), or indirectly (*e.g.*, through MITM attacks, such as nation state attacks against GitHub [105, 120, 104]).

There is evidence that, despite the use of HTTPS, MITM attacks are still possible due to different facts ranging from design flaws (POET [144], BEAST [75], CRIME [145], POODLE [128], ROBOT [108], CurveSwap [161], SLOTH [61]), to implementation bugs (*e.g.*, BERserk [148], Heartbleed [47], goto fail [164]) to highly resourced adversaries (FREAK [106], LogJam [53], SWEET32 [60], DROWN [57]).

Such an attacker will continue to violate the repository's integrity as long as these attacks remain undetected. Since commit objects created by the server as a result of user web UI actions are not signed by the user, the attacker may go undetected for a long amount of time. Thus, rather than relying exclusively on the ability of web services to remain secure, client-side mechanisms such as the one proposed in this work can provide an additional layer of protection.

The attacker can read and write any files on a repository that may contain a mix of signed commits (*e.g.*, created via Git's CLI tool) and unsigned commits (*e.g.*, created via the web UI). The integrity of commits not created via the web UI can be guaranteed only if these commits are signed by users using Git's standard commit signing mechanism. Our solution is independent of whether commits not created via the web UI are signed or not. We assume the attacker does not have a developer's signing key they are willing to use (such as insiders that do not want to reveal their identity). As such, the attacker cannot tamper with signed commit objects without being detected. However, commit objects that are not signed can be tampered with by the attacker. Since all commits created via the web UI are not user signed (as is the case with GitHub and GitLab today [3]), the attacker can tamper with these objects when they are created, or directly in the repository after they have been created.

Although the attacker can create arbitrary commits even when users are not interacting with the repository, these commits are not user-signed and will be detected upon verification. Removing an existing commit from the end of the commit chain, or entirely discarding a commit submitted via the web UI are actions that have a high probability of being noticed by developers. Otherwise, our solutions cannot detect such attacks, and a more comprehensive solution should be used, such as a reference state log [160].

[3]In late October 2017, GitHub started to sign commits made using the GitHub web interface (as an undocumented feature). Unfortunately, this only provides a false sense of security and does not prevent any of the attacks we describe in Section 2.4 because GitHub uses its own private key to sign the commits.

We focus on attacks that tamper with commits performed by the user via the web UI (specific attacks are described in Section 2.4). Such attacks: (1) are stealthy in nature, since subtle changes bundled together with a developer's actions are hard to detect, (2) can be framed as if the user did something wrong, (3) can be executed either by attackers than control the Git server, or by MITM attackers in conjunction with a user's web UI actions, and (4) may be performed by an unscrupulous developer who later denies having done it and blames it on the web UI's lack of security. Thus, we are mainly concerned with two attack avenues:

- Direct manipulation of the commit fields, so that the commit does not reflect the user's actions through the web UI.
- Tricking the user into committing incorrect data by manipulating the information presented to the user via the web UI. If not handled appropriately, this attack approach can even circumvent a defense that performs user commit signatures, because the user can be deceived into signing incorrect data.

We assume attackers cannot get access to developer keys. Alternatively, a malicious developer in control of a developer key may not want to have an attack attributed to herself and would thus be unwilling to use this key to sign data they have tampered with.

2.3.1 Security Guarantees

Answering to the threat model, the goal of a successful defensive system should be to enforce the following:

- **SG1: Prevent web UI attacks.** Developers should not be tricked into committing incorrect information based on what is displayed in the web UI.
- **SG2: Ensure accurate web UI commits.** The commits performed by users via the web UI should be accurately reflected in the repository. After each commit, the repository should be in a state that reflects the developer's actions.
- **SG3: Prevent modification of committed data:** An attacker should not be able to modify data that has been committed to the repository without being detected.

2.4 Attacks

A benign server will faithfully execute at the Git repository layer the operation requested by the user at the web UI layer. However, the user's web UI actions can be transformed into damaging operations at the repository layer. In this section, we identify new attacks that can result from some of the most common actions that can be performed using GitHub's web UI. Common to these attacks is the fact that the server creates a commit object that reflects a different state of the repository than the state intended by the user. In a project with multiple files, subtle changes in some of the files may go unnoticed by the user performing the commit via the web UI. As a result, anyone cloning or updating the repository will be unaware they have accessed a repository that was negatively altered.

2.4.1 Attacks Against Regular Commits

GitHub's web UI allows users to manipulate repository data. The user can add, delete, or modify files and directories. The user then pushes a "Commit" button to commit the changes to the repository. As a result, the GitHub server creates a new commit object that should reflect the current state of the project's files. Nevertheless, the server can instead create a commit object that corresponds to a different project state, in which files have been added, deleted, or modified in addition to or instead of those requested by the user.

The attack is easy to execute, as the server simply has to create the blob, tree and commit objects that correspond to the incorrect state of the repository. Nevertheless, the attack's impact can be significant. Since the server can arbitrarily manipulate the project's files, it can, for example, introduce a vulnerability by making a subtle modification in one of the project's files.

2.4.2 Attacks Against Merge Commits

The server can manipulate the various fields of a merge commit object that it creates. Based on this approach, the following attacks can be executed.

2.4.2.1 Incorrect Merge Commit Attacks.

The server can create an incorrect repository state by manipulating the "tree" field of the merge commit object. The server generates an incorrect list of blob objects by adding/deleting/modifying project files, then a tree object that corresponds to this incorrect blob list of blobs, and finally a merge commit object whose "tree" field refers to the incorrect tree object. A project owner or developer will not detect the attack when they clone/update the repository from the server. For example, in Figure 2.4 the `feature` branch is being merged into the `master` branch.

Under benign circumstances, the tree object pointed to by the merge commit C5 object should refer to a set of blob objects that is the union of the sets of blobs referred to by the trees in C2 and C4. However, the server can manipulate the contents of the tree object in C5 to include a different set of blobs. The server can introduce malicious content by adding a new blob that does not exist in the trees in C2 or C4. Or, the server can remove a vulnerability patch by keeping the blob from the `master` over the modified blob in the `feature` branch that contained the patch. Or it can simply not include blobs that contained the patch.

By manipulating the set of blobs pointed to by the tree object, the server can make arbitrary changes to the state of the repository pointed to by the merge commit.

2.4.2.2 Incorrect History Merge Attacks.

The server can also create an incorrect repository state by manipulating the "parent" fields of the merge commit object. Instead of using the heads of the two branches to perform the merge commit, the server can use other commits as parents of the merge commit.

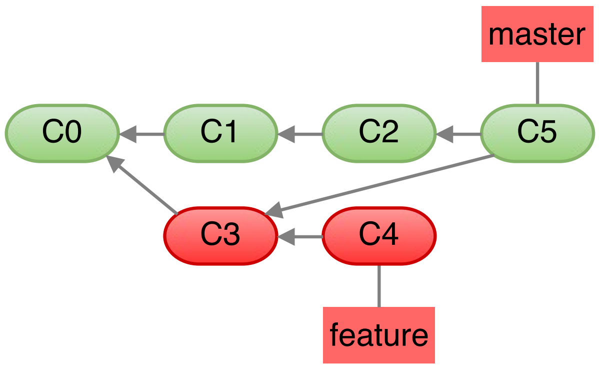


Figure 2.7 Incorrect history merge attack.

Consider the initial repository shown in Figure 2.1. As shown in Figure 2.4, a correct merging of the "master" and "feature" branches should result in a merge commit of C2 and C4 (*i.e.*, the heads of the two branches). However, the server can create the repository shown in Figure 2.7 by merging the head of the `master` branch with C3 instead of C4. This means only the changes introduced in C3 are merged. The "parent" fields of C5 are set to point to C2 and C3.

The impact of this attack can be severe. If C3 contained a security vulnerability, which was fixed by the developer in C4 before submitting the pull request, the fix will be omitted from the master branch after the incorrect merge operation. In a different flavor of this attack, the malicious server merges the head of the `feature` branch (C4) with C1, which is not the head of the `master` branch, thus omitting potentially important changes contained in C2.

Unlike the previous attack described in Section 2.4.2.1, the server does not have to manipulate blob and tree objects, but instead uses incorrect parents when creating the new merge commit object.

2.4.2.3 Incorrect Squash-and-merge Attacks. Consider the same scenario described in Figure 2.1, except that the project owner chooses the *squash-and-merge* option instead of the default recursive merge strategy to merge changes from the `feature` branch into the `master` branch.

As shown in Figure 2.5, the server should first create a new commit object by combining all the changes (commits) mentioned in the pull request, and then should add the newly created commit object C5 on top of C2, which is the current head of the `master` branch. During the creation of C5, a malicious server can add any malicious changes or delete/modify any of the existing changes mentioned in the pull request, and this action may go undetected.

2.4.2.4 Incorrect Rebase-and-merge Attacks. The server can also manipulate a client's request to use the *rebase-and-merge* option to merge changes from a pull request. Consider the merge scenario described in Figure 2.6(a), in which the *rebase-and-merge* option is used to merge the `feature` branch into the `master` branch. As shown in Figure 2.6(b), the server should duplicate the two commits from the pull request on top of C2, the head of the `master` branch. However, a malicious server can add two commits that are not equivalent to the commits in the `feature` branch, and this action may go undetected.

2.4.2.5 Incorrect Merge Strategy Attacks. Git can use one of five different merge strategies when merging branches: *recursive*, *resolve*, *octopus*, *ours*, and *subtree*. Each strategy may in turn have various options. The choice of merge strategy and options influences what changes from the merged branches will be included in the merged commit and how to resolve conflicts automatically (*e.g.*, "favoring" changes in one branch over other branches, or completely disregarding changes in other branches).

We note that web-based Git hosting services such as GitHub and GitLab allow a user to merge two branches using the web UI *only when there are no merge conflicts.* Currently, such services support only the recursive merge strategy with no options. Given their track record of constantly adding new features [94, 24], we adopt a forward-looking strategy and consider a scenario in which they might add support for a richer set of Git's merging strategies.

The merge strategy introduces an additional attack avenue, as an untrusted server may choose to complete the merge operation using a merge strategy different than the one chosen by the user. For example, the server can use a different `diff` algorithm to determine the changes between the merged branches than the one intended by the developer. Or, the server may choose a different automatic conflict resolution than the one preferred by the developer. This can result in removing security patches, or merging experimental code into a production branch. The defenses we propose in Section 2.5 are based on a future-proof design that can also protect against incorrect merge strategy attacks.

2.4.3 Web UI Based Attacks

The server could display incorrect information in the web UI in order to trick the user into committing incorrect or malicious data. Web UI attacks are dangerous because even if a mechanism was in place to allow the user to sign her commits via the web UI, these signatures would only legitimize the incorrect data.

Incorrect list of changes. Before doing a merge commit, the user is presented with a list of changes made in one branch that is about to be merged into the other branch. The user reviews these changes and then decides whether or not to perform the merge. The server may present a list of changes that is incomplete or different from the real changes. For example, the server may omit code changes that introduced a vulnerability. Thus, the user may decide to perform the merge commit based on an incorrect perception of the changes.

Inconsistent repository views. GitHub may provide inconsistent views of the repository by displaying certain information in the web UI and then providing different data when the user queries the GitHub API to retrieve individual Git objects. This might defeat defense mechanisms that rely on the GitHub API.

Hidden HTML tags. A web UI-based mechanism to sign the user's commits may rely on the information displayed on the merge commit webpage to capture the user's perception of the operation. For example, the head commits of the branches being merged may be extracted based on a syntactic check that looks for HTML tags with specific identifiers in the webpage source code. Yet, the server may serve two HTML tags with the same identifier, one of which has the correct commit value and will be rendered in the user's browser, and the other one referring to an incorrect commit that will not be displayed (*i.e.*, it is a hidden HTML tag). The signing mechanism will not know which of the two tags should be used, and may end up merging and signing the incorrect commit – while providing the user with the perception that the correct commit has been merged.

Malicious scripts. The webpage served by the server in a file edit operation for a regular commit may contain a malicious JavaScript script that changes the file content unbeknownst to the user (*e.g.*, silently removes a line of code). As a result, the user may unknowingly commit an incorrect version of the file.

2.5 Design

In this section, we present `le-git-imate`, our defense to address misbehavior by an untrustworthy Git hosting server. The fundamental reason behind these attacks is that the server is fully trusted to compute correctly the Git repository objects. Git's standard commit signature mechanism provides a solution to this problem by having the client compute a digital signature over the commit object and include this signature in the commit object that it creates.

We adopt a similar strategy and present first a solution based on a *lightweight design*, namely to embed a *verification record* in the commit object, even when the client does not generate the commit object. We then present an improved solution, our *main design*, in which the user is able to generate Git standard commit signatures in the browser and therefore can sign web UI commits.

2.5.1 Design Goals

We identify a set of design goals that should be satisfied by any solution that seeks to add verifiability to web-based Git repositories:

1. The solution should embed enough information into the commit object so that anyone can verify that the server's actions faithfully follow the user's requested actions. More specifically, the solution should offer the same (or similar) security guarantees as do regular Git signed commits.
2. For ease of adoption and to ensure that it can be used immediately, the solution should not require server-side changes.
3. The solution should not require the user to leave the browser. This will minimize the impact on the user's current experience with using GitHub.
4. The solution should preserve as much as possible the current workflows used in GitHub: to perform a commit operation, the user prepares the commit and then pushes one button to commit. In particular, the solution should preserve the ease of use of GitHub's web UI and must not increase the complexity of performing a commit, as this may hurt usability.
5. The solution must be efficient and must not burden the user unnecessarily. In particular, the solution should not add significant delay, as this will degrade the user experience and it may hurt usability.
6. The solution should not break existing workflows for Git CLI clients: Regular signed commits can still be performed and verified by Git CLI clients.

2.5.2 A Strawman Solution

A simple solution can mitigate one of the attacks described in Section 2.4.2, the basic attack against merge operations. By default, Git uses the recursive strategy with no options for merging branches. The tree and blob objects corresponding to the merge commit object are computed using a deterministic algorithm based on the tree and blob objects of the parents of the merge commit object.

As a result, the correctness of the merge operations performed by the Git server can be verified. After a user clones/pulls a Git repository, the user parses the branch of interest and computes the expected outcome of all merge operations based on the parents of the merge commit objects. The user then compares this expected outcome with the merge operation performed by the server.

This solution is insufficient because it can only mitigate the simplest attack against a merge commit operation — only when the recursive merge strategy with no options is used, and the server includes an incorrect list of blob objects in the merge commit object by adding/deleting/modifying project files. In particular, this solution cannot handle any of the other attacks we presented, including attacks against regular commits, against merge commits based on incorrect parents or incorrect merge strategy, against squash and merge operations, or web UI-based attacks. Instead, we need a solution that provides a comprehensive defense against all these attacks. In addition, we need to address design and implementation challenges related to the aforementioned design goals.

2.5.3 `le-git-imate` Design

We propose two designs for `le-git-imate`. The *lightweight design* computes a *verification record* on the client side and embeds it into the commit object created by the server. The *main design* gives the user the ability to sign the web UI commits, *i.e.*, the user creates standard Git signed commits. Both designs use information from GitHub's commit webpage as it is rendered in the user's browser, and thus capture what the user expects to be included in the commit object. Subsequently, anyone who clones the repository can check whether the server tampered with the commit objects by traversing the object tree and validating the *verification record* or the commit signatures. We compare the two designs later in Section 2.8.

<original commit message>

[<merge commit strategy>]

<commit size>

<tree hash (hash of tree object)>

<hash of 1st parent commit>

[<hash of 2nd parent commit>]

<author name> <author e-mail>

<committer name> <committer e-mail>

<signature over entire *verification record*>

Figure 2.8 The format of the signed *verification record*. Fields in between square brackets ([ ]) are included only for merge commit objects (merge strategy and hash of 2nd parent commit).

2.5.3.1 ***lightweight design.*** To achieve **design goal #1**, we are faced with two challenges. First, the user cannot compute the same exact commit object computed by the server, because a commit object contains a field, timestamp, that is non-deterministic in nature, as it is the exact time when the object was created by the server. The *lightweight design* takes advantage that, at the moment when the commit object is being created by the server, most of the fields in the commit object are deterministic and can be computed independently by the user. Second, we need to find a way to embed the *verification record* created by the user in the commit object that is created by the server. We add verifiability to the Git repository by leveraging the fact that GitHub (as well as any other web-based Git hosting service) allows the user to supply the commit message for the commit object. The user creates the *verification record* and *embeds the* verification record *into the commit message* of the commit object. The *verification record* contains information that can later be used to attest whether the server performed correctly each of the actions requested by the user through the web UI. By including the *verification record* in the commit message, our solution also meets **design goal #2** – no changes are needed on the server.

We include the deterministic fields of the commit object into the *verification record*, as shown in Figure 2.8. For merge commit objects, we also include the merge commit strategy chosen by the user. All these fields, except the "tree hash", are extracted from the GitHub page where the user performs the commit. The "tree hash" field is computed independently by the user (as described in Section 2.6.2). The user may describe her commit by providing a message in the GitHub commit webpage. However, our solution overwrites the user's message with the *verification record*. To preserve the original user's message, we include it in the *verification record* as the "original commit message" field.

commit <commit object size> **tree** <hash of tree object>

parent <hash of 1st parent commit object>

[**parent** <hash of 2nd parent commit object>]

author <author name> <author e-mail> <timestamp> <time zone>

committer <committer name> <committer e-mail> <timestamp> <time zone>

<commit message>

<**signature over commit fields**>

Figure 2.9 The format of the signed commit object. Fields in between square brackets ([ ]) are included only for merge commit objects (hash of 2nd parent commit).

2.5.3.2 ***main design.*** The main challenge that prevents us from computing a Git's standard commit signature for web UI commits is that the commit timestamp is not determined by the server. To tackle this issue, we create the commit object exclusively on the client side and push it to the server. That lets the user to set the commit timestamp and compute a standard commit signature over all the commit's fields, as shown in Figure 2.9.

When computing the signed commit object on the client side, our *main design* is faced with the challenge to meet **design goal #3**: creating a signed commit object without requiring the user to leave the browser. We pioneer the ability to create a standard GPG-signed Git commit object in the browser by re-implementing the functionality of the `git commit` and `git send-pack` commands exclusively in the browser. That allows the user to create a signed commit object locally and push it to the server (as described in Section 2.6.3). The commit signature can later be used to attest whether the server tampered with the web UI commits. By creating a signed commit in the browser, our solution also satisfies **design goal #2**.

Just like in the *lightweight design*, all commit's fields, except the "tree hash" and the commit timestamp, are extracted from the GitHub page. The "tree hash" field is computed independently by the user (as described in Section 2.6.2). As explained in Section 2.7, `le-git-imate` provides automated and manual checks to mitigate web UI attacks that attempt to trick the user by displaying incorrect information on the GitHub webpage.

2.5.3.3 Verification Procedure. When a developer retrieves the repository for the first time (*e.g.*, `git clone` or `git checkout`), or when she pulls changes from the repository (*e.g.*, `git pull`), she will check the validity of the retrieved commits as follows:

- for *lightweight design*: execute the Verify_Commits procedure which is implemented as a new git command. Note it can be implemented as a Git hook to be executed after a `git clone` or a `git pull` command.
- for *main design*: run the standard `git-verify-commit` command.

Verify Commits Procedure. The developer expects each commit to have either a valid standard commit signature (line 4) or a valid *verification record* (line 8). If there is at least one commit that does not meet either one of these conditions, the verification fails since the developer cannot get strong guarantees about that commit.

The function that validates a *verification record* (Validate_Verif_Record, line 8) returns success only if the following two conditions are true: (a) the *verification record* contains a valid digital signature over the *verification record*; (b) the information recorded in the *verification record* matches the information in the commit object. Specifically, we check that the following fields match: commit size, tree hash, first parent commit hash, author name, author email, committer name, and committer email. For merge commit objects, we also check the merge commit strategy and hashes of additional commit parents. With the verification procedures pointed above, `le-git-imate` achieves **design goal #6**.

PROCEDURE: Verify_Commits
Input: RepositoryName
Output: `success/fail`

```
1:  commits ← Get_Commits(RepositoryName)
2:  for (each commit in commits) do
3:    // Check if the commit is signed
4:    if Validate_Signed_Commit(commit) == false then
5:      commit_msg ← Extract_Commit_Msg(commit)
6:      verif_record ← Extract_Verif_Record(commit_msg)
7:      // Validate the verification record
8:      if Validate_Verif_Record(verif_record) == false then
9:        return fail
10:     end if
11:   end if
12: end for
13: return success
```

2.6 Implementation

With the aim of meeting design goals #2, #3, and #4, we implemented our solution as a client-side Chrome browser extension [5]. After preparing the commit, instead of using GitHub's "commit" button to commit the change, the user activates the extension via a "pageAction" button that is active only when visiting GitHub. The extension is intended to aid the user in creating a *verification record* (for the *lightweight design*) or a standard signed commit (for the *main design*). To do so, the extension first parses the GitHub web UI to obtain the relevant information regarding the user request. That includes the commit type, the head of the repository branch(es). Then the "tree hash" of the new commit is computed. Next, the extension run the following steps depending on which design is implemented.

- for *lightweight design*:
  - Compute the signed *verification record*
  - Include the signed *verification record* into the commit message, and submit the commit to the server
- for *main design*:
  - Compute the signed commit object
  - Push the commit object to the server

In the following, we first give an overview of the implementation of each design. Then, we outline computing the "tree hash" field, which is a core component of both designs. We then describe creating a signed commit object in the browser as the main improvement in the *main design* over the *lightweight design*. Finally, we present the key management component of `le-git-imate`.

2.6.1 Overview

The extension consists of two JavaScript scripts that communicate with each other via the browser's messaging API as follows:

1. The *content script* [70] runs in the user's browser and can read and modify the content of the GitHub webpages using the standard DOM APIs. The content script collects information about the commit operation from the GitHub commit webpage and passes this information to the background script.
2. The *background script* [71] cannot access the content of GitHub webpages but computes the "tree hash". This script then performs automatic and manual checks to prevent web UI-based attacks. In short, the automatic checks ensure that GitHub providing consistent repository views between the web UI and the GitHub API (or any other API used by the Git hosting provider). For the manual checks, the background script allows the user to check the accuracy of the commit fields by displaying it inside a separated pop-up window. If the user is satisfied, she hits a button called "finalize commit". Upon pushing the button, the following steps are performed.

- for *lightweight design*:
  - The background script transfers the *verification record* to the content script.
  - The content script includes the signed *verification record* into the GitHub commit message and triggers the commit button on the GitHub webpage. As a result, the signed *verification record* is embedded into the GitHub repository as part of the commit message.
- for *main design*:
  - The background script creates a signed commit object and pushes the commit to the server.
  - The content script triggers the commit button on the GitHub webpage to reload the page and notify the user about the changes.

Performing a commit using GitHub's web UI requires the user to push one button. With `le-git-imate` in place, the user can commit with two clicks while browsing GitHub's commit webpage (one to activate the extension, and one to transfer the signed commit to the server and reload the GitHub's web page). Based on this design, we argue that `le-git-imate` achieves **design goals #4**.

The extension consists of a total of 4273 lines of Javascript code, excluding HTML templates, JSON manifests, and libraries. All operations to compute commits, signing and verification are done in pure browser-capable Javascript, which required the re-implementation of some fundamental Git functions (such as *git-merge-file* and *git-send-pack*) in JavaScript-only versions. The code to fetch arbitrary information and objects from the repository uses the GitHub API [93], but it could use Git's pack protocol [95] to work with other hosting providers just as well.

At the time of developing the `le-git-imate` extension, previous attempts [27, 35, 26, 13, 33] to implement various Git functions in JavaScript did not offer all the needed functionalities. `le-git-imate` provides the first implementation of Git's merge commit in JavaScript, which is of independent interest[4]. In addition, it implements the `git commit` and `git push` commands without needing access to the entire repository and without creating a working directory on the client side,

[4]We note that isomorphic-git [33] released its first implementation of the `git merge` command based on the diff3 algorithm on September 4, 2019 [112].

which is not possible in the standard Git. Although we implemented `le-git-imate` as an extension for the Chrome browser, it relies purely on JavaScript and can be instantiated in other browsers with minimal effort. We have released `le-git-imate` as free and open-source software [118].

2.6.2 Computing the "Tree Hash" Field

The extension can populate most of the fields of the new commit by extracting them from the GitHub commit webpage, except for the "tree hash" field which needs to be computed independently. We now describe how to compute this field, which is expected to have the same value as the "tree" field of the commit object (*i.e.*, the hash of the contents of the tree object associated with the commit object that is about to be created).

To compute the tree hash, the background script needs the following information, which is collected by the content script and passed to the background script:

- for regular commits: branch name on which the commit is performed, name of the directory(es) that might have been affected by the file operation, and the following file information depending on the user's operation that is being committed:
  - add: the name and content of added file(s).
  - edit: the name and updated content of edited file(s).
  - delete: the name of deleted file(s).
- for merge, squash-and-merge, or rebase-and-merge commits: branch name of the branches that are being merged.

Basic approach 1. The background script can delegate the computation of the tree hash field to a script that runs on the user's local system (outside the browser). The local script runs a local Git client that clones the branch(es) involved in the commit from the GitHub repository into a local repository. The Git client simulates locally the user's operation and performs the commit in a local repository, from where the needed tree hash is then extracted.

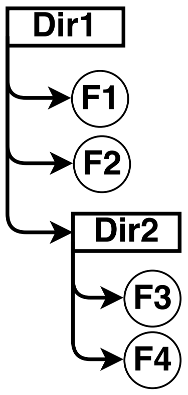


Figure 2.10 An example of the object tree.

Basic approach 2. The previous approach is inefficient for large repositories, as cloning the entire branch can be time consuming. To address this drawback, the client could cache the local repository in between commits. That helps the local Git client to retrieve only new objects that were created since the previous commit.

Optimized approach for regular commits. Delegating the computation of the tree hash field to a local script is convenient since a local Git client will be responsible to compute the necessary Git objects. However, whenever GitHub's web UI is preferred for commits, this usually implies that the user does not have a local Git client. Moreover, assuming that the repository is cached in between commits is a rather strong assumption. We explore an approach in which the tree hash is computed exclusively using JavaScript in the browser. For this, we have reimplemented in JavaScript the regular and the merge commit functionality of a Git client. As such, both designs are implemented exclusively in the browser, without the need to rely on any software outside of the browser, and without assuming any locally-cached repository data. **Design goal #3** is thus achieved.

Instead of cloning entire branches, we propose an optimized approach. Our analysis of the top 50 most starred GitHub projects reveals that for a regular commit performed using GitHub's web UI, only one file is edited on average, and the median size of the changes is 76.5 bytes. For merge commits, the median number of changed files in the pull request branch is 2. The median number of commits in the master branch and in the pull request branch after the common ancestor of these branches are 10.2 and 3.7, respectively. This raises the possibility to compute the tree object without retrieving the entire branch. Instead, we only retrieve a small number of objects and recompute some of the objects in order to obtain the needed tree object.

Our optimized algorithm leverages the fact that GitHub provides an API to retrieve individual Git objects (blob, tree, or commit). We illustrate the optimized algorithm with an example for the object tree shown in Figure 2.10. Assume the user performs an operation on a file under Dir2 and then commits. To compute the tree object for the commit, the background script first retrieves the tree object TDir2 corresponding to Dir2, followed by the following steps, which depend on the performed operation:

- add a file under Dir2: compute a blob entry for the newly added file; re-compute TDir2 by adding the blob entry to the list of entries in TDir2.
- edit a file under Dir2: compute a blob entry for the edited file; re-compute TDir2 by replacing the blob entry corresponding to the edited file with the newly computed blob entry.
- delete a file under Dir2: re-compute TDir2 by removing the blob entry corresponding to the deleted file.

The change in the TDir2 tree object needs to be propagated to its parent tree object TDir1 (*i.e.*, the tree object corresponding to Dir1). To do this, the background script retrieves the TDir1 tree object using GitHub's API, and then updates it by changing the tree entry for TDir2 to reflect the new value of TDir2. In general, the propagation of changes to the parent tree object continues up until we update the "root" tree object which corresponds to the commit object that will be created by the server. This "root" tree object is the tree object that we need to compute.

Unlike the basic approach 1 presented earlier, this optimized approach proves to be much faster (as shown by our evaluation in Section 2.8) and does not require a

Git client installed on the user's local system. We note that all Git objects retrieved through the API are verified for correctness before being used (they need to either have a `le-git-imate` *verification record*, or a true Git commit signature).

Optimized approach for merge (and squash-and-merge) commits. We describe this approach for a case of merging two branches: the `feature` branch is merged into the `master` branch. However, it can be extended in a straightforward manner to handle the merging of multiple branches. Just like in the optimization for regular commits, we leverage the GitHub API for retrieving a minimal set of repository objects that are needed to compute the tree object for the merge commit. The merge commit is a complex procedure that reconciles the changes in the two branches into a merge commit object. At a high level, the tree of the merge commit (*i.e.*, the merge tree) is obtained by merging the trees of the head commits of the two branches. Similarly, we initialize the merge tree using the tree of the `master` branch, and then add/update/remove its entries to reflect the blobs were added, modified, or deleted in the `feature` branch. To determine the lists of added, modified, and deleted blobs in the `feature` branch, we use the following algorithm:

1. Retrieve the tree of the head commit of the `feature` branch. Let L1 be the list of all the blob entries in this tree.
2. Retrieve the tree of the commit that is the common ancestor of the two branches. Let L2 be the list of all the blob entries in this tree.
3. Given lists L1 and L2:
    - if a blob entry exists in both lists (*i.e.*, same blob path), but the blob has different contents (*i.e.*, different SHA1 hash), then add the blob entry to the list of modified blobs.
    - if a blob entry exists in L1 and does not exist in L2, then add it to the list of added blobs.
    - if a blob entry exists in L2 and does not exist in L1, then add it to the list of deleted blobs.

Since the entries in the trees retrieved from the GitHub API are already ordered lexicographically based on the paths of the blobs, this algorithm can be executed efficiently (execution time is linear in the number of tree entries). Having obtained the lists of blobs that were added, modified, and deleted in the feature branch, we add to the merge tree the entries for the blobs that were added, and remove the entries for the blobs that were deleted. For modified blobs, we update the corresponding entries as follows: We use the GitHub API to retrieve the corresponding blobs from the two branches and then compute the modified blob via a three-way merge.

We note that changes in the tree of a subdirectory have to be propagated up to the tree of the subdirectory's parent directory. Similar to our optimization for regular commits, the propagation of changes to the parent tree object continues up until we update the "root" tree object which corresponds to the merge commit object that will be created by the server. This "root" tree object is the tree object that we need to compute.

2.6.3 Creating and Sending the Commit Object to the Server

`le-git-imate`'s *main design* gives the user the ability to sign the web UI commits in the browser. To do so, the extension creates the signed commit object and pushes it to the server by reimplementing the functionality of the `git commit` command and `git send-pack` command, respectively. We note that, unlike other existing libraries such as isomorphic-git [33], we implement this functionality without needing access to the entire repository and without creating a working directory on the client side.

Create the signed commit object. Once the deterministic fields of the new are extracted from the GitHub page, `le-git-imate` simulates the *git-commit* command to create the commit object as follows.

1. determine the commit timestamp locally.
2. compute the "tree hash" as detailed in Section 2.6.2.
3. compute a Git's standard commit signature over the commit's fields.
4. create the commit object and insert the commit signature into it.

5. create all new blob and tree objects related to the new commit.

Push the commit object. Git provides transferring data between two repositories using two sets of protocols, a pair for pushing data from the client to the server, and another for fetching data from the server to the client. Both protocols can be run over ssh, http(s) or ftp. To push data to a remote server, Git uses two processes,"*send-pack*" and "*receive-pack*". The *send-pack* process runs on the client and connects to a *receive-pack* process on the server. The entire protocol is aimed to publish what is updated locally to the remote repository on the server.

The data is sent over a custom file called "*Packfile*" that is a file used to store Git objects in a highly compressed format. Git objects are normally stored in the "*Loose*" format which is storing each entire version of a file in the repository. Unlike the *Loose* format, the *Packfile* stores a single version of a file, and maintain different patches to derive the other versions of the file.

When the user wants to update a remote repository, Git client runs the *send-pack* process to initiate a connection to the server. The *receive-pack* process on the server responds with the server's state, specifying the head of each branch. Using the server's response, the client determines the smallest amount of data should be sent to the server (commits that the server does not have). Then, the client uses *send-pack* process to tell the server which branches are going to be updated. For each branch, the client sends the old head and the new head. Next, the client sends a *Packfile* of all the objects the server does not have. Finally, the server replies with a success (or failure) indication.

To run the protocol depicted above, `le-git-imate` can simply use Git client to run the "*git-push*" command. However, **design goal #3** prevents us from leaving the browser. `le-git-imate` tackles this challenge by simulating the entire protocol in the browser. In sum, `le-git-imate` re-implements the following functionalities of a Git client in the browser:

1. run *git-receive-pack* process to talk to the server and get the state of the remote repository.
2. create the packfile of all the objects (i.e, commit, blob, tree) that the server does not have.
3. send the packfile to the server.
4. get the server's respond (a success or failure).

2.6.3.1 Key Management. Key management can be either performed manually or in an automated fashion. `le-git-imate` provides users with both options to manage the private key that is used to sign their commits.

- Automatic (Private key store): `le-git-imate` asks the user to log into a third-party private key store. Upon successful login, the user's private key is retrieved from the third-party server. Then the key is stored locally to avoid asking it every time that the user wants to perform a commit. However, if the user prefers not to cache the private key locally, she must authenticate to the third-party server once per commit.
- Manual (Import local keys): The extension supports manual key management for those users who dislike storing even a passphrase-protected private key on a third-party server. Such users have options to either load an existing private key or generate a new one.

Out of several key management systems ([36, 78, 119]), we leverage Keybase [36] as a private key store based on its relatively high popularity (over 400,000 active users[5]) and on its rich set of APIs. It allows users to store passphrase-protected private keys on Keybase servers without trusting the Keybase servers. This system simplifies the private key management by allowing users to retrieve their private key from a server anywhere anytime and even use one private key across different devices. We note that Keybase puts the private keys at risk if the passphrase is compromised.

We note that GitHub has recently introduced a feature to verify GPG signed commits using the public key of the signer [96], which is stored and managed by GitHub. Unfortunately, relying on an untrusted server to manage user keys does not fit our threat model, and so `le-git-imate` does not leverage this feature.

[5]as of August 28, 2019 [74]

2.7 Security Analysis

In this section, we analyze the security guarantees provided by `le-git-imate`.

2.7.1 Prevent Web UI Attacks

`le-git-imate` relies in part on extracting information from the commit webpage in order to compute the *verification record* (for *lightweight design*) or the commit signature (for *main design*). To prevent the web UI attacks described in Section 2.4.3, `le-git-imate` has additional checks that retrieve Git objects via the API, and verify their correctness before use based on a *verification record* or a true Git commit signature.

To defend against a server that presents an incorrect list of changes before a merge commit, we use the API to compute independently the list of changes based on the heads of the branches that are being merged. We then compare it with the list of changes presented on the webpage, and alert the user of any inconsistencies. Since the "hash tree" field is computed based on Git objects retrieved via the API, the GitHub server has to create commit objects that are consistent with the commit signature. Otherwise, any inconsistencies will be detected when the verification procedure is run.

To counter the hidden HTML tags attack, we leverage the fact that a benign GitHub merge commit webpage should present only one HTML tag describing the number of commits present in the branches being merged. If more than one such tag is detected, we notify the user. We also inform the user about the number of commits that should be visible in the rendered webpage, and the user can visually check this information. Assuming there are n commits, we then check that there are n HTML tags describing a commit and report any discrepancy to the user as well.

Before pushing the commit to the server, `le-git-imate` displays in a pop-up window three text areas as follows:

1. information about parent commit (author, committer, and creation date), retrieved via the API. This helps the user to detect if the new commit is added on top of a commit other than the head of the branch.
2. for regular commits, the differences between the parent commit (retrieved via the API) and the commit that is about to be created. This allows the user to detect any inconspicuous changes made by malicious scripts in the commit webpage.
3. the commit's field. This allows the user to check if the fields of the new commit match the information displayed on GitHub's commit webpage.

Whereas these checks may not be 100% effective since they are done manually by the user, they provide important clues to the user about potential ongoing attacks by the malicious scripts. Notably, that the pop-up window could be integrated with the original GitHub webpage. However, the content of GitHub page can be manipulated by malicious scripts coming from the un-trusted server or other adversaries. To mitigate this threat, `le-git-imate` uses an isolated pop-up window to display the signed information. That is a standard approach followed by similar extensions like Mailvelope [37] and FlowCrypt [15]. From above discussions, we argue that `le-git-imate` achieves **security guarantee SG1**.

2.7.2 Ensure Accurate Web UI Commits

Creating a signed *verification record* or a true Git commit signature, `le-git-imate` uses information about the user (*i.e.*, author, committer), the state of the repository (*i.e.*, the head of the branch), and the user's requested actions (*i.e.*, the "tree hash" computed over the changes made by the user). To capture such information, `le-git-imate` relies on the content of GitHub webpage and a minimum set of Git objects retrieved from the server. Since all this information is verified before use (as described in Section 2.7.1), `le-git-imate` can not be tricked to use incorrect information. Thus the commits created by `le-git-imate` reflects the user's actions, and our solution achieves **security guarantee SG2**.

2.7.3 Prevent Modification of Committed Data

Commits created by `le-git-imate` extension, either have a signed *verification record* or a true commit signature. In either case, the signature is calculated over a payload of information, including the commit size, the "tree hash", the hash of parent(s) commit, and the commit's author. That prevents the attackers from changing the committed data without being detected. Indeed, any unauthorized modifications (*e.g.*, delete a file or change the author) in the committed data will be caught during the verification procedure (described in Section 2.5.3.3).

We, therefore, conclude that `le-git-imate` achieves **security guarantee SG3**.

2.8 Evaluation

In this section, we study the performance of our browser extension prototype to see whether it meets **design goal #5**. Specifically, we investigate whether the time to sign a web UI commit remains within usable parameters for our different implementations. In addition, we consider the tradeoffs between setup time and disk space required.

For this evaluation, we covered five variants of our tool:

- No-Cache: In this approach, a local Git CLI client clones an entire branch and computes the new Git commit object, whereas the browser extension computes the verification record based on information from the new commit. This is the "Basic approach 1" described in Section 2.6.2.
- Cache: This approach is the same as above, but it uses a local copy of the repository (as cache). Thus, the client retrieves only new objects that were created since the previous commit. Based on our findings of the top 50 most starred GitHub projects, we assume a cached local repository is behind the remote repository by 4 commits (for a regular commit) and by 10 commits (for a merge commit). This corresponds to the "Basic approach 2" described in Section 2.6.2.
- NativeSign: A baseline approach in which the local script of the extension performs a signed commit locally using a Git client. This is the same as the Cache approach, however, it results in a standard signed Git commit object.

- Optimized1: An optimized approach based on the *lightweight design* [54], that queries for Git objects on demand to compute the verification record exclusively in the browser. This does not require a local repository nor any additional tools outside of the browser.
- Optimized2: An optimized approach based on the *main design* [55], that queries for Git objects on demand to create signed commit objects exclusively in the browser. Compared to the Optimized1 variant, it creates a standard signed Git commit object on the client side.

2.8.1 Experimental Results

To test our implementations against a wide range of scenarios, we picked five repositories of different history sizes, file counts, directory-tree depths and file sizes, as shown in Table 2.1. To simulate real-life scenarios, they were chosen from the top 50 most popular GitHub repositories (popularity is based on the "star" ranking used by GitHub, which reflects users' level of interest in a project[6]).

The client was run on a system with Intel Core i7-6820HQ CPU at 2.70 GHz and 16 GB RAM. The client software consisted of Linux 4.8.6-300.fc25.x86_64 with git 2.19 and the gnupg 2-2.7 library for 2048-bit RSA signatures. Experimental data points in the tables of this section represent the median over 30 independent runs. For all variants, the time to push the Git commit object to the server is not included in the measurements. When running the five variants of our tool, we noticed that one CPU core (out of 8 cores) was used.

We note that, compared to an earlier version of this article [54], the experimental numbers in this section for the Optimized1 of `le-git-imate` are smaller. This is because we have improved the implementation of that variant by reducing the number of GitHub API calls by two times for regular commits (from four to two API calls) and by four times for merge commits (from twelve to three API calls).

Regular commits. Table 2.2 shows the execution time for regular commits for all variants of our tool. A regular commit consists of editing a file that is two subdirectory

[6]The statistics refer to the top 50 GitHub projects as of August 25, 2019.

Table 2.1 Repositories Chosen for the `le-git-imate`'s Evaluation

Repo.	Size (MB)	File Count	File Size (Bytes)	History Size (# of commits)
gitignore	1.6	193	496	2,883
vue	11.4	526	9,438	2,611
youtube-dl	44.5	916	6,080	16,447
react	87.3	935	10,695	10,199
atom	290.4	827	20,021	35,507

levels below the root level and committing the changed file (we also measured the time for commits in a subdirectory nested up to four levels below the root level, but the difference is negligible – under a tenth of a second). The size of the changes for the edited file was 1.2 kilobytes, which is the maximum size of the changes observed for the top 50 most starred GitHub projects.

In the case of the No-Cache variant, the execution time is dominated by the time to clone the repository. Notice that this only requires to retrieve one commit object with all its corresponding trees and blobs, which leaves little space for optimization. In contrast, the Cache and NativeSign variants are barely affected by network operations since only new objects are retrieved from the remote Git repository.

The optimized variants fetch the minimum number of Git objects needed to compute the commit object. As a result, they are influenced by two factors: 1) the number of changed files and 2) the location of these files in the repository, which determines the number of tree objects needed to be retrieved. In particular, repository size is not a major factor for the performance of the optimized variants.

It is important to point out that the execution time for the optimized variants is dominated by the time to retrieve the Git objects from the remote server over the network. On average, Optimized1 is about 300ms faster than Optimized2 due to the

Table 2.2 Regular Commit Execution Time (in seconds)

Repo.	No-Cache	Cache	NativeSign	Optimized1	Optimized2
gitignore	0.37	0.16	0.16	0.33	0.61
vue	0.85	0.22	0.22	0.35	0.62
youtube-dl	4.01	0.22	0.23	0.32	0.66
react	5.46	0.25	0.25	0.36	0.63
go	14.81	0.20	0.21	0.45	0.72

Table 2.3 Merge Commit Execution Time (in seconds)

Repo.	No-Cache	Cache	NativeSign	Optimized1	Optimized2
gitignore	0.41	0.17	0.17	0.87	1.11
vue	0.92	0.36	0.36	0.90	1.19
youtube-dl	4.39	0.22	0.23	1.06	1.39
react	5.73	0.29	0.30	1.31	1.53
go	15.59	0.25	0.25	1.16	1.41

fact that Optimized2 needs to create a packfile of all new objects the server does not have. That includes, as explained in Section 2.6.3, making additional network connections. Our experimental results show that if we exclude time to create the packfile, Optimized2 has a similar performance with Optimized1.

Finally, we point out that optimized variants use the OpenPGP Javascript library [38] to compute in the browser a digital signature for the verification record or for the commit object. As opposed to that, computing signatures in the Cache and NativeSign variants is faster because it is done by the Git client, which is optimized for specific architectures. If we exclude the signature creation time, Optimized1 exhibits similar performance with NativeSign.

Merge commits. Table 2.3 shows the execution time for merge commits for all variants of our tool. A merge commit is created by merging into the `master` branch an open `pull request` branch that has no conflicts. Each number in the table is the median over merging the last open 30 pull requests for each repository (at the time when the experiment was performed). As such, each pull request consists of a number of commits ranging from 1 to 16, and a number of changed files ranging from 1 to 75.

None of our variants have complexity worse than linear. Similarly to the regular commit experiment, the No-Cache variant exhibits a running time linear with the size of the repository. Likewise, the Cache and NativeSign variants exhibit a slightly higher time for merge commits when compared to regular commits due to the computation of the merge operation itself.

The optimized variants perform under 1.5 seconds for all cases – regardless of repository size, because the time it takes to perform the operation depends on the number of changed files and directories in the target branch and in the pull request branch. This explains why the time for the "react" pull request is higher than for "go", which is a bigger repository. Similarly to regular commits, the Optimized2 variant is about 300ms slower than Optimized1 on average because it creates the packfile of all new Git objects that are necessary.

2.8.2 User Experience

From the results above, we concluded that a No-Cache version is out of usable parameters due to its high execution time. However, the Cache and Optimized versions perform well under website responsiveness metrics.

Work by Nielsen and Miller [135, 124, 134] suggests that a response under a second is the limit in which the flow of thought stays uninterrupted, even though the user will notice the delay. From then on, and up to 10 seconds, responsiveness is harmed, with 10 seconds being a hard limit for the time a user is willing to spend waiting for a website's response. Further work [83, 149] presents an "8 second rule" as a hard limit in which websites should serve information. In addition, work by Nah [130] sets a usable limit of around two seconds if there is feedback presented to the user (*e.g.*, a progress bar). Work of Arapakis [56] argues that $1000ms$ of increased response time is still hard to notice by some users, depending on the nature of the activity. Finally, further studies suggest that response times that range from two seconds to seven seconds are associated with low user drops (and high conversion rates), given that users are engaging in activities understood to be complex [141]. Using GitHub's web UI for actions such as code commits and merge commits usually requires the user to review the code changes, which can take from seconds to minutes.

Under these considerations, and in the context of the above experiments, we conclude that the Cache, NativeSign, and Optimized versions fall under usable boundaries.

2.8.3 Disk Usage and Other Considerations

Among the three implementations, NativeSign requires to store a local copy of the repository. In contrast, the Optimized versions run entirely on the browser and with fairly minimal memory requirements.

Likewise, the Optimized versions do not require a local installation of a Git client, a shell interpreter, and any other tools. The size of this Optimized implementation is much smaller than the official Git binary (as of version 2.19). The disk space needed for the whole extension is 465KB for the Optimized1 version and 735KB for the Optimized2 version. If we also consider dependencies (which include other JavaScript libraries that are needed), the storage grows to 1.2MB and 1.67MB, respectively.

Finally, we contrast the required configuration parameters, such as paths to executables, cache paths, and private key settings. In this case, the Optimized versions also shine in contrast to the remaining three. Since all operations are performed in-browser, the Optimized variants can almost work out of the box, as they only require configuring the key for signing the *verification record* or the commit.

Due to the reasons outlined above, we consider our Optimized variants to fall under reasonable parameters for usability. We conclude that, with minimal disk and memory footprints, minimal configuration parameters, and reasonable delays, our optimized implementation meets **design goal #5**.

2.8.4 Comparison Between the *lightweight* and *main* Designs

In this section, we compare the two designs by summarizing their various advantages and drawbacks:

- Verifiability and Compatibility with Existing Workflows: The *main design* computes standard Git signed commits which can be verified with the standard Git CLI tool. The *lightweight design* introduces a verification record which requires adding a Git command to the Git CLI tool in order to perform verification. This may require slight changes to existing workflows, as the verification now relies on information that exists in the commit message.
- Security: The *main design* provides the exact security guarantees offered by Git's standard commit signing mechanism. The *lightweight design* provides security guarantees comparable and compatible with Git's standard commit signing mechanism
- Performance: Both designs have comparable performance with Git's standard commit signature mechanism. However, the *lightweight design* is slightly faster than the *main design*, because it does not need to create a packfile of all new objects on the client side.
- Storage: The *lightweight design* has smaller storage and memory requirements (15,838 lines of JavaScript code and 1.2MB) compared to the *main design* (25,611 lines of JavaScript code and 1.67MB).
- User interface: Both designs have the same user interface.

We conclude that the *main design* is preferable in general due to its full compatibility with existing workflows, but the *lightweight design* may be preferable when performance and storage are critical and even a slight improvement in these parameters would make a difference.

2.9 User Study

Having received IRB approval, we conducted a user study on 49 subjects with two primary goals in mind. The first goal was to evaluate the stealthiness of our attacks against web-based Git hosting services. The second goal was to evaluate the usability of our `le-git-imate` browser extension when used by Git web UI users.

2.9.1 User Study Setup

In order to measure user's interactions with the web-based Git UI, we hosted an instrumented GitLab server using Flask [14] and the original GitLab source code [28]. For each participant, we assigned a copy of the `retrofit` repository, which is among the top 5 most starred GitHub projects in Java. We chose `retrofit` due to the participants' familiarity of Java and the repository being representative for a medium-to-large repository size (1503 commits, 265 files and 4.5KB average file size).

Our study used the `le-git-imate` implementation based on the *lightweight design* [54]. We argue that the results are applicable to both designs because their implementations have the same graphical user interface and have very similar performance (as shown in Section 2.8).

The subjects were recruited as volunteers from the student population at our institutions, with a majority of them receiving extra course credit as an incentive to participate. After a screening process to ensure that participants had a basic understanding of Git and GitHub/GitLab services, 49 subjects took part in the study. We also discarded six additional participants given that they were unable to complete any or most of tasks in the user study. Table 2.4 provides demographics about the study participants.

Table 2.4 Demographics for User Study Participants

Subjects	43
GENDER	
Male	33
Female	10
AGE	
20 to 25 years	34
25 to 35 years	8
35 years or older	1
GITHUB/GITLAB MEMBERSHIP	
More than 2 years	13
Between 1-2 years	18
Less than 1 year	6
Less than 6 months	3
Not using a web-based Git repository	3
GITHUB/GITLAB USE	
A few times per day	5
Once per day	4
A few times per week	17
A few times per month	15
Not using GitHub/GitLab	2
FAMILIARITY WITH GIT COMMIT SIGNING	
Very familiar (use it on a daily basis)	6
Somewhat familiar (use it sometimes)	23
Not familiar (never use it)	14
FAMILIARITY WITH PUBLIC KEY CRYPTOGRAPHY	
Very familiar	14
Somewhat familiar	27
Not familiar	2

2.9.2 User Study Description

The study consisted of two parts, each of which comprises several tasks. Each task required participants to interact with the GitLab web UI in order to perform either a branch merge, or to edit, add, or delete one file in their copy of the `retrofit` repository.

During the first part, we collected a baseline usability data of the GitLab web UI usage, as well as the participants' ability to detect any of our GitLab web UI attacks. Participants had to perform 10 tasks, 4 were related to merge commits operations and 6 were related to regular commits using the web UI. To test the attack-stealthiness aspect, the GitLab server would maliciously transform their actions using a pre-commit hook on 5 out of the 10 tasks.

During the second part of the user study, which consisted of 8 tasks (of which 4 were merge commits and 4 were regular commits), we tried to measure the usability of our `le-git-imate` browser extension. Subjects were asked to perform the commits using the `le-git-imate` browser extension (which subjects were asked to install during the study) and a newly-generated GPG key.

To measure the stealthiness of the attacks, we asked the subjects if they think that the GitLab server performed the tasks correctly after they were done with both parts. While answering this question, access to the GitLab repository was disabled, to ensure the users only noticed the attacks *before* being asked explicitly about them.

In order to assess the usability of the tool and the web UI usage, we recorded the time taken to perform each task. We compared the time taken to perform similar tasks with and without the extension in order to assess the burden our tool adds to the time users take to perform operations. In addition, the subjects were then asked to rate the usability of the browser extension on a scale of 1 to 10 (1 = least usable, 10 = most usable).

Finally, in order to gain additional insight about the users' individual answers, they were required to answer a few general questions about their experience level with using web-based Git hosting services and demographic questions (age, gender, etc.).

We note that due to two reasons, the usability of the extension was evaluated only for the *lightweight design* of `le-git-imate`. First, both designs benefit from the same graphical user interface. Second, they have almost the same performance as detailed in Section 2.8. Thus different designs have no impact on the usability of the `le-git-imate` extension, and the results are applicable for both designs.

2.9.3 User Study Results

While performing the study, a user could fail on performing a task by either performing a wrong type of commit than the one required, or because the user did not perform any commit (*i.e.*, a *skipped* task). Tasks that were skipped in a time in which a user did not spend a realistic time to attempt the task (*i.e.*, less than 4 seconds), were labeled as *ignored tasks*.

Attack stealthiness. During the first part of the study, we expected that a few participants would detect some of the attacks, especially those that made widely-visible changes to the repository (such as those that changed multiple files in the root-level). However, results indicate the opposite, as no participant was able to detect any attacks. The reason behind it may be that most users are not expecting a Git web UI to misbehave.

Extension usability. We evaluate the usability of our extension based on several metrics: percentage of successful tasks and average completion time for tasks in Part 2 compared to tasks in Part 1, and direct usability rating by participants.

In Part 1, subjects were able to successfully complete on average 97.6% of the tasks (9.76 out of 10). The average time needed to perform a task was *63 seconds*. In Part 2, subjects were able to successfully complete on average 92.1% of the tasks (7.37 out of 8). Nevertheless, if we discard the ignored tasks (which subjects may have skipped due to a lack of interest), the successful completion rate increases to 94.8%. It is worth nothing that 10 participants had to perform the same task twice, as they performed it the first time without using the extension. Notably, once they realized their mistake, they performed the rest of the tasks using the extension. In Part 2, the average time needed to perform a task was *44 seconds*.

Interestingly, the tasks in Part 2, which are using our browser extension, were completed faster than those in Part 1. This is likely because users familiar with GitHub, but not with GitLab, initially needed some time to learn how to perform various types of commits in GitLab. The extension received a direct usability rating of 8.3 on average.

2.10 Related work

This work builds on previous work in three main areas: version control system (VCS) security, security in VCS-hosting services, and browser/HTML-based attacks. In this section, we review the primary research in each of these areas.

Security of VCS-es. Wheeler [163] provides an overview of security issues related to software configuration management (SCM) tools. He puts forth a set of security requirements, presents several threat models (including malicious developers and compromised repositories), and enumerates solutions to address these threats. Gerwitz [87] provides a description of creating and verifying Git signed commits by focusing on mechanisms to sign commit data remotely via a web UI on an untrusted server. Commit signatures were also proposed for other VCS systems, such as SVN [147]. This work focuses on providing mechanisms to sign commit data remotely via a web UI on an untrusted server. There have been proposals to protect sensitive data from hostile servers by incorporating secrecy into both centralized and distributed version control systems [2, 140]. Shirey *et al.* [150] analyze the performance trade-offs of two open source Git encryption implementations. Secrecy from the server might be desirable in certain scenarios, but it is orthogonal to our goals in this work. Finally, work by Torres-Arias *et al.* [160] covers similar attack vectors where a malicious server tampers with Git metadata to trick users into performing unintended operations. These attacks have similar consequences to the ones presented in Section 2.4.

Security of VCS hosting services. In parallel to the VCS security issues, Git hosting providers face the same challenges as other Software-as-a-Service (SaaS) [165, 154] systems. NIST outlines the issues of key management on SaaS systems on

NISTIR-7956 [66], such as blind signatures when a remote system is performing operations on behalf of the user. This work is a specific instance of the challenges presented by NIST. Further work explores usable systems for key management and cryptographic services on such platforms. For example, work by Fahl *et al.* [78] presents a system that leverages Facebook for content delivery and key management for encrypted communications between its users. The motivation behind using Facebook and other works of this nature [116, 122] is the widespread adoption and the ease of usage for entry-level users. Based on similar motivation, this work seeks to bring Git commit signing to the web UI.

Countering Web and HTML-based Attacks. In addition to the effort in VSC and SaaS security, web UI issues are of particular interest. Substantial research was done in the field of web-based vulnerability detection that can target the web application's database (*e.g.*, SQL Injection) or another user (*e.g.*, Cross Site-Scripting). While automatic detection of these vectors is relevant to the overall security of our scheme, we assume that a repository may be malicious or impersonated (*e.g.*, via a MiTM attack). Additional work in this area, a direct motivation for Section 2.4.3, explores ways that a UI can use to force user behaviors [69]. While we do not consider phishing attacks to be part of the threat model (besides a possible pathway for a MiTM attack), research into the detection of phishing schemes could be used to identify and leverage compromised web UI's that trick users into performing unintended actions [50]. Specifically, we highlight the work by Kulkarni *et al.* [117] and Zhang *et al.* [170], which attempt to identify known-good versions of a web UI, and warn users of possible impersonations.

CHAPTER 3

ADDING VERIFIABILITY TO WEB-BASED CODE REVIEW SYSTEMS

3.1 Introduction

Code review is a crucial step for software development, aiming to find defects in the software and to improve software quality [142, 79]. With a code review system in place, new code changes will be integrated into the project's codebase only after being reviewed and approved by a number of reviewers. Over the years, a considerable amount of research [142, 133, 76, 137, 146, 151] provided evidence on the benefits of the code review, in particular to (1) catch and fix bugs in an early stage of the software development, (2) check the clarity of the code and improve maintainability, (3) comply with coding conventions, (4) mentor less-experienced developers, (5) share knowledge across the team, (6) accelerate the software development process.

The code review process is currently identified as the top practice in companies to improve code quality. Both industrial and open source communities have adopted a modern code review practice which is: (1) tool-based, (2) informal, (3) asynchronous, and (4) focused on reviewing code changes instead of the entire existing source code [137, 146]. Modern code review systems such as Gerrit [16], Collaborator [9], Crucible [11], ReviewBoard [43], and Helix Swarm [30] have become very popular as they facilitate the code review process by providing an interactive web UI to discuss and review the code changes. Gerrit, for example, is a highly extensible and configurable tool that allows developers to review and evaluate each other's code easily. It is used by Google for code review in open-source projects like Go, Chromium, and Android [6]. As of April 2021, GitHub hosts over 224 million repositories [98, 100] and is used by Microsoft to host thousands of projects and perform many development tasks [51, 121, 123]. It is also notable that just in 2019, 29% of companies are conducting tool-based reviews on a daily basis [151].

Despite providing a rich set of features, modern code review systems do not incorporate safeguards against tampering with the code review process. The lack of a secure code review step is the cause behind a significant percentage of attacks [126]. A compromised code review system can cause great damage [115]. Several important threats are associated with the code review system [153].

Code review systems are susceptible to attacks that can manipulate the review process without being detected. For example, consider a project in which a reviewer finds a security bug in a proposed code change and then gives the change a negative review score which should block it from being merged into the project's codebase. A malicious server, however, can hide or manipulate the negative review in order to make the change mergeable. As another example, a malicious server can bypass or tamper with the minimum number of approving reviews required by the review policy before a change can be merged.

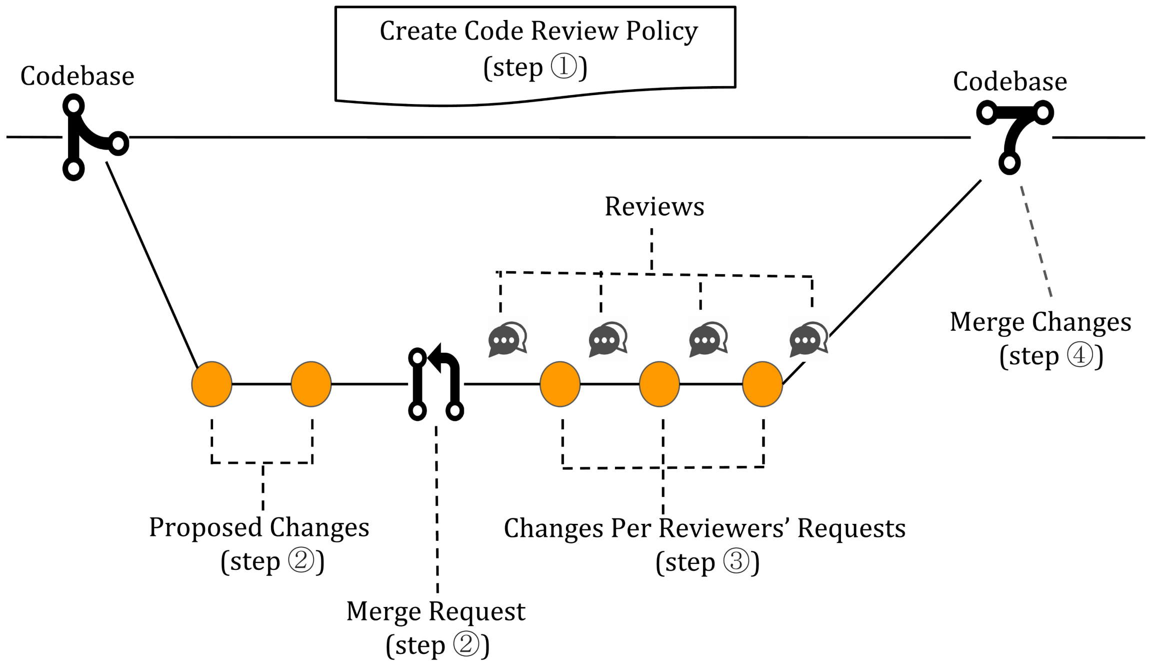
Such attacks are mainly possible due to two major shortcomings of code review systems: (1) There is no accessible record of the code review process as the review history is stored in a local database on the code review server. As such, neither the end user nor anyone else can verify after the code review step what occurred in that step. (2) There is no reliable code review history as the reviews and the review policy are not protected and can be tampered with. Signing code commits provides protection against tampering with the code but does not protect against manipulation of the code review process itself.

To improve this status quo, we start by identifying a set of key design principles necessary to secure the code review process. We then use these principles to propose `SecureReview`, a security mechanism that can be applied on top of a code review system in order to ensure the integrity of the code review process and provide verifiable guarantees that the code review process followed the intended review policy. We implement `SecureReview` as a client-side browser extension to help developers sign their reviews in the browser and include them as part of the source code repository. Although our extension works for the Chrome browser, it can be adapted to work with other browsers with minimal effort as the entire implementation relies on pure JavaScript. We also note that `SecureReview` is implemented on top of GitHub and Gerrit. Nevertheless, our design is general enough to be used on any web-based code review service that is integrable with a Git repository (such as GitLab [28], BitBucket [4], GerritHub [20], Crucible [11], and Phabricator [40]). None of these protect the code review process and are vulnerable to the same attacks.

Our goal in this work is not to improve the effectiveness of a code review system (*i.e.*, design better code review policy rules and best practices). Instead, we seek to lay the foundations for securing a given code review process. In other words, when a code review policy is in place, our goal is to ensure that the policy is actually respected, *i.e.*, to ensure the integrity of the code review process.

`SecureReview` can have an immediate positive impact on the security of the code review process, an area that has been largely overlooked and is becoming an appealing target as part of a growing trend of attacks against the software development chain [156, 10]. Specifically, we make the following contributions:

1. We identify attacks that tamper with the integrity of the code review process. The attacks are easy to execute, stealthy in nature, and can have significant impact. For instance, the attacks can lead to shipping a vulnerable or backdoored piece of code into the software product.
2. We identify a set of key design principles necessary to secure the code review process. We then apply these principles to design `SecureReview`, a mechanism that can be applied on top of a code review system in order to ensure the integrity of the code review process and provide verifiable guarantees about the code review process. We then show how to integrate our design into two popular code review systems, GitHub and Gerrit.
3. We implement `SecureReview` as a Chrome browser extension for GitHub and for Gerrit. Our solution features several advantages that can facilitate its practical



Create Code Review Policy (step ①)
Codebase
Codebase
Reviews
Merge Changes (step ④)
Proposed Changes (step ②)
Changes Per Reviewers' Requests (step ③)
Merge Request (step ②)

Figure 3.1 A typical code review workflow.

adoption: (1) it does not require any changes on the server side and can be used today, (2) it preserves typical code review workflows used in most popular code review systems and does not require the user to leave the browser, (3) commits generated by `SecureReview` can be easily verified by existing client tools (such as Git).

4. We analyze the security guarantees provided by `SecureReview` and show its effectiveness in defending against the aforementioned attacks. We also evaluate the efficiency of our implementation with a wide range of repository sizes and show that `SecureReview` adds only a slight overhead.

3.2 Background

This section provides background on the code review process. After a general overview, we look at two popular code review systems, GitHub [23] and Gerrit [16].

3.2.1 Code Review Process

Code review is common practice with the purpose of improving the quality, readability, and maintainability of the source code as well as the knowledge sharing [137]. Code review is usually done in a peer process in which new pieces of code are reviewed by developers other than the author of the code. Over the years, different approaches were used to review code, from offline code inspection meetings to an asynchronous tool-based code review process. Over the past decade, a modern code review process has been adopted by both open source and industrial communities [146, 142]. For instance, Microsoft, Google, Facebook, and VMware perform the review process using CodeFlow [7], Gerrit [16], Phabricator [40] and ReviewBoard [43], respectively.

After a code author submits a new code change, reviewers check differences between the proposed change and the codebase (i.e., the stable version of the source code). Reviewers may accept, reject, or ask for further changes. This process is repeated until either the reviewers are satisfied and the code change can be integrated into the codebase, or they reach the conclusion that the code change cannot be integrated into the codebase.

The "pull-based development model" is a specific form of modern code review, in which a developer forks a repository and makes changes in the fork. Then, she submits the changes as a `Merge Request`. Once the `Merge Request` is reviewed and accepted by reviewers, it is integrated into the codebase. Popular web-based code repository hosting services (such as GitHub [23], GitLab [28], Bitbucket [4]) and code review systems such as Gerrit [16] adhere to this model.

3.2.2 Code Review Workflow

As shown in Figure 3.1, a typical code review workflow has four steps:

- **Step ①**: A code review policy is created by the project owner. The policy defines the rules that govern the code review process and the conditions that must be satisfied before proposed changes can be integrated into the codebase. For example, a minimum number of positive reviews may be required before merging new changes.

- **Step ②**: Developers who want to modify the codebase propose changes and request to merge the proposed changes into the codebase. We refer to this request as a "merge request" (though different code management systems have specific names for it: "pull request" in GitHub, "merge request" in GitLab and "change" in Gerrit). A merge request is then created, and one or more reviewers are assigned to review the proposed changes.
- **Step ③**: After reviewing the proposed changes, reviewers provide feedback, either positive (*e.g.*, approve changes) or negative (*e.g.*, request new modifications). In the latter case, developers propose new changes to address the reviewers' concerns The review-change cycle continues until reviewers are satisfied with and approve the changes.
- **Step ④**: When the merge policy is satisfied, the approved change is merged to the codebase by an authorized user. (*e.g.*, by the project owner).

3.2.3 GitHub

GitHub, the most popular web-based hosting service for open source projects, features integrated code review capabilities for those who want to collaboratively develop code. A merge request, referred to as "pull request", allows developers to ask for code review and to receive feedback about their proposed code changes before those changes can be integrated into the codebase. This is a highly popular feature: In 2019 alone, developers made over 87 million pull requests on GitHub [100], which represents a growth of 28% compared to 2018. We provide next an overview of the GitHub's code review process.

3.2.3.1 GitHub Permissions. GitHub defines several permission levels for contributing to a repository, but four are relevant for the code review process[1]: *read* (can read code and provide code reviews), *write* (can read/write code and provide code reviews), *maintain* (can do most actions related to repositories, including read/write of code, provide code reviews, and merge pull requests; project managers usually have

[1]We focus on *organization repositories*, which are typical for collaborative projects. However, our work can also cover *user account repositories*, which have a more limited set of permissions.

this permission) and *admin* (full access, including changing configuration and security settings, and change user permissions; the project owner has *admin* permissions).

3.2.3.2 Code Review Policy. The owner of a GitHub project can define a code review policy which describes, on a per branch basis, the rules pertaining to the code review process for changes to that branch. By default, code review is disabled, but can be enabled from a configuration option called “Branch Protection Rules” [97]. The remainder of this section refers to the case when code review is enabled.

With each review, a reviewer provides a rating for the proposed changes and text feedback. The rating is mandatory and can be one of three values: *Approve* (reviewer approves merging the proposed changes), *Request changes* (reviewer’s feedback must be addressed before the changes can be merged), and *Comment* (general feedback without explicitly approving the changes or requesting additional changes). The text feedback consists of comments about the proposed changes and is optional if the *Approve* rating is chosen.

One of the most common code review policy rules defines the required number of approving reviews before changes can be merged[2]. When this number is met, then the changes can be merged. Although anyone with *read* access to the repository can submit a review, only approving reviews from reviewers with *write* permissions count towards the required number of approving reviews. It is notable that even though developers may review their own pull requests, they can only leave comments and, therefore, their reviews are not counted towards satisfying the required number of approving reviews. Finally, we note that if a person with *write* or *admin* permissions makes a *Request changes* review, then that person must later give an approving review before the changes can be merged.

The review policy may contain additional optional rules. One such rule is to dismiss existing approving reviews when a code-modifying commit is pushed to the

[2]By default, the number of required approving reviews is set to 1.

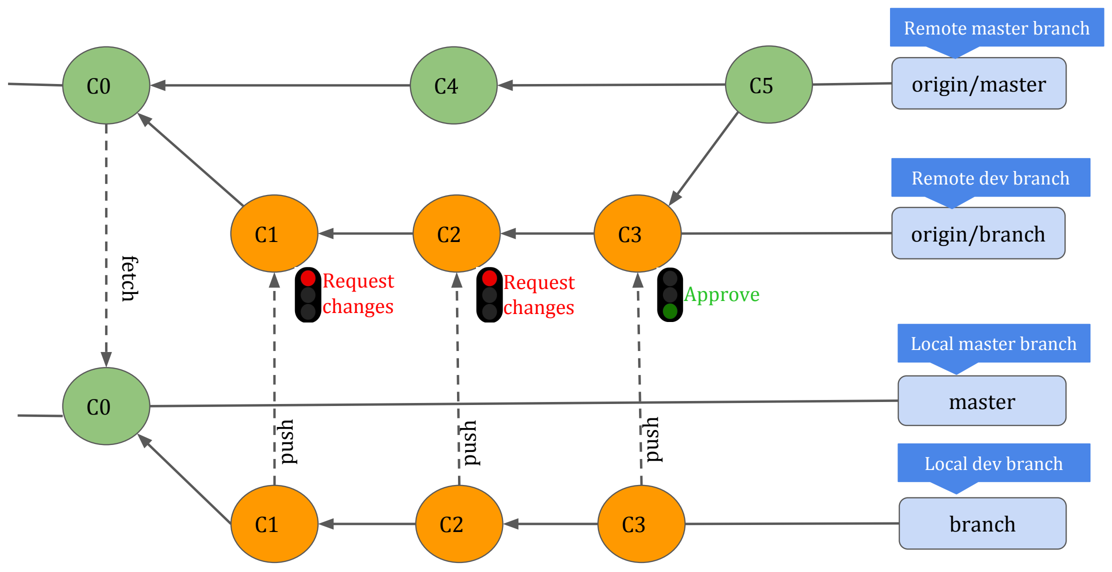


Figure 3.2 The typical lifecycle of a pull request on GitHub.

pull request branch. In other words, the code review process is reset and any existing approving reviews before this new commit will not be counted towards satisfying the required number of approving reviews.

The review policy may have a rule that requires approving reviews from "code owners", which is a set of designated individuals that are responsible for code in a repository. In this case, the required number of approving reviews must be from these specific individuals. Finally, we note that the project owner can bypass the code review policy rules, based on her *admin* permission. For instance, the project owner can approve her own pull requests, and can merge a pull request even though there are not enough approving reviews.

3.2.3.3 Create and Merge Pull Requests. In a typical workflow for a pull request on GitHub, the developer first clones the repository. Then she updates the code locally and submits the new code change to the GitHub repository. Next, the developer creates a pull request which will be reviewed by a number of reviewers. If reviewers request for changes or the developer herself decides to submit additional changes, she can update the pull request by submitting new commits. Finally, when the pull request is approved (per the code review policy), it can be merged into the codebase.

Although initially only the project owner has the ability to merge a pull request (based on her *admin* and permission), she can extend the *admin* or *maintain* permissions to other trusted users in order to manage day-to-day operations such as merging pull requests.

For example, consider the pull request shown in Figure 3.2. The developer creates a local feature branch ("dev") in which she places her changes and creates a new commit C1. Receiving feedback from reviewers, she improves her pull request and submits two new commits (C2 and C3). When C3 is approved by reviewers, the pull request is merged into the base branch ("master"), which results in commit C5.

3.2.4 Gerrit

Gerrit, a popular code review system used in big open-source projects like Go, Chromium, and Android [6], is a highly configurable tool that was designed specifically to support the code review process. Unlike GitHub andother popular web-based hosting services for open source projects, Gerrit is designed specifically to support the code review process. Essentially, Gerrit is a Git server that provides a web UI for the code review process. A merge request, referred to as a "change", allows code changes to be reviewed before being integrated into the codebase.

As shown in Figure 3.3, A Gerrit server manages the source code using two locations: an "authoritative repository" which contains the stable version of the codebase and a "pending changes" location which is a staging area for new code changes. According to the Gerrit workflow, developers fetch code from the authoritative repository and push their new changes to the staging area. The proposed

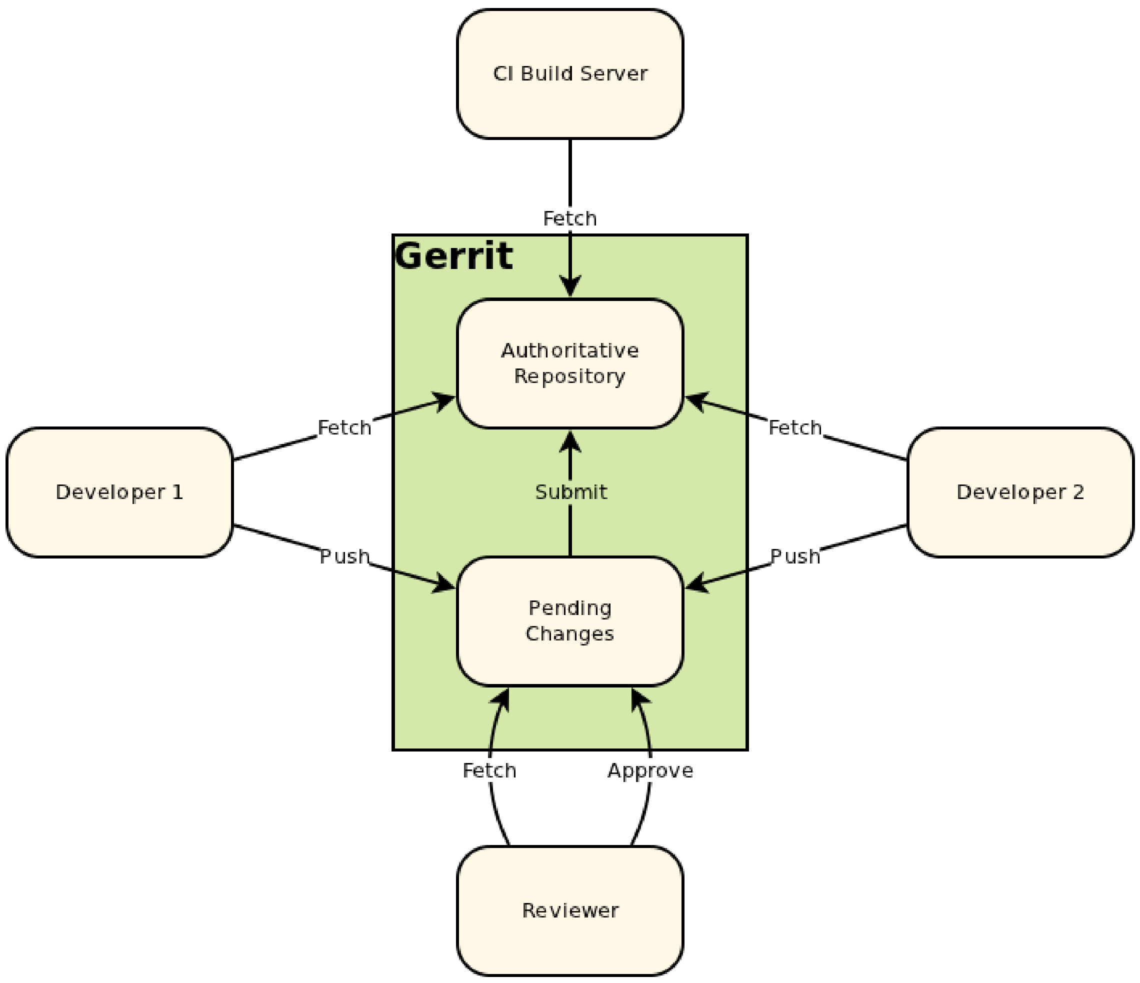


Figure 3.3 Gerrit workflow [32].

changes are being reviewed, possibly updated, and eventually are getting merged (*i.e.*, submitted) into the authoritative repository.

Gerrit users' activity is centered around two types of actions. First, they Gerrit users can perform reviews on a change. All review information is stored by Gerrit in a dedicated database, separated from the underlying Git source code repository. Second, Gerrit users can update the source code in the change (*i.e.*, adding, modifying, or deleting files). Any modification to a Gerrit change is referred to as a "patch set" and results in a new Git commit object. We note that users can update the commit message of the change through the web UI, which also results in a new patch set being added to the change.

3.2.4.1 Gerrit Permissions.

Compared to GitHub, Gerrit has a more complex framework for defining permissions. Access rights are defined based on groups. Every user is a member of one or more groups, and the users' access rights are inherited from the groups to which they belong. The following access rights are the most relevant for the code review process and can be defined on a per branch basis:

- *Read*: allows to read any data of a project, including any proposed code changes.
- *Code-Review[-2 .. +2]*: allows to submit a code review with a rating score that can range between -2 and +2.
- *Upload to Code Review*: allows to create a new change for code review. We also refer to this as a "Create Change" access right.
- *Add Patch Set*: allows to upload a new patch set to existing changes.
- *Submit*: allows to merge a change into the destination branch. In addition to needing this access right, a user can merge a change only if the change also satisfies the existing code review policy.
- *Rebase*: allows to rebase changes via the web UI.
- *Owner*: allows to modify a project's configuration, which includes granting/revoking any access rights.

3.2.4.2 Gerrit User Groups.

Gerrit comes predefined with the following user groups which are relevant for the code review process:

- *Registered Users* are signed-in users who have the *Code-Review[-1 .. +1]* permission, meaning they can provide feedback on a change (score between -1 and +1), but cannot cause it to become approved (which requires score +2) or rejected (which requires score -2). Users in this group also have *Read*, *Create Change*, and *Add Patch Set* permissions.
- *Change Owners* are users who have created a change. They have the *Code-Review[-1 .. +1]* permission for that change, meaning they can review and rate the change but cannot cause it to become approved or rejected. By default, users from this group also have the following permissions: *Read*, *Add Patch Set*, *Rebase*, and *Submit*. We note that a Change Owner will be able to submit their change only when it meets the rules of the code review policy (i.e., it gets approved by other reviewers).

- *Project Owners* are users who have the *Owner* permission and as such can grant/revoke themselves (or others) any access rights. For example, they can change the permissions of the default groups (*e.g.*, allowing *Registered Users* to block a change). By default, users from this group also have all the permissions that are relevant for code reviewing (*Read*, *Code-Review[-2 .. +2]*, *Create Change*, *Add Patch Set*, *Submit*, and *Rebase*). Many Gerrit instances use the name *Mantainers* for this group.

- *Administrators* are the most powerful users in Gerrit. In addition to all permissions of *Project Owners*, users in this group also have capabilities needed to administer a Gerrit instance and typically have direct filesystem and database access. It is noteworthy that, in a typical Gerrit's workflow, *Administrators* are not involved in the code review process.

In addition to these predefined groups, many Gerrit instances also commonly define the *Developers* group. This group is created by Project Owners, who can define new custom groups. Users in this group are typically core developers, who have all the permissions of *Registered Users*, plus the ability to *Code-Review[-2 .. +2]* and *Submit* changes.

3.2.4.3 Code Review Policy. The main component of the Gerrit code review policy is the *submit rule*. The submit rule is a logic that evaluates code changes through a rating process to determine if the proposed changes can be merged into the codebase. Each code review consists of a rating (*i.e.*, a score) and text feedback. The text feedback is optional and consists of comments about the proposed changes. The rating is mandatory and is an integer that ranges from -2 to +2. The lowest rating (-2) indicates that the proposed change cannot be merged unless the viewer's feedback is addressed. The highest rating (+2) means that the reviewer approves the change. Other scores (*i.e*, -1, 0, +1) indicate that the reviewer prefers other reviewers to make the final decision on rejecting or approving the change.

The default submit rule in Gerrit is that a change can be merged if it has received at least a review with the highest score (+2) and no reviews have the lowest score (-2).

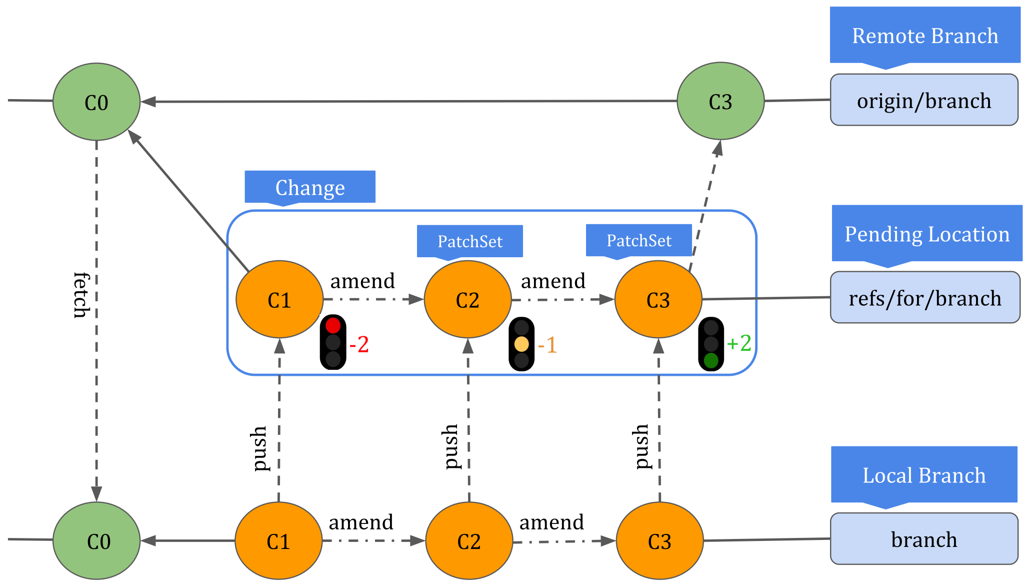


Figure 3.4 The typical lifecycle of a code change in Gerrit under the default "Merge If Necessary" strategy.

3.2.4.4 Create Changes. To create a change on Gerrit, developers upload a new commit to the Gerrit server which is automatically sent to the `pending changes` location to be reviewed. During the course of the review process, reviewers discuss the change and might ask for improvements.

Any modification to an existing change is called a "patch set". At the repository level, a change is always represented by a single commit object. When a patch set is added, this commit object is amended (*i.e.*, a developer amends the previous commit using "`git commit --amend`" and then pushes it to the `pending changes` location). Amending a commit in Git means that the commit is being replaced by a new commit and will not be visible anymore in the repository history. The new commit applies the modifications in the patch set on top of the previous commit; it may differ from the previous commit in either or both the commit message or the repository files. Thus, the latest patch set is equivalent to the entire change, as it contains all the code modifications introduced by this change.

We note that patch sets are normally created by developers to either update the code or the commit message. However, anybody (including the reviewer) who has the *Add Patch Set* permission can create a new patch set.

3.2.4.5 Merge Changes. When enough reviewers approve the change, the last patch set will be merged into the `authoritative` repository. For example, consider the change shown in Figure 3.4. A developer creates the change by sending commit C1 to the Gerrit server. Reviewers ask for improvements twice and, the developer submits two amended commits (C2 and C3) to get the change approved.

Different strategies may be employed for merging a change into the codebase. In Gerrit terminology, the merge strategy is referred to as the “submit type”. Gerrit supports six submit types, such as *Fast Forward Only* (the head of the codebase repository is fast-forwarded to the change commit), and *Merge Always* (equivalent to “`git merge --no-ff`” command that always results in a new merge commit object). The default submit type in Gerrit is *Merge If Necessary*, which means Gerrit attempts a fast-forward strategy if possible, (*i.e.*, no commits were submitted to the codebase branch after the change was created), otherwise a merge commit is created.

Consider the repository shown in Figure 3.4, in which a developer forks the `authoritative` repository with an initial commit C0. She works on her local branch and submits commit C1 to the server. Gerrit creates a new change and puts C1 in the staging area to be reviewed. The reviewer gives a -2 score and asks for modifications in the code. As a result, the developer updates the code and creates commit C2 by amending the previous commit, C1. The reviewer looks into commit C2, is still not satisfied with the change, and requests further modifications. The developer then modifies the code and submits commit C3 which is given a +2 score by the reviewer. At this point, the change is approved for merging into the `authoritative` repository. Finally, the project owner merges the change by fast-forwarding the head of the codebase repository to the change commit.

3.2.4.6 Changing Defaults. We note that the project owner may change most default review policies through the Gerrit web UI. For example, the project owner can create new groups or can change the permissions of existing groups.

The owner can also change other default global project settings, such as the review workflow, the *submit rule* and the *submit type*. These settings are defined in three files, `project.config`, `groups`, and `rules.pl`, located under the `refs/meta/config` branch. The owner can change these settings based on her ability to write to this branch. However, we note that customizing the project settings cannot be done via Gerrit's web UI and must be done instead through the command line using the Prolog logic programming language [17].

3.3 Threat Model

We assume a threat model in which the attacker seeks to violate the integrity of the code review process in a web-based code review system. This can have severe negative consequences for the code repository, such as merging into the production codebase code that has not been properly vetted and contains vulnerabilities. Such dangerous code may consist for example of experimental features that were rejected, debugging code, or code in which security features were removed intentionally for testing purposes.

To tamper with the integrity of the code review process, the attacker has two main avenues: Manipulate the code review policy and/or the steps of the actual code review process (*i.e.*, the individual code reviews and their sequence). The former attack can be carried out through a variety of approaches. A malicious server may deem a code change mergeable even though the minimum number of approving reviews is not met. It can also add unauthorized users to the code review process or disable some rules (*e.g.*, mandatory reviews from specific users). Moreover, an attacker may manipulate the code review policy by counting outdated reviews or changes from unauthorized users. The latter attack can be executed by modifying or removing existing code reviews (*e.g.*, removing a review with a low score, or changing the score in a review), or by adding illegitimate code reviews. Specific attacks are described in Section 3.4.

We assume the attacker is able to tamper with the repository (*e.g.*, modify data stored on the Git repository) and can modify the internal database that stores the code review information. The attacker can also tamper with the configuration files that define the code review policy and other important settings that can affect the code review process. Finally, the attacker may manipulate the information displayed to users, for example, by hiding some of the reviews and making a change look like it is ready to be merged when in fact it is not. This scenario may happen either directly (*e.g.*, a compromised or malicious code review server), or indirectly (*e.g.*, through MITM attacks, such as government attacks against GitHub [105, 120])

We assume that all Git commits are signed by the clients who create the commits and digital signatures provide adequate security (*i.e.*, the attacker cannot compromise a digital signature scheme). The ability to sign commits on the client side is available either through Git's command line tool or by using a tool such as le-git-imate [55] for commits created using a web-based UI such as GitHub/GitLab. In practice, this assumption is supported by the fact that most code review systems allow a rule in the code review policy to require that all commits need to be signed. Even though the attacker can write to the Git repository, signed commits combined with Git's commit hash chaining mechanism greatly limit the attacker's ability to arbitrarily tamper with the repository. Removing an existing commit from the end of the commit chain, or entirely discarding a commit submitted via the web UI are actions that have a high probability of being noticed by developers. Otherwise, our solutions cannot detect such attacks, and a more comprehensive solution should be used, such as a reference state log [160].

We focus on attacks that tamper with the integrity of the code review process (specific attacks are described in Section 3.4). For general attacks that tamper with commits performed by the user via the web UI, we refer the reader to a comprehensive list of attacks and defenses [55].

The source code repository associated with the code review system can be updated by developers. We assume that there are trustworthy individuals in the system, such as the developers who propose code changes and the users who are authorized to merge changes (*e.g.*, the project owner or a maintainer). The attacker does not have the signing key of these trusted individuals. As such, the attacker cannot tamper with the signed commits.

In addition, we assume that reviewers are not fully trusted and they may attempt to bypass the code review policies, as long as they do not self incriminate. The code review policy is calibrated so that the minimum number of approving reviews reflects the trustworthiness of the reviewers (i.e., there are enough honest reviewers to avoid a situation where reviewers give misleading scores in order to merge dangerous code).

Finally, we assume that addressing the following problems is outside the scope of this dissertation:

- A reviewer who always approves a merge request regardless of the quality of the proposed change. In other words, we do not address human carelessness.
- A weakness or bug that is not caught by a regular code review system. Our goal is not to improve the code review system's ability to catch subtle code bugs/vulnerabilities.
- A flaw in the code review policy that allows dangerous code to become part of the codebase. For example, consider a review policy that requires three approving reviews for code changes, but does not dismiss existing code reviews upon receiving a new code update. Assume that a merge request gets approved by two honest reviewers, and then the third reviewer maliciously injects a

backdoor to the merge request as well as approves it. This results in merging dangerous code to the codebase without violating the code review policy. This attack could be avoided if the review policy enforces all reviewers to re-review the latest version of the code that is about to be merged. We do not seek to detect such review policy flaws.

3.3.1 Security Guarantees

Answering to this threat model, the goal of a secure code review system should be to enforce the following:

- **SG1: Prevent unauthorized modification of stored code reviews and code review policy.** An attacker should not be able to modify stored code review information or the code review policy without being detected.
- **SG2: Ensure the integrity of the code review process.** The code reviews (including their content and sequence) should match what the code review policy prescribes. In other words, no code change should be deemed as mergeable if the corresponding sequence of code reviews does not satisfy the intended code review policy.
- **SG3: Ensure verifiability of the code review process.** The code review process should be verifiable. This would allow auditors to verify the correctness of the code review process even after the code review phase.

3.4 Attacks

In this section, we identify new attacks against the code review process. Common to these attacks is the fact that an unscrupulous code review server can manipulate arbitrarily code review metadata. Two main attack avenues can be used to manipulate the code review process: (1) Manipulate the steps of the actual code review process (*i.e.*, the individual code reviews and their sequence); (2) Manipulate the code review policy. The goal of these attacks is twofold:

- **AG1: Cause the merging of a code change that does not satisfy the intended code review policy.** For example, the code review server can prevent code defects from being discovered during the code review process and, therefore, facilitate the merging of dangerous code into the codebase.

- **AG2: Prevent or delay the merging of a code change even if it satisfies the code review policy.** For example, the server can stop a security patch from being deployed.

Whereas a scenario involving AG2 has a higher chance of being detected, we note it as a concern nonetheless.

The attacks described in this section are easy to execute, as the server simply has to manipulate data in the code review database and/or configuration files describing the code review policy. Nevertheless, the attacks' impact can be significant. Many of the attacks are stealthy in nature due to the lack of protections for code review metadata and also because subtle violations of the code review policy are difficult to detect.

3.4.1 Code Review Manipulation Attacks

An attacker can achieve the aforementioned goals by modifying, deleting, or adding to the existing code reviews. To achieve AG1, the attacker may decide to improve the rating of existing code reviews, or delete negative reviews, or add new illegitimate positive reviews. To achieve AG2, the server may lower the rating of existing reviews, remove positive reviews, or add new negative reviews.

For example, assume that the scenario presented in Figure 3.4 uses a review policy in which a change is mergeable if there is at least one review with highest score (+2). Under benign circumstances, the code is being reviewed three times and reviewers are satisfied with the change after two new patch sets are applied (*i.e.*, when commit C3 is submitted). However, the server can alter the reviews as shown in Figure 3.5 to prevent the code from being carefully reviewed. In this example, the second review score is improved by 1 (from +1 to +2), thus making the code mergeable – only changes introduced in C2 are merged. The impact of this attack can be severe. If C2 contained a security vulnerability, which was fixed by the developer in C3, the fix will be omitted from the project's codebase.

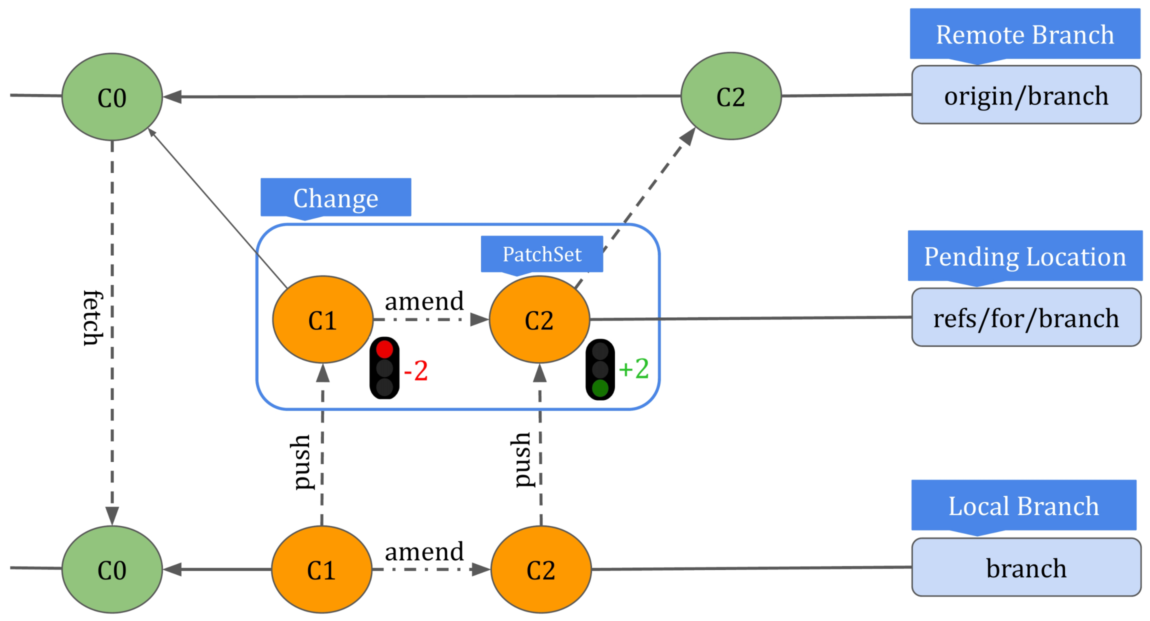


Figure 3.5 Review manipulation attack.

3.4.2 Code Review Policy Manipulation Attacks

When the code review server determines the review policy is satisfied for a change, it notifies the merger to proceed with the merging operation (*e.g.*, by rendering a green "Merge" button on the GitHub pull request page or a blue "Submit" button on the Gerrit change page). The assumption is that the server correctly assessed that the review policy is satisfied based on the existing reviews. However, an unscrupulous server may automatically merge the code change or mislead the merger to merge the change even if the review policy has not been satisfied.

One way the server can execute a code review manipulation attack is to tamper with the code review policy (*i.e.*, the rules defining the policy or the configuration files that are relevant for the policy). This may lead to merging of dangerous code, for example, if the minimum number of approving reviews is reduced.

Another way to execute a code review manipulation attack is to falsely declare that a change is mergeable even when it does not satisfy the review policy. The server counts that such an attack will go unnoticed based on two facts: (1) The merger will

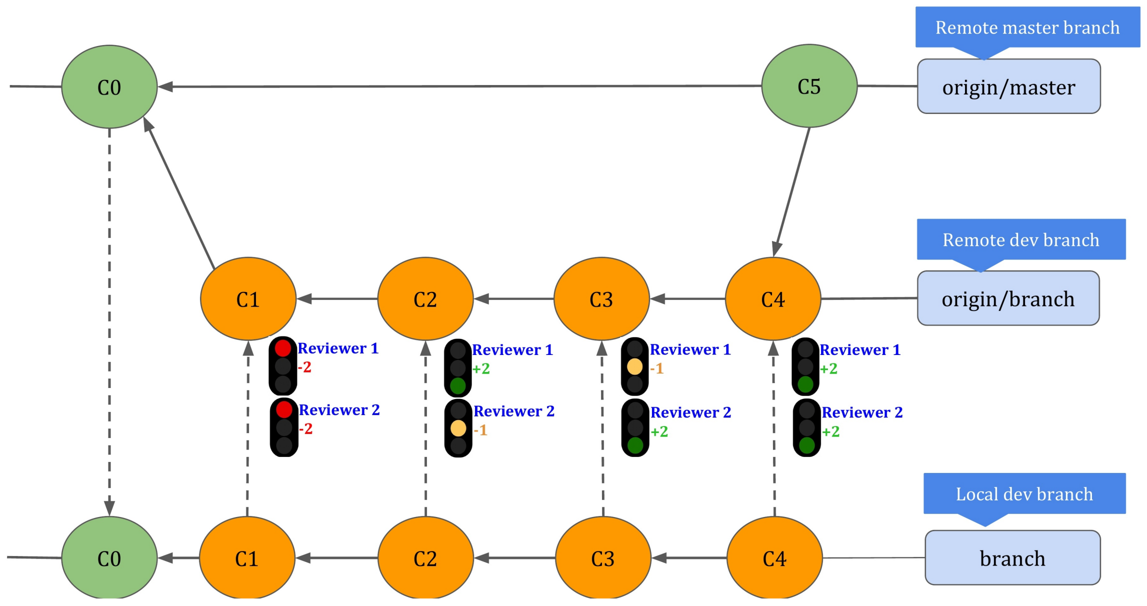


Figure 3.6 A core review example in which the pull request is merged to the master branch after receiving two scores of +2, and there is no -2 score.

not notice that the review policy is not satisfied, especially when a small detail is not respected. For example, if the review policy requires at least two approving reviews, the merger may not notice if out of two approving reviews, one review is performed by a user who is not authorized to provide reviews; (2) After the change is merged, there is no record of the reviews in the repository, so an auditor cloning the repository cannot verify the correctness of the code review process. Even when the auditor has access to the code review server which allows access to the code reviews database, there is no mechanism for independent validation of the code review process.

We introduce next a variety of code review policy manipulation attacks that can be used by a malicious server to achieve AG1 and AG2.

3.4.2.1 Bypass a Minimum Number of Approving Reviews. A popular code review policy rule is that a code change can be merged if it receives a minimum number of approving reviews. A malicious server can bypass this rule by either changing the minimum number of approvals or by completely ignoring this rule. For

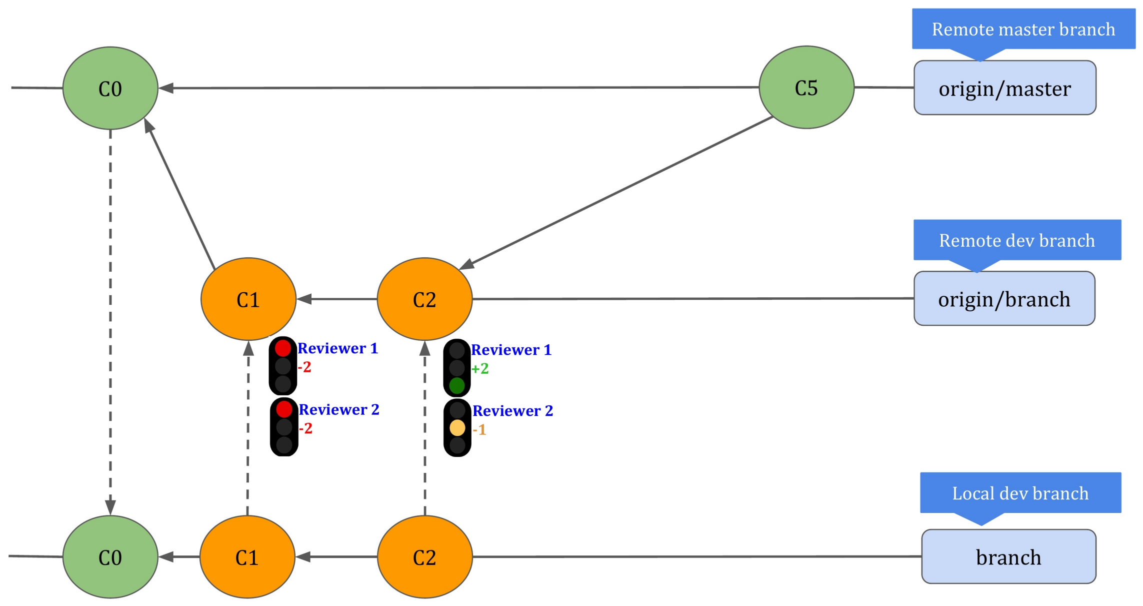


Figure 3.7 Code review policy manipulation attack. A malicious server changes the minimum number of approving reviews.

example, consider a GitHub code review policy that require that at least 3 reviewers provide approving reviews. The server can tamper with the review process by declaring a change is mergeable after only 2 approving reviews. As a result, a change that has not received enough scrutiny and may still contain security vulnerabilities will be merged.

For example, consider Figure 3.6 in which a change is submitted to be reviewed. Assume the review policy is as follows: "A new change is mergeable if it is approved by at least two reviewers which means receiving a score of +2 from each." Under benign circumstances, the pull request is updated tree times and each commit is reviewed by two reviewers. After submitting commit C4, both reviewers are satisfied with pull requests and allow it to be merged to the master branch. However, a malicious server can temper the review process by reducing the minimum required number of approving reviews as shown in Figure 3.7. Consequently, the pull request becomes mergeable without incorporating C3 and C4. Because by reviewing C2, the pull request has enough approvals and there is no -2 score to block the merge. The impact of this attack could be severe if C2 contained a security vulnerability that was fixed in C3 and C4. In other words, security patch sets were omitted from the codebase by manipulating the review scores.

3.4.2.2 Count Reviews from Unauthorized Reviewers. A server can upgrade or downgrade the reviewers' permissions in order to achieve AG1 and AG2, respectively. The server can make it look like a user who submitted an approving review has write permissions, when in fact the user has only read permissions and the review should not be counted towards the minimum required number of approvals. The server may also count reviews from the author of the code change. Most code review system allow receiving reviews from the user who owns code changes, however, their approval does not count toward the minimum number of approvals. A malicious server can ignore this policy and the attack can go undetected easily. Finally, the malicious server can add an unauthorized reviewer to the code review process.

For instance, consider Figure 3.6, and the code review policy described in Section 3.4.2.1. As shown in Figure 3.8, by adding an unauthorized reviewer the malicious server can merge the code changes without incorporating C3 and C4.

These attacks are simple to execute, either by manipulating the configuration files that define user permissions, or by temporarily misrepresenting the user permissions when the merger is merging the change. In all these cases, the merger can be deceived to prematurely merge a change that has not received enough reviewing scrutiny. Similarly, an auditor who wants to later verify if the code review policy was correctly enforced, has no way to reliably do so.

3.4.2.3 Exclude Required Reviews from Specific Users. Code review systems may enforce a rule which requires that code changes be reviewed by specific users. GitHub, for example, allows requiring approving reviews from users in a

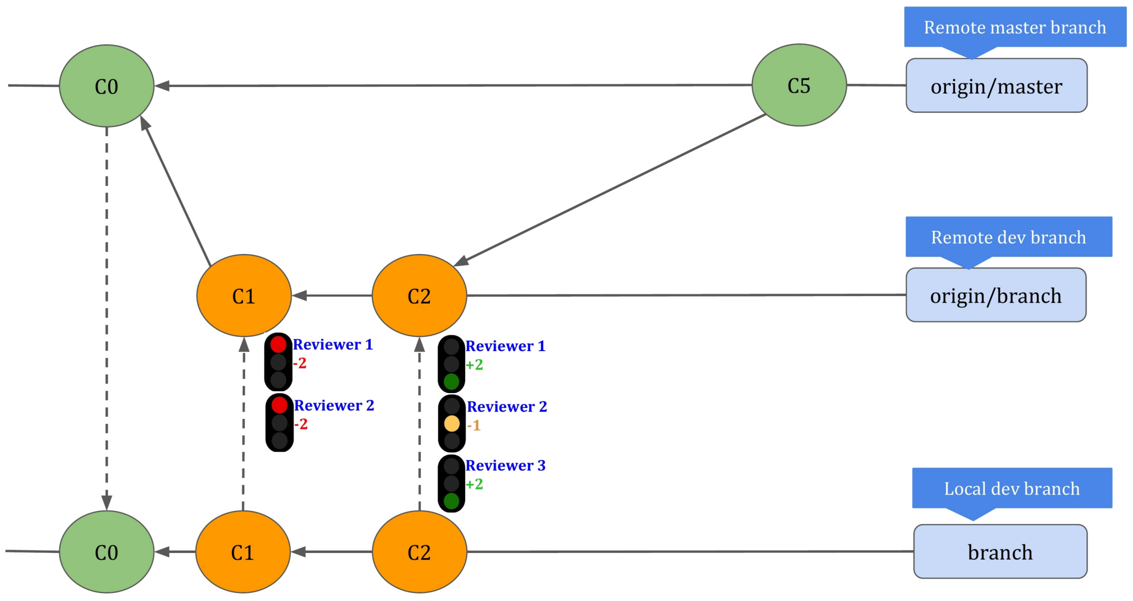


Figure 3.8 Code review policy manipulation attack. A malicious server adds an unauthorized reviewer to the code review process.

group called "code owners". Gerrit also allows defining a rule called "Master and Apprentice", according to which all code changes introduced by a user (*i.e.*, Apprentice) must be approved by another user called Master.

However, a malicious server may deem a code change mergeable even though there are not enough approving reviews from specific users. The server can also manipulate the composition of this group of users from which reviews are required. The attack can not be detected easily since neither the merger (before merging) nor a verifier (later, after merging) cannot reliably verify if the server enforced the policy correctly.

3.4.2.4 Count Outdated Reviews. One common practice is to reset the code review process when a code-modifying commit (*i.e.*, a patch set) is pushed in a change. That means existing approving reviews are dismissed and should not be counted towards the require number of approving reviews. This policy helps to catch bugs

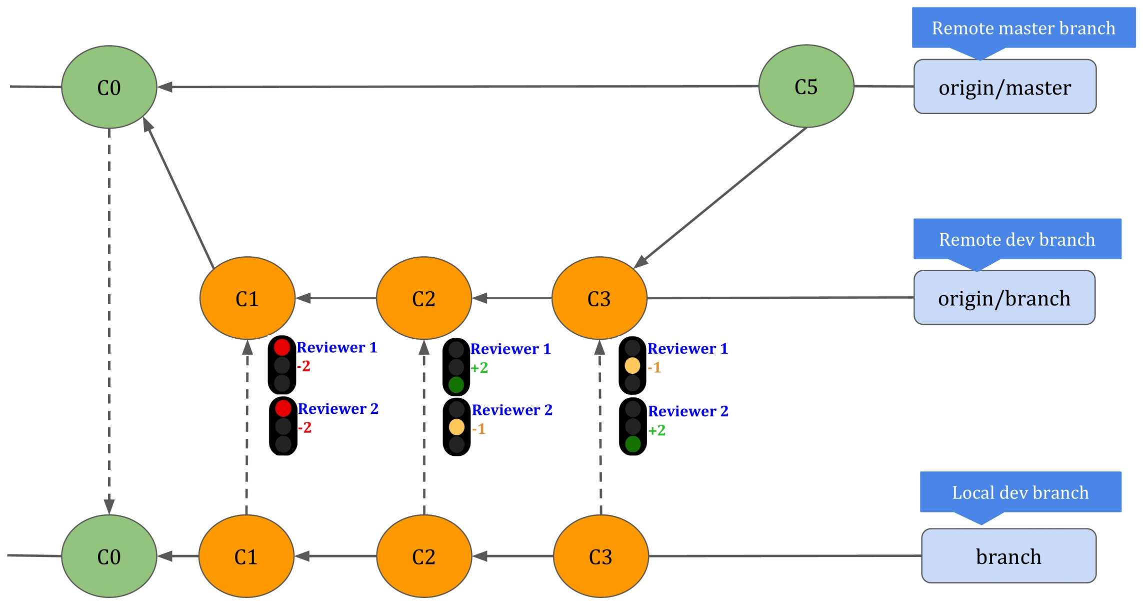


Figure 3.9 Code review policy manipulation attack. A malicious server disables the policy of dismissing stale pull request approvals.

introduced by a patch set. A malicious server, however, may disable/ignore this policy to take advantage of stale approving reviews and deem a change as mergeable with insufficient review scrutiny.

Consider Figure 3.6, and the code review policy described in Section 3.4.2.1, except when new changes are pushed, previous reviews are dismissed. If the server ignores this policy, the pull request becomes mergeable without addressing comments by reviewer 1 and also without incorporating C4 as shown in Figure 3.9.

3.4.2.5 Merge Changes without Addressing Comments. A code review system may enforce addressing all comments before allowing the code change to be part of the codebase. GitHub by default does not allow to merge a change unless requests for changes are addressed. Gerrit also allows defining such policy – make a change submittable if all comments have been resolved. Thus, if any reviewer rejects a code change, it cannot be merged unless the same reviewer approves it. A malicious server, however, can bypass this policy to prevent discovered code defects from being patched.

To illustrate this attack, consider Figure 3.6 in which the review policy is as follows: “A new change is mergeable if it is approved by at least one reviewer and all comments are addressed (there are no negative scores).” Under benign circumstances, the pull request is merged when commit C4 is approved. However, a malicious server can make the code change mergeable by ignoring the fact that all comments must be addressed. Consequently, the pull request becomes mergeable without incorporating C3 and C4. Because once the first reviewer approves C2, the pull request has enough approvals though comments by the second reviewer is not addressed (as shown in Figure 3.7).

3.4.2.6 Misuse the Project Owner’s Authority. In most popular code review systems, the project owner has the ability to bypass the code review policy rules – this includes merging code changes even if the review policy is not satisfied. A malicious server can take advantage of this authority to merge vulnerable code changes. The impact of this attack could be severe since during the verification step, the auditor has no way to verify if the project owner made the merge or a malicious server impersonates the project owner’s role.

3.4.2.7 Accept Changes from Unauthorized Users. A code review policy rule may enforce accepting changes only from a specific group of authors. For example, Gerrit allows setting a rule making a code change submittable if it has a specific commit author. GitHub also allows to define a similar policy – specify users that are not allowed to push to matching branches. This rule prevents unintentionally accepting code changes from inexperienced developers even though their code changes get approved by reviewers. A malicious server may ignore this policy.

3.4.2.8 Modify Project Settings. A malicious server can modify different project settings to achieve AG1 and AG2. The server can manipulate user permissions, for example by modifying project configuration files that control access rights and composition of user groups in Gerrit. Consequently, unauthorized users will be able to participate in the review process. In a similar scenario, the server may modify the rating score associated with a group of users. That could result in approving or rejecting a code changes maliciously. Moreover, an unscrupulous GitHub server may alter the gitattribute file to keep certain files out of the pull request's diff. This can hide a piece of dangerous code from being reviewed by reviewers. Another malicious change in the configuration is disabling the required status checks. That could prevent certain CI tests (*e.g.*, vulnerability scans) before merging a code change to the codebase. Finally, the attacker may take advantage of the fact that some code review systems such as Gerrit allow users to define new code review policies. For instance, the default policy in Gerrit is that a change can be merged if it has received at least a review with the highest score and has no review with the lowest score. A malicious server may override this policy by adding a new rule saying a code change is mergeable if it has received a review from a user with special authorization.

3.5 Solution

In this section, we introduce `SecureReview`, a mechanism that can be applied on top of a code review system in order to ensure the integrity of the code review process and provide verifiable guarantees about the code review process. We refer to a code review system where `SecureReview` was deployed, as a *secure code review system.* We first put forth a set of design principles and then apply them to design the major components of a secure code review system.

3.5.1 Design Principles

Popular code review systems such as GitHub and Gerrit do not provide verifiable guarantees about the integrity of the review process due to the following shortcomings:

- **No accessible record of the code review process**: Code review systems store all code review metadata in an internal database that is not tightly connected to the source code. Such systems provide no or little access to the code review information in subsequent steps of the development chain (*e.g.*, build). As such, neither the end user nor anyone else can verify after the code review step what occurred in that step. For instance, in services like GitHub that allow managing the entire software lifecycle, only an authorized user can access the code reviews. In services like Gerrit that are dedicated to the code review process, an authorized user should go through different systems (packaging, code versioning) to access the code reviews.
- **No reliable code review history**: Even if there was an automated way to extract the code review history, it is not possible to validate if the review data matches what the review policy prescribes. Indeed, current code review systems do not provide an option to sign the reviews or the review policy nor web UI commits. Instead, they assume that the reviewers and the code review server are trusted not to tamper with the review process. Unfortunately, as described in Section 3.4, code review systems are susceptible to attacks that can manipulate the review process without being detected. `SecureReview` addresses this issue by allowing independent auditors to verify the integrity of the code review process.

To address the aforementioned shortcomings, we identify a set of design goals that should be satisfied by any solution that seeks to add integrity to a code review system.

- **DP1: Ability to access the code review history**: The code review history should be accessible for later auditing of the code review process relative to the source code repository. Code review information can be available either in partial (*e.g.*, only a summary, such as the rating), or in full form.
- **DP2: Verifiability of the entire code review process**: The solution should make it possible to verify the integrity of the code review process relative to the code review policy.
- **DP3: Least attack surface**: The solution should minimize the number of trusted entities.

- **DP4: Ease of adoption and deployment**: For ease of adoption and to ensure that it can be deployed immediately on top of existing systems, the solution should require no (or minimal) server-side changes.
- **DP5: Ease of use**: The solution should preserve as much as possible the current workflow of web-based code review systems. In particular, it should preserve the ease of use of the system's web UI and should not introduce unnecessary complexity to these systems, as this may hurt usability.
- **DP6: Minimum impact on the user's experience**: The solution should not require the user to leave the browser. This will minimize the impact on the user's current experience with using code review systems.

3.5.2 A Strawman Solution

Current code review systems do not have mechanisms to ensure the integrity of the code review policy. As a result, the integrity of the code review process can be violated by simply tampering with this policy.

It may be tempting to assume that protecting the integrity of the code review policy (*e.g.*, by having the project owner digitally sign it) will ensure the integrity of the entire code review process. Whereas doing so is a necessary step, we argue that is it not sufficient. Indeed, this approach can mitigate only a subset of the attacks presented in Section 3.4 (only attacks that manipulate directly the policy rules, but not attacks that simply ignore the policy or that manipulate the code reviews). Also, only signing the main rules of the review policy is not enough, because leaving other configuration information unprotected may lead to code review integrity violations. Instead, we need a solution that (1) provides a comprehensive defense against all these attacks, and (2) addresses the design and implementation challenges related to the aforementioned design principles.

3.5.3 `SecureReview` Design

In a typical code review workflow (as described in Figure 3.1 of Section 3.2.2), a `Merge Request` contains the proposed code changes that will go through a review process before being approved for integration into the codebase. As explained in Section 3.5.1, existing code review systems that follow this workflow have two fundamental shortcomings: The code review history is neither accessible nor reliable. To address these shortcomings, we propose `SecureReview`, a mechanism to ensure the integrity of a code review system, by which (1) the reviewers can sign their reviews, (2) signed reviews are stored along with the codebase, (3) auditors can validate a posteriori whether the code review process followed the intended review policy. `SecureReview` consists of three major components, which we describe next: (1) Create and store a signed code review policy, (2) Create and store signed reviews, and (3) Merge changes. We describe next these three components.

3.5.3.1 Create and Store a Signed Code Review Policy. Popular code review systems allow users to define a code review policy, *e.g.*, GitHub's Branch Protection Rules [97] GitLab's Merge Request Approvals [102] or Gerrit's Submit Rules [17].

Common to these code review systems is that a code review policy consists of a set of rules by which the owner of a software project can define the requirements that must be fulfilled before a proposed change can be merged into the codebase. GitHub, for instance, provides a small set of rules such as the minimum number of required approving reviews and dismissing stale pull request approvals. Gerrit, on the other hand, has a more complex set of rules and also allows authorized users to add new customized rules to the code review policy.

Unfortunately, current code review systems, do not protect the code review policy against a malicious server or against malicious reviewers. To address this issue, `SecureReview` adds two new attributes to the review policy.

- *Reviewers*: A list of users (*i.e.*, public keys) who can review the code. This ensures that only authorized reviewers are allowed to participate in the code review process.

- *Signature*: The review policy must be signed with the project owner's private key. This protects the review policy from being tampered with.

To protect the code review policy, we are faced with several challenges. First, we need to determine what should be included as part of the code review policy. In addition to the explicit code review policy rules, there is additional information that must be protected to ensure the integrity of the code review process, such as the CODEOWNERS file in GitHub, or the configuration files in Gerrit (`project.config`, `groups`, and `rules.pl` that reside under the `refs/meta/config` branch). Although not immediately obvious, if these are not protected, an attacker will be able to subvert the code review process, as shown in Section 3.4.

Second, the project owner needs a mechanism to sign the code review policy. Third, the signature should be stored on the server in a way that is accessible, so that it can be retrieved later, whenever verification is performed. Finally, most existing popular code review systems have a fixed set of policy rules and do not allow us to define new fields, *e.g.*, a field to store a signature over the review policy.

3.5.3.2 Create and Store Signed Reviews. `SecureReview` encapsulates each review in a *review unit* as shown in Figure 3.10. The <**current review information**> field contains the relevant information in the review, such as the reviewer's rating and/or a reviewer comment. The signature field is computed by the reviewer over the data shown in Figure 3.11, creating a "chaining" effect between review units: *Each review unit depends on the prior review unit, thus preventing unauthorized changes in the middle of the chain.*

The signatures will prevent the reviews from being tampered with, thus addressing one of the limitations of current code review systems. To address the other limitation, (*i.e.*, reviews are not accessible after the code review phase), we are faced with a challenge: How to make the reviews accessible in a way that requires no (or minimal) server-side changes in current code review systems? To tackle this challenge,

<current review information>

<reviewer name> <reviewer e-mail>

<review unit signature>

Figure 3.10 The *review unit*'s format. Each review unit is computed by the reviewer who performed the review.

<signature field of the previous review unit>

<current review information>

<reviewer name> <reviewer e-mail>

Figure 3.11 The data over which the signature field in a review unit is computed. Note that the signature for the first review in a `Merge Request` omits the first field, as there is no previous review unit.

`SecureReview` leverages the fact that most popular code review systems allow the user to update the `Merge Request` during the code review process. For instance, users can customize the commit message for the committed data either through the web UI or using REST APIs [18, 93, 101]. Using this feature, `SecureReview` creates a new signed commit object for each review and allows a reviewer to create and embed the corresponding review unit in the commit message of the commit object. Depending on the system, the new commit object is either a completely new object (GitHub), or is an amended commit (Gerrit) on top of the Merge Request branch. This ensures that the review is stored in accessible storage (*i.e.*, the source code repository). As a result, `SecureReview` provides verifiable guarantees about the integrity of the code review process. The new commit object is created by the reviewer on her local system and is pushed to the code review server (as described in Section 3.6).

`SecureReview` enables the reviewer to create the new commit object on her local system by first retrieving the signed head commit of the Merge Request branch, verifying the signature on this commit, and extracting the necessary information from it (*i.e.*, commit hash, tree hash, commit message) to compute a new signed commit

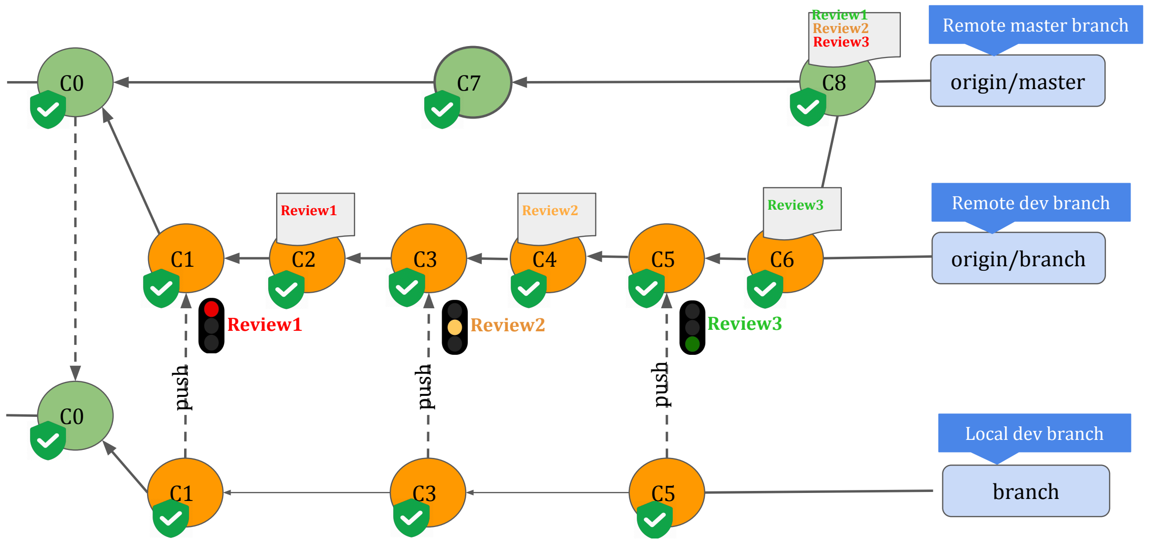


Figure 3.12 Create signed code reviews.

object. This new commit is then pushed to the code review server (as described in Section 3.6). Thus, each code review results in a new signed commit object in the `Merge Request` as depicted in Figure 3.12.

3.5.3.3 Merge Changes. When the `Merge Request` receives enough reviews and finally satisfies the code review policy, the Merger – a person in charge of merging changes into the codebase – merges the `Merge Request` into the codebase, which results in a new merge commit.

With `SecureReview` in place, the Merger needs to perform some additional actions. First, she retrieves and verifies the signed head commits of the branches that are being merged and all the signed review units from the commits of the `Merge Request`. Second, the Merger checks if each review unit contains the signature field of the previous review unit. Next, she checks the validity of the signature on each review unit. Then, the Merger checks if the sequence of reviews indicated by the signed review units leads to a code change that respects the existing review policy.

Finally, the Merger embeds the review units into the commit message of the merge commit. For this, `SecureReview` allows the Merger to create a standard Git signed merge commit object on the client side and push it to the server. This will ensure that the integrity of the code review process can be verified independently, even after the code review phase.

`SecureReview` provides a trade-off between security and usability by giving the Merger two options for integrating the review units into the merge commit:

- Compact integration: the Merger integrates the review units from the commits of the Merge Request, without the signature field in each review unit.
- Full integration: the Merger integrates the entire review units from the commits of Merge Request, including the signature field in each review unit.

From a security perspective, in the Full integration option, the Merger is trusted to include the review units, but cannot tamper with or remove review units from the middle of the chain of review units. However, the Merger may omit review units from the end of the chain. In general, since the Merger is usually a trusted entity (such as the project owner), this should not be a major concern. Still, the Full integration provides some accountability for the Merger, as it allows a verifier to check that no information was tampered in the review units that are included in the merge commit, thus reducing the trust in the Merger. As opposed to that, the Compact integration option fully trusts the Merger to correctly integrate the review units.

On the other hand, the Full integration option has the drawback that the commit message of the merge commit may increase significantly because of the extra signatures that are included. For example, if a change on the Gerrit server receives 10 reviews [3], the commit message will contain 10 additional signatures, which will add 100 lines to the commit message. A large commit message can affect the readability

[3]We note that at Google, each code change receives a peak of 12.5 reviews for changes of 1250 lines [146].

of the commit message, which will slow down the review process, and the its usability to automatically generate a release note.

PROCEDURE: Validate_Branch
Input: Branch, ReviewPolicy
Output: `success/fail`

```
 1: if Validate_Signature(ReviewPolicy) == false then
 2:   // The review policy does not have a valid signature
 3:   return fail
 4: end if
 5: C ← The set of all commits in the Branch
 6: h ← The head commit of the Branch
 7: // Check if all the branch commits have valid signatures
 8: for all c ∈ C do
 9:   if Validate_Signature(c) == false then
10:     return fail
11:   end if
12: end for
13: while (C is not empty) do
14:   // Extract the commits in the merge request that
15:   // corresponds to h
16:   MRC ← Extract_Merge_Request_Commits(h)
17:   // Check if the sequence of reviews embedded in the
18:   // merge request is valid against the review policy
19:   if Validate_Reviews(MRC, ReviewPolicy) == false then
20:     return fail
21:   end if
22:   // Remove commits in merge request from the set C,
23:   // and find the head of remaining commits
24:   C ← C \ MRC
25:   h ← The head commit of the branch represented by commits left in C
26: end while
27: return success
```

3.5.3.4 Verifying the Code Review Process.

When cloning or pulling changes from a repository, an auditor can verify the integrity of the code review process for each branch in the repository by executing the Validate_Branch procedure. The procedure first checks if the code review policy (lines 1-2) and the commits in the branch (lines 7-9) have valid signatures. It then traverses the commit tree in the branch and extracts the commits corresponding to a merge request by calling the

Extract_Merge_Request_Commit procedure (line 13). The Validate_Branch procedure then calls the Validate_Reviews procedure to check whether the merge request was properly approved according to the intended code review policy (line 16).

At a high level, Validate_Reviews contains three steps: (1) extract the review units from the commits of the merge request and validate the signature over each review unit; (2) check the chaining between review units (i.e., that each review includes the signature field of the previous review unit); (3) check whether the sequence of reviews indicated by the signed review units leads to a code change that respects the intended code review policy. For step (3) of this procedure, we check the most common policy rules such as if the minimum number of approvals is met and if the reviewers are authorized to provide approving reviews. However, since Gerrit allows a project owner to customize the review policy arbitrarily, a complete treatment of step (3) would require a more complex check that is outside the scope of this dissertation.

The Validate_Reviews procedure receives as input a valid signed review policy and a set of commits corresponding to a merge request and checks whether the merge request was approved according to the intended review policy. After validating the signatures on the review units and checking the chaining between review units (lines 1-5), it checks if a direct push (*i.e.*, no reviews) is detected (lines 6-12). Then if the merger is authorized (lines 13-18), the most common code review policy rules are checked (lines 19-22) such as if the merge request's creator was allowed to submit a code change; if reviews are created by authorized reviewers; if the sequence of review units meets the minimum number of approving reviews defined by the policy; if only those approvals that satisfy the review policy are counted; if there are required reviews from specific users; and if outdated approving reviews are dismissed correctly.

PROCEDURE: Validate_Reviews
Input: Commits, ReviewPolicy
Output: `success/fail`

```
Reviews ← getReviewUnits(Commits)
// Check if review units have valid signature and the chain of review units is valid.
if Validate_Review_Units(Reviews, ReviewPolicy) == false then
  return fail
end if
ru ← length(Reviews)
if ru == 0 and FirstCommit(Commits) == false then
  // Check if the committer has the direct push access
  if DirectPush(Commits, ReviewPolicy) == false then
    // Unauthorized direct push is detected
    return fail
  end if
else
  // Check if the merger has the permission
  if AuthorizedMerger(Commits, ReviewPolicy) == false then
    // Unauthorized merge is detected
    return fail
  end if
  // Check the common rules are followed.
  if CommonRules(Reviews, ReviewPolicy) == false then
    return fail
  end if
end if
return success
```

3.5.3.5 Versioning the Code Review Policy. Although it may happen seldom over the lifetime of a project, changing the code review policy is allowed in code review systems, including in GitHub and in Gerrit. `SecureReview` assumes that the code review policy does not change. However, it can be extended to support multiple versions of the code review policy as follows. When a code change is merged into the codebase, `SecureReview` includes the identifier of the current code review policy (we make the minimal assumption that each code review policy has a unique identifier). When an auditor runs the Validate_Branch procedure, it needs to retrieve the code review policy that was in place at the time each merge was performed. `SecureReview` also needs that the code review system maintains a history of the code review policy. In a system like Gerrit, the review policy is automatically versioned because the review policy is maintained in three files (`project.config`, `groups`, and `rules.pl`) located under the `refs/meta/config branch`. However, in a system like GitHub, `SecureReview` would need explicit server-side support for keeping the history of the review policy.

3.6 Deployments

In this section, we show how to integrate our design into two popular web-based code review systems, GitHub and Gerrit.

We implement `SecureReview` as a client-side Chrome browser extension. To review a `Merge Request`, a reviewer activates the extension via a browser toolbar button. The extension uses an isolated pop-up window to allow the reviewer to create a signed code review and integrate it into the Git repository before merging the `Merge Request` into the codebase. Though the pop-up window could be integrated with the original webpage, it is isolated to prevent malicious scripts (originating from the untrusted server) from tampering with the code review data. Indeed, this window can only be written by scripts associated with the `SecureReview` extension. This is a normal approach used by security-conscious browser extensions such as Mailvelope [37] and FlowCrypt [15]. The extension extracts information about the `Merge Request` from the web UI and from the code review server via REST APIs [18, 93, 101]. The review unit(s) are stored as part of the commit message of Git commit objects that are newly created by the extension on the client side and then pushed to the Git server.

To capture and store reviews, `SecureReview` provides two options in its Settings page: (1) Compact versus Full integration, (2) Include versus Exclude review text feedback. The first one, as described in Section 3.5.3.3, determines if the review units are integrated with or without their signature field. The second one determines whether the review text should be included in the review unit, in addition to the review rating. These two options help users control the size of the review units, thus influencing the readability of the merge commit messages.

`SecureReview` consists of two JavaScript scripts that communicate with each other via the browser's messaging API as follows:

- The *content script* runs in the browser and can read or modify the `Merge Request` page. It obtains the necessary information about the code review and passes the information to the background script.
- The *background script* cannot access the content of the `Merge Request` page but uses information relayed by the content script to perform the following core functionality of the extension, and then notifies the *content script* to reload the `Merge Request` webpage:
  - For signing and storing reviews: retrieve parent commit object (*i.e.*, parent commit, tree hash, commit message) create a *review unit*, embed it in the commit message, create a signed commit object and push it to the server;
  - For merging the change: check if the code review policy is being followed, create a signed merge commit object that includes the review units in the commit message and push it to the server.

To maintain efficiency and satisfy design principle DP6 (Minimize minimize the impact on the user's experience (*i.e.*, DP6), `SecureReview` is implemented exclusively in JavaScript and does not require users to leave the browser. For which, we leverage prior work that fetches information from the Git repository without retrieving the whole repository and without creating a working directory on the client side [55]. That is not possible in the standard Git client, which needs the entire repository locally to create a new commit. `SecureReview` creates signed commit objects on the client side and pushes them to the server by reimplementing in JavaScript Git operations such as committing, amending, merging files, pushing.

With `SecureReview` in place, an auditor can verify the integrity of the code review process if she has the source code repository along with the signed code review policy.

3.6.1 `SecureReview` for GitHub

In this section, we describe how `SecureReview` adds verifiability to GitHub code review process by providing code reviewers the ability to sign reviews and store them as part of the Git repository.

Create and Store a Signed Code Review Policy. In GitHub, the information that must be protected because it is relevant for the code review policy, consists of the actual policy rules and the CODEOWNERS file which contains a list of individuals that can provide approving reviews.

Since GitHub provides no mechanism to protect this information, `SecureReview` allows the project owner to compute a digital signature over it. To store the signature, `SecureReview` defines a new *status* for the corresponding branch. Normally, a status in GitHub is used to check if the commits meet the conditions set for a branch (*e.g.*, if the build after a commit is successful or not) [99, 90], but `SecureReview` stores the signature as a status parameter, which accepts arbitrary strings. The signature can be retrieved later from the server, whenever there is a need to attest the integrity of the code review policy.

Create and Store Signed Reviews. Once a GitHub developer submits a new code change through a pull request, it can be updated by adding new commits. This allows her to either update the code or commit information such as the commit message. Developers can use either the web UI, or the command line, or the GitHub REST API [93] to update the pull request. `SecureReview` leverages GitHub's REST API to embed the reviews in the Git repository.

For each new review, `SecureReview` updates the pull request by creating a new commit object. For which, `SecureReview` first creates a review unit that encapsulates the review. The "current review information" field includes the rating (which can be one of *Approve*, *Request changes*, or *Comment*), and text feedback with the reviewer's comments. It then retrieves the latest commit in the pull request and uses its fields to create a new Git commit object that is almost identical, except for two fields: the commit message field, in which it embeds the review unit, and the committer field, which is set to be the reviewer. Finally, `SecureReview` pushes the new commit to the GitHub server.

To illustrate the "store reviews" on GitHub, we consider the example shown in Figure 3.2. In which, the reviewer makes three reviews to approve the `Pull Request`. Having `SecureReview` in place, all review information is securely captured as shows in Figure 3.12. Reviewing commit C1, the reviewer requests for changes by making *Review1*. As a result, a new commit (C2) is created on top of C1. At a high level, `SecureReview` helps the reviewer to perform the following steps to embed the review in the GitHub repository: (1) form a *review unit* to store *Review1*; (2) retrieve the latest commit from the server to extract the commit hash (*i.e.*, C1) and the "tree hash" (*i.e.*, T1) (3) compute a signed commit object (*i.e.*, C2) to embed the *Review1* in the commit message; (4) push C2 to the server. In a similar manner, the reviewer uses `SecureReview` to embed *Review2* and *Review3* in the repository as shows in Figure 3.12.

Merge Changes. A pull request branch may be merged into the codebase in four different ways: (1) as a fast-forward merge, (2) as a regular merge commit with two parents, (3) using a rebase-and-merge, or (4) using a squash-and-merge.

For the first three merge methods, `SecureReview` does not interfere with the regular GitHub merge operation, and the commits created by `SecureReview` (which contain review units) will become part of the project's commit history. For the last merge method (squash-and-merge), `SecureReview` extracts the review units from the commits of the pull request branch and integrates them into the merge commit object as described in Section 3.5.3.3. When `SecureReview` is in place on GitHub, it is recommended that the squash-and-merge method should be used to merge a pull request in order to avoid adding unnecessary commits to the commit history.

3.6.2 `SecureReview` for Gerrit

Gerrit's code review policy is defined in three configuration files (*i.e.*, `project.config`, `groups`, and `rules.pl`) stored under the `refs/meta/config` branch. However, Gerrit does not provide a mechanism to protect these files. To address this issue, `SecureReview` allows the project owner to sign the code review policy (*i.e.*, configuration files) per repository. Gerrit allows the project owner to customize the code review policy by creating labels on which reviewers vote to express their opinion about a change (*e.g.*, a predefined label is the "Code-Review" label) [85, 84]. The value of a label can be an arbitrary string and `SecureReview` defines a custom label to store the signature over the code review policy. This custom label can be later retrieved from the Gerrit server to verify the integrity of the code review policy.

Create and Store Signed Reviews. Upon creating a change, the developer usually provides a text description for the change. This description is included in the commit message of the initial commit corresponding to the change. Gerrit allows developers to update the commit message of the change by using the web UI, or the command line, or the Gerrit REST API [18]. `SecureReview` leverages the Gerrit REST API to embed the reviews in the Git repository.

When a reviewer performs a new review, `SecureReview` creates a new patch set locally in the reviewer's browser, embeds the review in this patch set, and pushes the patch set to the Gerrit server. To do this, `SecureReview` first creates a review unit that encapsulates the review. The "current review information" field includes the rating (which by default is an integer ranging from -2 to +2) and text feedback with the reviewer's comments. It then retrieves the latest patch set in this change and uses its fields to create a new Git commit object for the new patch set. This new patch set is a standard Git signed commit that differs from the latest patch set in two fields: the commit message field (which will contain the newly created review unit) and the committer field, which is set to be the reviewer. Finally, `SecureReview` pushes the new patch set to the Gerrit server.

Merge Changes. Gerrit allows the Merger to select among six merge strategies called "submit types". Acting on behalf of the Merger, `SecureReview` first retrieves all patch sets in the `change` from the server. Then, it examines all the patch sets with review units in this change and verifies the validity of the signature field of each review unit. It also verifies that each review unit includes the signature field of the review unit in the previous patch set that contains a review unit. If security checks are passed, `SecureReview` extracts the review units from the patch sets of the change and integrates them into a new patch set, represented by a new merge commit object) as described in Section 3.5.3.3. Depending on the merge strategy, the merge commit object will have either one parent (*e.g.*, for fast forward and rebase) or two parents (for merge always). Finally, `SecureReview` pushes the signed merge commit object to the server.

3.7 Security Analysis

In this section, we first show how `SecureReview` achieves the security guarantees defined in Section 3.3.1. Then, we analyze `SecureReview`'s ability to mitigate the attacks described in Section 3.4.

3.7.1 Achieve Security Guarantees

`SecureReview` allows the project owner to protect the code review policy using a digital signature. The review of a proposed change consists of a chain of signed review units. Each review unit is protected by a digital signature. Thus, any tampering with the review policy, or with the sequence of review units, or with individual review units, will be detected during the verification procedure. Thus, under the considered threat model, `SecureReview` achieves **security guarantee SG1**.

When a proposed code change is about to be merged, `SecureReview` checks whether (1) each review unit contains the signature field of the previous review unit, (2) each review has a valid digital signature, and (3) the code review policy has a valid signature. `SecureReview` also checks whether the sequence of review units satisfies the intended code review policy, by checking if the policy rules are satisfied. Each individual review cannot manipulate the reviews of other reviewers, since `SecureReview` does not rely on individual reviewers handling reviews other than their own reviews. As such, under the considered threat model, the code review process cannot be manipulated, and `SecureReview` achieves **security guarantee SG2**.

An auditor should be able to independently verify the integrity of the code review process even after the code review phase. That is not possible in the current code review systems (such as GitHub, GitLab, and Gerrit) even if the auditor could have access to the code review database. `SecureReview` embeds the review of each proposed change in the commit message of a signed commit object which is created by a trusted individual and is stored in the source code repository. Auditors that clone the source code repository can retrieve the signed code review policy and can verify the correctness of the code review process. Thus, `SecureReview` achieves **security guarantee SG3**.

3.7.2 Mitigate Attacks

Code Review Manipulation Attacks. The server may alter the code reviews, for example, by improving the review score to make a change mergeable. Since code reviews are signed and `SecureReview` does not expect a reviewer to handle reviews other than their own reviews, any modification in the review score will be detected during the verification procedure – the Validate_Reviews procedure (described in Section 3.5.3.4) fails in line 4.

Bypass a minimum number of approving reviews. The server can make a change mergeable even though it has not received the minimum approved reviews. With the help of the code review chaining and the signed code review policy, the verification procedure can detect this malicious behavior – the Validate_Reviews procedure fails in line 11 or 21.

Count reviews from unauthorized reviewers. The server may count approving reviews from unauthorized reviewers (*i.e.*, author of the code change) toward the minimum number of approvals. However, the attack will be detected during the verification procedure when the reviewer's permission is checked against the code review policy – the Validate_Reviews procedure fails in line 21.

Exclude required reviews from specific users. A malicious server may declare a change mergeable even though there are not enough approving reviews from specific users (*e.g.*, code owners in a GitHub project). Checking that policy rule, the verification procedure will detect the server's misbehavior – the Validate_Reviews procedure fails in line 21.

Count outdated reviews. The attacker may disable a policy that helps to catch bugs introduced by a patchset by counting outdated review toward the minimum number of approvals. With the sequence of signed review units, `SecureReview` can help to verify whether the stale approvals were dismissed during the code review process – the Validate_Reviews procedure fails in line 21.

Misuse the project owner's authority. A malicious server can impersonate the project owner's role to merge vulnerable code changes. This attack will be detected during the verification procedure when we check if the merger was authorized to merge the code change – the Validate_Reviews procedure fails in line 17.

Accept changes from unauthorized users. A malicious server may ignore the policy that enforces accepting changes only from a specific group of authors. With a deeper inspection over the code review chain, the verification procedure can detect the server's misbehavior – the Validate_Reviews procedure fails in line 21.

Advanced Attacks. There are some sophisticated attacks that may not be caught during the verification procedure if it relies on the checks described above. For example, the server may make a change submittable even though all requests for changes have not been addressed. Moreover, a malicious server can modify different project settings (such as user permissions, required status checks, adding arbitrary review policies, etc) to achieve AG1 and AG2. A very complex check (which is outside the scope of this chapter) is needed to address these types of attacks.

3.8 Experimental Evaluation

In this section, we explore the different costs of deploying `SecureReview` on both Gerrit and GitHub. For benchmarks, we picked five popular repositories from GitHub [4]. and five popular repositories from Gerrit Google Source [21]. To cover diverse repository configurations, we chose repositories of different history sizes, file counts, and file sizes (shown in Table 3.1[5]).

We conducted our experiments on a client system with an Intel Corei7 CPU at 2.70 GHz and 16 GB RAM. The client software consisted of Linux 5.0.16-100.fc28.x86_64 with git 2.19 and GnuPG 2-2.7 for 2048-bit RSA signatures. On the server side, we used GitHub.com itself and a self-hosted Gerrit server on an Amazon EC2 instance with two vCPUs (Intel XeonE5-2686 at 2.30GHz), 8 GB RAM, Linux version 5.3.0-1023-aws, Gerrit server 3.2.2, and git 2.17. We also note that: (1) the time to push the Git commit object to the server is not included in the measurements, (2) when running `SecureReview`, only one CPU core on the client side was used, (3) all commits in the repository have a GPG signature (as mentioned

[4]Popularity is based on the "star" ranking by GitHub users, which reflects their level of interest in a project as of July 20, 2020.

[5]The top five are GitHub repositories, and the bottom five are Gerrit repositories.

Table 3.1 Repositories Chosen for the `SecureReview`'s Evaluation

Repository	Size (MB)	File Count	File Size (Bytes)	History Size (# of commits)
gitignore	4	235	590	3,253
vue	24	549	10,848	3,104
youtube-dl	64	925	6,473	17,720
react	134	1,815	9,576	13,415
go	338	9,140	10,423	44,192
bazlets	1	43	2,209	377
gitiles	6	185	4,818	933
gitblit	29	1095	9,997	3,072
jgit	53	2,463	5,791	7,938
gerrit	234	4,717	7,288	44,944

in Section 3.3), (4) all experimental data points in this section represent the median over 30 independent runs. For each run, we picked the latest (preferably open) 30 merge requests for each repository.

We focused our experiments on two performance parameters: execution time and storage overhead. We benchmarked both integration options of `SecureReview` (*i.e.,* "Full" and "Compact"), in all these parameters except for execution time, where the runtime differences were negligible (and thus we only show one metric).

Execution time. Table 3.2 shows the execution times for two major functionalities of `SecureReview`: Creating signed reviews and Merging changes. The former includes reviewing a code change in a merge request branch. For which `SecureReview` downloads only the head of the merge request branch. The latter is merging a code change that has been reviewed four times.

Table 3.2 Execution Time for Storing Signed Reviews and Merging Changes

Repo.	Sign and Store	Merge Changes
gitignore	0.61	1.16
vue	0.67	1.47
youtube-dl	0.65	1.58
react	0.69	1.65
go	0.68	1.94
bazlets	0.37	0.40
gitiles	0.36	0.40
gitblit	0.37	0.41
jgit	0.37	0.42
gerrit	0.37	0.41

To perform a merge commit, `SecureReview` carries out two major steps. First, it retrieves all review units, validates their signature, and integrates them into the commit message. Second, it computes the merge commit object, which could result in retrieving several tree and blob objects from the server. This depends on the number of changed files and the location of these files in the repository. Of note, the size of the repository is not a major factor for the `SecureReview`'s performance – the execution time is affected by the number of retrieved Git objects

On Gerrit, `SecureReview` can sign and store code reviews in less than half a second. On GitHub, however, it takes a bit more to perform the same operation. `SecureReview` can merge changes on Gerrit repositories in under a second whereas, for Github, this time is closer to a second slower. These differences occur because the GitHub server's response to download Git commit objects is not as fast as our customized Gerrit server. This plays a major role in our tests since `SecureReview` requires more GitHub API calls to compute a merge.

We note that increasing the number of reviews per merge request will not affect execution time significantly because we use REST API [18, 93] to get all merge request commits (*i.e.*, all review units embedded in commits) at once. More review units could affect the time to verify signatures over review units. However, in the merge operation, signature verification has a very small portion of the overall execution time (for a merge request with four review units, signature verification takes 0.023 seconds out of the 0.40–1.94 seconds needed to execute the merge request).

Table 3.3 Storage Overhead per Merge Commit

Repository	Commit Size (Bytes)	C1	C4	F1	F4
DIFF (Bytes)		58	232	572	2288
gitignore	995	6%	23%	57%	230%
vue	784	7%	30%	73%	292%
youtube-dl	1127	5%	21%	51%	203%
react	1061	5%	22%	54%	216%
go	985	6%	24%	58%	232%
bazlets	903	6%	26%	63%	253%
gitiles	1807	3%	13%	32%	127%
gitblit	788	7%	29%	73%	290%
jgit	2257	3%	10%	25%	101%
gerrit	1392	4%	17%	41%	164%

Storage overhead. Table 3.3 shows `SecureReview`'s storage overhead for both integration options and a different number of review units per merge request. The storage overhead caused by `SecureReview` is the data added to the commit message – equal to the size of review units. As shown in Table 3.2, we measured the storage overhead for four configurations: (C1) Compact integration with one review unit of

58-byte size, (C4) Compact integration with four review units with a total size of 232 bytes, (F1) Full integration with one review unit of 572-byte size, (F4) Full integration with four review units with a total size of 2288 bytes.

To measure that overhead, we first computed the size of merge commit objects created during our execution timing tests (which included four review units per merge commit) and contrasted it to the same merge commits without `SecureReview` enabled. Then, we computed the median difference size between two equivalent merge commits and therefore measured the overhead caused by embedding the review units in the repository. Also, we repeated the previous experiment when each merge request has only one review unit.

When Compact integration is in place, each review unit added about 58B to the merge commit object (and a total of 232B for four review units). For Full integration, the storage overhead for one and four review units was 572B and 2288B, respectively. The Compact integration adds less than 8% overhead per review unit. Full integration, however, has an overhead of 25% to 73% for different repositories. To put these values in context, the largest storage overhead in Table 3.3 is approximately 2 KB per commit for full integration of reviews, which represents less than 0.0006 of the repository size even for a small repository like gitignore. Of note, the storage overhead depends on the size of the original commit message and the integration option –the larger the commit message is, the less the overhead caused by `SecureReview`.

3.9 Related Work

`SecureReview` is, to the best of our knowledge, the first tool to secure the integrity of the code review process. Previous attempts in this area have been mostly focused on identifying best practices to prevent security vulnerabilities in the early steps of the software development life cycle (i.e., prevent implementation defects before releasing a software product [157, 127]), improving the code review tools themselves, and improving merge request management. In addition, there is a trove of work related to the version control system (VCS) security that proves relevant to the security of code review systems.

There is plenty of work on improving code review quality. Work by Bosu [63] describes a methodology to evaluate the impact of peer code review on security vulnerabilities in Open Source Software (OSS) communities. For instance, it suggests that analyzing a set of keywords used in the code review comments could lead to identifying security vulnerabilities. This approach is extended in another study [64] by finding the code changes that are more prone to contain security vulnerabilities by asking the code reviewers to pay more attention to such code snippets. Work by Ogale [136] proposes a tool to identify the scope of mandatory code review which is part of the source code that should be manually reviewed. Ibrahim *et al.* [110] give a checklist for the secure code review process that should be done before releasing the software or even before committing the code to the codebase. While all these tools improve the security stance of a project, they do not address that the review process and policy are followed. In this case `SecureReview` can be useful to ensure that guidelines by these systems are present (*e.g.*, by encoding them in the review policies).

Kalyan *et al.* [113] explore the shortcomings of existing code review tools and introduce Fistbump, a tool to improve the code review process on GitHub. Fistbump manipulates the existing pull request interface on GitHub and allows to manage the discussion between the author of a merge request and the selected reviewers. Fistbump also applies best practices in web security, such as cross site scripting (XSS) prevention, and HTTPS communication. Using GitHub APIs, CodeReviewHub [8] creates a task list per pull request on GitHub and adds a pending task for each comment. The main goal is to improve the pull request management by keeping track of unaddressed comments and open issues. RevRec [166] tries to improve the code review process on pull request based systems such as GitHub by recommending the best code reviewers. These systems are improving the developer experience and workflow, and their improvements can accommodate `SecureReview` mechanisms to provide verifiability that these workflows are followed.

In addition, there are studies that improved the security of the VCS which relate to the security of the code review process as well. Wheeler [163] provides an overview of security issues related to software configuration management tools and provides a set of security principles, threat models, and solutions to address these threats. Gerwitz [87] provides a detailed description of creating and verifying Git signed commits. Work by Torres-Arias *et al.* [160] covers some attack vectors against VCS where a malicious server tampers with Git metadata to trick users into performing unintended operations. Whereas this line of work focuses on the consistency of the VCS itself, `SecureReview` extends this notion to provide integrity and consistency to the process by which changes are integrated into the VCS itself.

Perhaps most closely related work to `SecureReview` is `le-git-imate` [55], a defense scheme that provides security guarantees for code changes performed through untrusted web-based Git repository hosting services such as GitHub and GitLab. However, `le-git-imate`'s focus is on signing a web UI commit and creating a standard GPG-signed Git commit object in the browser. Though this feature is crucial for the success of `SecureReview`, it does not provide the ability to validate the integrity of the code review process.

CHAPTER 4
AUTOMATED VALIDATION OF THE CODE REVIEW PROCESS

4.1 Introduction

Code review is the process of spotting bugs and errors at the early stages of software development to improve the quality of the source code. Thus, it reduces the amount of time developers must spend to fix defects after the software product is delivered to the end user. The code review process, however, can be incredibly tedious and time-consuming if performed only by a human.. Code review tools address this issue by automating the code review process. Such tools help developers to easily track code changes, find bugs and errors, and ensure that issues are resolved correctly [58, 146, 59].

In the past years, a variety of features have made the code review tools more popular. Our analysis of the top 50 most starred GitHub projects (*i.e.*, evaluating 690 thousand commits, 207 thousand pull requests, and 77 thousand reviews) show that 42 out of 50 top projects have started using GitHub code review features, and an average of 38% of all pull requests have been reviewed. Notably, many of pull requests with no code reviews were created before 2015 when GitHub did not have even basic code review features such as "Highlighted Diffs" (which adds colors to the code changes) [25].

Code review services, such as GitHub [23] and Gerrit [16], allow the project owners to define a code review process enforcing certain requirements before merging code changes into the codebase. GitHub, for instance, allows defining a set of policies called protection rules for each branch of a Git repository. Doing so, all proposed code changes (*i.e.*, pull requests) must undergo a code review process before being merged to the codebase. The review process itself can be defined or modified only by authorized users such as the project owner or the repository administrators. Besides, project owners can automate many steps of the code review process and find security vulnerabilities by using code review tools [91], as well as code scanning tools [92]. This all leads to less human errors during the code review and also fastens software development.

Despite many features provided by modern code review tools, one essential feature is still missing. Even if a project owner defines a flawless code review workflow, there is no guarantee that the code review process was followed as intended by the project owner – the integrity of the code review process is not protected. Indeed, the users or any independent auditor must assume that the code review server faithfully followed the code review policy. That is not a wise assumption while a malicious or a compromised server can tamper with any aspect of the code review workflow. For instance, a server may impersonate the project owner's role to dismiss negative code reviews, or add unauthorized reviewers to participate in the review process, or bypass any policy (such as the minimum number of approving reviews) without being detected.

Unfortunately, when the code review step is done, and the source code is shipped out to the next step of the software development, there is no mechanism, as an independent auditor, to fully verify whether a review policy was violated during the code review process. In other words, current code review systems cannot provide strong guarantees about the code review process and are vulnerable against a wide range of attacks (as described in Section 3.4). We could tackle this issue if we were able to validate the integrity of the entire code review process – verify after the code review step what occurred in that step. To this end, we must provide two essential security properties: (1) create verifiable metadata about the code review process, (2) a way to adequately verify a sequence of code reviews against the code review policy.

In Chapter 3, we achieved the first goal by introducing `SecureReview` which helps to sign the code review policy as wells embedding verifiable guarantees about the code review process in the source code repository. To attain the second goal, we present `PolicyChecker` that allows auditors to automatically verify the correctness of the code review process by evaluating the code review metadata. A tool like `PolicyChecker` is useful in two steps of the software supply chain: 1) when a maintainer merges a branch, so she does not have to blindly rely on the code review server, which happens a priori to the code merging step, 2) when someone pulls a repository and wants to check if the code was merged according to the code review policy, which happens a posteriori to the code merging step.

We note that verifying the code review metadata is not only a matter of verifying the authenticity and integrity of the code reviews (*i.e.*, verifying a digital signature). It is also about ensuring that a sequence of code reviews which led to the approval of the code changes respects the intended code review policy. Depending on the code review workflow, this process can be quite complex and error prone if done manually. For instance, assume a GitHub project that enforces three approving reviews for any code change, dismissing any reviews after a new code change, ignoring code changes from specific users, and finally requiring review from certain users. Having such a code review policy in place, it is not easy to manually determine if a code change was merged correctly. This could get worse, for example when it comes to validate a software release with many code changes.

In this chapter, we make the following contributions:

1. We perform a comprehensive analysis of two popular code review systems, GitHub [23] and Gerrit [16]. That leads to answer the critical question of what can be done by whom? For instance, we find out the internal code review workflow, the exact components of the code review policy on each system, the user permissions, and how code reviews are evaluated on GitHub and Gerrit.
2. This analysis enables us to design `PolicyChecker`, the first tool that allows independent auditors to verify the correctness of the code review process.

`PolicyChecker` receives as inputs the source code repository, the code review policy, the code review format, and the corresponding public keys. It then automatically validates that the code review process was followed as prescribed by the project owner.

3. We implement `PolicyChecker` as a git command which can be run whenever a repository is cloned or pulled. Our implementation does not impose any changes on the server side so that it can be used today. It also preserves the current code review system's workflows. `PolicyChecker` is implemented in Python and can verify the code review process on GitHub and Gerrit. However, it can be easily adapted to work with any Git-based hosting services (such as GitLab [28], BitBucket [4], and GerritHub [20]) as long as the code review format and the code review policy are known.

4. We analyze our implementation's efficiency and show that `PolicyChecker` can validate the code review process in a repository branch in a timely manner – on average less than 0.12 seconds to evaluate one merge request, and less than 1.50 seconds to verify a software release.

4.2 Background

Section 3.2 introduced the code review process and two popular code review systems, GitHub [23] and Gerrit [16]. In this section, we elaborate on the code review policy (*i.e.*, the rules and the user permissions) and the merge strategy as the background for the code review process verification.

4.2.1 GitHub Code Review Policy

Github allows project owners to define a review process on a branch basis called "Branch Protection Rules" [97]. The review process for each branch can be defined by enabling at least one of the protection rules. We call GitHub's code review policies "semi-customized" rules, in which the project owner has a limited set of options to update the code review process and does not allow to define arbitrary customized rules.

In the following, we explain the rules that are relevant for the code review on GitHub [1]:

1. Required number of approving reviews: A number between 1 to 6 that indicates the minimum number of approving reviews that a change must receive before it can be merged.
2. Dismiss stale pull request approvals: If enabled, the code review process is reset when code-modifying commits are pushed to the merge request branch. Thus, any existing approving reviews will not be counted towards satisfying the required number of approving reviews.
3. Require review from code owners: If enabled, the required number of approving reviews must only be from "code owners", a set of designated users who are responsible for certain files in a repository.
4. Restrict who can dismiss pull request reviews: This rule specifies users or teams (*i.e.*, administrators or people with write access) that are able to remove reviews. That could help when the project owner does not want to dismiss all stale approvals automatically, or the reviewer who requested for changes is not available anymore to give an approving review.
5. Require linear history: If enabled, merge commits can not be created. Therefore, a merge request can be merged only under the "squash and merge" or "rebase" merge strategy.
6. Include administrators: If enabled, any code changes by administrators must be reviewed before being merged to the codebase.
7. Restrict who can push to matching branches: This rule specifies users, or teams that are allowed to push to a branch.

4.2.2 Gerrit Code Review Policy

Code changes on the Gerrit server are reviewed under a set of policies called *submit rules*, a logic that evaluates proposed code changes through a rating process to determine if the code changes are mergeable. As described in Section 3.2.4, *submit rules* are group based and are defined in three files, `project.config` (which contains group permissions per namespace, and code review ratings), and `groups` (which shows user groups). `rules.pl` (which indicates the vast majority of customized code review

[1]We focus on *organization repositories*, which are typical for collaborative projects. However, our work can also cover *user account repositories*, which have a more limited set of permissions.

policy), We note that Gerrit's default rules and any customized rules defined through the web UI reside under `project.config` and `groups` files; other customized rules will be to the `rules.pl` file.

Standard Rules. Gerrit default rules determine user premissions to: (1) create a branch, (2) create a code change, (3) review a code change, (4) puch commits without reviews, (5) merge code changes, (6) change the project settings, (7) change the default merge strategy. In addition, the default code review rating ranges from -2 to +2. The highest rating (+2) means that the reviewer approves the code change, and the lowest rating indicates that the code change cannot be merged unless the viewer's feedback is addressed. The project owners and administrators are the only user groups that can give the highest or lowest ratings (*i.e.*, approve or block a code change). Another user group that may participate in the code review process is called registered users, who can rate the changes from -1 to +1. Thus, they have no power to reject or approve a code change. The default submit rule indicates that a code change is mergeable if it has received at least a review with the highest score (+2) and no reviews with the lowest score (-2).

Customized Rules. Gerrit allows users to modify the default or standard code review rules either through the web UI or the command line. Considering seven permissions determined by default, the first five permissions can be updated through the web UI. We call such changes "semi-customized" rules in which the project owner just modifies some standard rules through the web UI. It is notable that "semi-customized" rules will be reflected in two files, `groups` and `project.config`. Updating the last two standard permissions (*i.e.*, change the project settings, change the default merge strategy) or adding any arbitrary rules (*e.g.*, modifying the code review rating range) must be done through the command line. We call such changes "customized rules" which are added to the `rules.pl` file using the Prolog language [17]. We note that Gerrit has implemented the default submit rules in

Java [19] and does not have any `rules.pl` by default. This file is only created by users who want to customize the submit rules. Note that the new rules defined in the rules.pl file will always override the default rules (the new rules have a higher priority).

4.2.3 Merge Strategy

GitHub. GitHub provides three strategies to merge a pull request into the codebase: *Merge* (equivalent to "`git merge --no-ff`" command that always results in a new merge commit object even if the fast-forward option is possible), *Squash and Merge* (the pull request's commits are squashed into a new single commit in the codebase), *Rebase* (slightly different from "`git rebase`" command since the committer information is updated by GitHub). The Merge strategy will always add at least two new commits to the codebase: (1) a new merge commit with two parents (*i.e.*, the previous head of branch, the head of the merge request branch), (2) one or more commits from the merge request branch. The *Squash and Merge* strategy will always result in one new commit in the codebase. Finally, the *Rebase* strategy will rewrite all the merge request branch's commits to the codebase. It thus adds one or more commits to the codebase depending on the number of commits in the merge request branch.

Gerrit. Gerrit provides six merge strategies (*i.e.*, "submit type"): *Fast Forward Only* (the head of the codebase repository is fast-forwarded to the change commit), *Merge Always* (equivalent to "`git merge --no-ff`" command), *Merge If Necessary* (equivalent to "`git merge --ff`" command that fast-forwards the head of the repository unless it is not possible and a new merge commit must be created), *Rebase Always* (equivalent to "`git rebase`" command that rewrites the latest patchset and then fast-forwards the head of the repository to this new commit), *Rebase If Necessary* (equivalent to *Fast Forward Only* the fast-forward option is possible. Otherwise, it is equivalent to *Rebase Always*), and *Cherry Pick* (picking the last patchset to create a brand new commit on top of the codebase). Upon integrating a merge request, Gerrit creates one or at most two new commits in the codebase. In case of the *Merge Always* or *Merge If Necessary*, two commits will be added to the codebase: (1) a new merge commit with two parents (*i.e.*, the previous head of branch, the head of the merge request branch), (2) one or more commits from the merge request branch. In other merge strategies, only one commit will be added to the codebase (*i.e.*, the head of the merge request branch).

4.3 Threat Model

We consider a threat model in which the attackers aim to compromise the code review process – prevent the source code from being properly vetted by the code reviewers. That can result in shipping a counterfeit or vulnerable version of the source code to the next step of software development. The attackers can achieve their goal by manipulating either the code review policy or the code review process's steps. As an illustration, the attackers can impersonate the project owner's authority or count reviews from unauthorized reviewers to merge a code change that does not satisfy the intended code review policy. They can even take advantage of stale approvals to merge a code change with insufficient review scrutiny. The attackers may also decide to change the reviewers' permissions to delay the merging of an approved security patch. They can also improve the rating of existing code reviews, or delete negative reviews, or add new illegitimate positive reviews.

Normally, the code review server should faithfully enforce the correct code review policy before merging code changes into the codebase. However, a malicious code review server (either unscrupulous or compromised) may not respect the prescribed code review policy by executing some of the aforementioned attacks. For example, the code review server may present a code change as mergeable, when in fact, it does not meet the prescribed code review policy. Concretely, a GitHub server may show the "green Merge button" to the project owner, even when not enough approving reviews have been submitted.

We assume that all commits in the source code repository and the code reviews are signed. We assume that code reviews are stored in the source code repository (*e.g.*, as part of the commit messages). The ability to sign and store the code reviews is available by using a tool such as `SecureReview` (which is described in Section 3.5). We assume that the code reviews are either created by `SecureReview`, or any tool that preserves the sequence of code reviews performed for each code change. We assume that the code review policy format is known. For example, GitHub's code review policy is a combination of branch protection rules, code owners, and collaborators. Gerrit's code review policy is formed by the project.config, rules.pl, and groups files. We assume that the code review policy is signed by the project owner and is accessible to any independent auditor (either through access to the source code repository or an offline channel). We assume that the public keys of project owners, reviewers, and developers are known and that the attacker can not compromise the corresponding secret keys.

4.4 Solution

Current code review systems would be secure against a wide range of attacks described in Section 3.4, if we could achieve two security goals: (1) capture verifiable metadata about the code review process, (2) find a way to verify the sequence of code reviews. The first goal was fulfilled in Section 3 by proposing `SecureReview` which allows signing the code review policy and capturing verifiable metadata about the code review process. In this section, we present `PolicyChecker`, a tool that enables automatic verification of a given set of code reviews against a given code review policy. `PolicyChecker` first interprets the code review policy to extract the rules and user permissions. It then performs multiple checks to ensure that the review process was followed as intended by the project owner.

A Strawman Solution A basic attack described in Section 3.4.1 is to manipulate the code reviews (*i.e.*, lower the rating of existing reviews, remove positive reviews, or add new negative reviews) which can result in approving a buggy code change. One solution to mitigate this attack is to verify the code review signatures. That helps to ensure the integrity of the code reviews and, therefore to mitigate the above attack. However, this solution is incomplete since it can not detect or prevent many code review policy manipulation attacks (described in Section 3.4.2). A malicious server, for example, can count outdated reviews, bypass a minimum number of approving reviews or accept changes from unauthorized users. None of these attacks are detectable by just checking the code review signatures. Hence, we need a comprehensive solution that verifies the code review signatures, interprets the code review policy, and checks if the code review policy sequence followed the code review policy (as prescribed by the project owner).

4.4.1 Design Overview

`PolicyChecker` allows independent auditors to verify the correctness of the code review process performed on the web-based code review services (*i.e.*, GitHub and Gerrit) that are integrable with a Git repository. When the review process is enabled on code review systems such as GitHub or Gerrit, each code change (*i.e.*, merge request) must go through a review process before being merged into the codebase. Thus, we can validate this process by first examining individual merge requests (finding out which code reviews led to a merge request being integrated into the codebase), and then by determining if the sequence of the code reviews satisfies the intended code review policy.

To design `PolicyChecker`, we are faced with three challenges. First, we should interpret a variety of policy rules in the code review process. A comprehensive analysis of Gerrit and GitHub internal workflow allows us to build a system that extracts different types of rules, determines the priority that each rule shall have in making a code change mergeable, and deals with a range of dynamic changes in the code review policy (*i.e.*, customized rules described 4.2.2). We illustrate the `PolicyChecker`'s interpreter in Sections 4.4.2 and 4.5.2 for GitHub and Gerrit. Second, we must be able to verify the integrity of an entire chain of reviews. Our solution relies on the fact that each code review is signed by the reviewer and also includes a valid GPG signature computed over the previous code review in the merge request. Hence, we can validate the integrity of a set of reviews embedded in a merge request. Third, we need to determine the boundary of each merge request. Without finding the first and last commit of merge requests, it is not possible to a group of linked reviews (*i.e.*, reviews embedded in one merge request). We address this challenge by leveraging the fact that for each merge request, a combination of the following information is unique: the merge strategy, the number of parent commits, the committer, and the number of code reviews embedded in each commit. Section 4.4.4 describes an algorithm that can determine the boundary of each merge request on Gerrit and GitHub.

`PolicyChecker` is built upon three main components: (1) interpret the code review policy, (2) extract the code reviews, (3) validate the code reviews against the code review process. Next, we describe each component, when applied to GitHub and Gerrit.

4.4.2 Interpret the Code Review Policy

Recall from Section 4.2 that a code review policy falls into three categories:

1. Standard rules: the project owner relies only on a set of fixed predefined rules (*e.g.*, default policies on Gerrit).

2. Semi-customized rules: the project owner changes some standard parameters of the existing rules (*e.g.*, branch protection rules on GitHub, new rules defined through Gerrit's web UI).
3. Customized rules: the project owner defines rules arbitrarily (*e.g.*, new rules defined through Gerrit's command line).

Though interpreting each code review policy faces different challenges, in any case, we have two common goals: (1) extract individual rules and user permissions, (2) determine the relation between different rules (*e.g.*, if a certain rule has a higher priority than others). As an illustration, assume a GitHub repository in which every code change must go through the code review process. If we find a piece of code with no code reviews, and we ensure that code review policy allows merging code without code reviews (*i.e.*, the committer must have the "admin" permission to the repository and the "push" permission to the branch), we may miss a violation in the code review process. Indeed, there might be another rule enforcing the administrators to follow the code review process. In a similar scenario, we may find a code change being merged by a user who has "write" permission to the branch (*i.e.*, the user is an authorized merger), while there is another rule saying that only a specific group of users can push into the branch. That said, we must not only understand individual rules but also find out how different rules affect each other and code review validation.

An ideal version of `PolicyChecker` will support all three types of the code review policy. However, our current implementation can handle the first two categories (*i.e.*, Standard and Semi-customized rules) and a limited number of customized rules on Gerrit. Now we discuss how `PolicyChecker` interprets different code review policies on GitHub and Gerrit.

GitHub. We categorize GitHub's code review policy as semi-customized rules since it provides a limited set of options to define the code review policy – it does not offer any predefined rules nor allow to define an arbitrary code review policy. `PolicyChecker` interprets the code review policy on GitHub by extracting the following information:

(1) branch protection rules (a set of policies to perform the code review process); (2) collaborators (a list of all project members as well as the permissions granted to each member); (3) code owners (a specific group of users who must approve a code change). Some rules (*i.g.*, who can dismiss pull request reviews, reviews from code owners) have higher priorities. In Section 4.5.2, we describe our implementation to evaluate different code review rules on GitHub.

Gerrit. Gerrit allows the project owners to create a code review policy by using predefined rules or by adding new customized rules. Therefore, the code review policy on Gerrit can fall into any of three categories. `PolicyChecker` interprets Gerrit's code review policy by extracting the following information: (1) submit rules (a set of policies to perform the review process, which resides on `project.config` and `rules.pl` files); (2) groups (a list of groups, their members, as well as the permissions granted to each member). In the case of predefined or customized rules defined through the Web UI (*i.e.*, semi-customized rules), the `rules.pl` file will be empty. Therefore, the code review policy is interpreted by extracting information only from the `project.config` and `groups` files. Otherwise, `PolicyChecker` must deal with arbitrary rules defined in the `rules.pl` file using the Prolog language [17]. We can think of two approaches to handle all Gerrit customized rules: (1) implement a light Prolog interpreter dedicated to Gerrit rules; (2) integrate `PolicyChecker` with an available Prolog interpreter (*e.g.*, SWI-Prolog [46], PySwip [42], Gerrit Prolog Shell [86]);

We have worked on these approaches; however, our Prolog interpreter is still a work in progress. The current version of the `PolicyChecker` [41] can handle an important set of customized rules as follows:

- the review score range,
- users that can bypass the code review process,
- users that can never add code changes,

- if change can be merged with unresolved comments,
- if a specific file can be changed,
- if a specific user committed the change,
- the submit type.

A summary of our efforts towards adding a Prolog interpreter into `PolicyChecker` is described in the Appendix.

PROCEDURE:PolicyChecker
Input: Branch, ReviewPolicy, Server
Output: `success/fail`

```
1: if Validate_Signature(ReviewPolicy) == false then
2:   // The review policy does not have a valid signature
3:   return fail
4: end if
5: C ← The set of all commits in the Branch
6: h ← The head commit of the Branch
7: // Check if all the branch commits have valid signatures
8: for all c ∈ C do
9:   if Validate_Signature(c) == false then
10:    return fail
11:  end if
12: end for
13: while (C is not empty) do
14:   // Extract the commits in the merge request that
15:   // corresponds to h
16:   MRC ← Extract_Merge_Request_Commits(h)
17:   // Check if the sequence of reviews embedded in the
18:   // merge request is valid against the review policy
19:   if Validate_Reviews(MRC, ReviewPolicy, Server) == false then
20:     return fail
21:   end if
22:   // Remove commits in merge request from the set C,
23:   // and find the head of remaining commits
24:   C ← C \ MRC
25:   h ← The head commit of the branch represented by commits left in C
26: end while
27: return success
```

PROCEDURE: Validate_Reviews_GitHub
Input: Commits, ReviewPolicy
Output: `success/fail`

```
 1: Reviews ← getCodeReviews(Commits)
 2: // Check if code reviews have valid signature and the chain of code reviews is valid.
 3: if Validate_Review_Signatures(Reviews, ReviewPolicy) == false then
 4:   return fail
 5: end if
 6: // Check if the merger has the permission
 7: if AuthorizedMerger(Commits, ReviewPolicy) == false then
 8:   return fail
 9: end if
10: ru ← length(Reviews)
11: if ru == 0 and FirstCommit(Commits) == false then
12:   // Check if the committer has the direct push access
13:   if DirectPush(Commits, ReviewPolicy) == false then
14:     // Unauthorized direct push is detected
15:     return fail
16:   end if
17: else
18:   rules ← inspectReviewPolicy(ReviewPolicy)
19:   // Check if reviews created by authorized reviewers
20:   if AuthorizedReviewers(Reviews, rules) == false then
21:     // Unauthorized reviewer is detected
22:     return fail
23:   end if
24:   // Check if there are required reviews from specific users
25:   if RequiredReviews(Reviews, rules) == false then
26:     return fail
27:   end if
28:   // and if the minimum number of approving reviews is met
29:   if MinApprovals(Reviews, rules) == false then
30:     return fail
31:   end if
32: end if
33: return success
```

4.4.3 Validate the Code Reviews

An auditor can verify the integrity of the code review process for each branch in the repository by running `PolicyChecker`. It first checks if the code review policy has a valid signature (lines 1-2). Then, it validates the commit signatures (lines 7-9). Next, it traverses the commit tree in the branch and extracts the commits corresponding to a merge request by calling the Extract_Merge_Request_Commits procedure (line 16). Finally, the Validate_Reviews procedure attests if the merge request was properly approved according to the code review policy (line 19). In general, this procedure performs three major checks: (1) Validate the commit signatures (2) Validate the code review signatures (3) Check whether the sequence of code reviews led to approve the merge request respects the intended code review policy.

GitHub. The Validate_Reviews_GitHub procedure receives as input a valid signed review policy and a set of commits corresponding to a merge request. It first checks if the code reviews have valid signatures and the chain between code reviews is valid (lines 1-5). Then it checks if the merger has permissions to perform the merge (lines 7-9). Next, it finds the number of code reviews embedded in the merge request. If there are no reviews, it checks if the merger has permissions to directly merged the code changes – the merger must be an administrator and the code review policy must exclude the administrator from the code review policy (lines 12-16). Otherwise, the Validate_Reviews_GitHub procedure performs the following checks: if reviews are created by authorized reviewers – the reviewers are granted the read access (lines 23-25); if there are approving reviews from code owners (lines 26-28); if the sequence of code reviews meets the minimum number of approving reviews defined by the policy, such that only those approving reviews that satisfy the policy are counted (lines 30-32). More details on checks for the minimum number of approvals are provided in the implementation, Section 4.5.2.

Gerrit. Unlike our GitHub version, the Validate_Reviews_Gerrit procedure must validate the code review process against all three types of policies (*i.e.*, standard, semi and customized rules). It receives as input a valid signed review policy and a set of commits corresponding to the code change, and checks whether the change was merged according to the intended code review policy.

PROCEDURE: Validate_Reviews_Gerrit
Input: Commits, ReviewPolicy
Output: `success/fail`

```
Reviews ← getCodeReviews(Commits)
// Check if code reviews have valid signature and the chain of code reviews is valid.
if Validate_Review_Signatures(Reviews, ReviewPolicy) == false then
  return fail
end if
// Check if the merger has the permission
if AuthorizedMerger(Commits, ReviewPolicy) == false then
  return fail
end if
ru ← length(Reviews)
if ru == 0 and FirstCommit(Commits) == false then
  // Check if the committer has the direct push access
  if DirectPush(Commits, ReviewPolicy) == false then
    return fail
  end if
else
  rules ← inspectReviewPolicy(ReviewPolicy)
  if isCustomizedRules(rules) == false then
    // Check if the author of code was authorized
    if AuthorizedAuthor(Commits, ReviewPolicy) == false then
      return fail
    end if
    // Check if reviews created by authorized reviewers
    if AuthorizedReviewers(Reviews, rules) == false then
      return fail
    end if
    // Check if the blocked reviews are authorized
    if isAllowedBlock(rules, Reviews) == false then
      return fail
    end if
    // Check if the approving reviews are authorized
    if isAllowedApprove(rules, Reviews) == false then
      return fail
    end if
    // Check if the minimum number of approving reviews is met
    if MinApprovals(rules, Reviews) == false then
      return fail
    end if
  else
    // Check if customized rules are followed correctly
    if CustomizedRules(rules) == false then
      return fail
    end if
  end if
end if
return success
```

After validating the code review signatures, and the chain between code reviews is valid (lines 1-5), it checks if the merger has permissions to perform the merge (lines 7-9); if there is a direct push, the committer has direct push access (lines 11-15). In the case of no customized rules, it checks if the author of the code was allowed to submit a merge request (lines 20-22); if reviews are created by authorized reviewers (lines 24-26); if block reviews (*i.e.*, lowest scores) are given by authorized users (lines 28-30); if approving reviews (*i.e.*, highest scores) are given by authorized users (lines 32-34); if the sequence of code reviews meets the minimum number of approving reviews defined by the policy, such that only those approving reviews that satisfy the policy are counted, and outdated approving reviews are dismissed correctly (lines 36-38). In the case of customized rules, `PolicyChecker`performs additional checks, which are detailed in Section 4.5.2.

4.4.4 Extract the Code Reviews

`PolicyChecker` extracts all code reviews related to a merge request by determining the boundary of a merge request – finding all commits that belong to a merge request. At a high level, `PolicyChecker` performs three steps to get the code reviews: (1) finding the merge strategy, (2) finding the number of commits per merge request, (3) extracting merge request's commits.

GitHub. The Extract_Merge_Request_Commits_GitHub procedure first finds the number of the commit's parents (line 2). If there are no parents, the repository's first commit is found (lines 3-5). If the commit has two parents, the merge strategy is "Merge" and all merge request's commits are extracted by calling the getPRCommits procedures (line 29). In one parent's case, the procedure extracts the code reviews embedded in the commit (line 12). Then, it detects the merge strategy based on the number of code reviews embedded in the commit.

PROCEDURE: Extract_Merge_Request_Commits_GitHub
Input: Commit
Output: MergeRequestCommits

```
 1: MRC ← {}
 2: p ← The number of Commit's parents
 3: if p == 0 then
 4:   // The first commit in the repository
 5:   return MRC, "FirstCommit"
 6: end if
 7: // Set the default merge strategy to "Merge"
 8: MergePolicy ← "Merge"
 9: if p == 1 then
10:   MRC ← MRC ⋃ Commit
11:   // Extract the code reviews embeded in the Commit
12:   CodeReviews ← getCodeReviews(Commit)
13:   n ← The number of CodeReviews
14:   if n == 0 then
15:     // A direct push
16:     MergePolicy ← "DirectPush"
17:   else if n == 1 then
18:     if isFirstReview(CodeReviews) == false then
19:       // A rebase commit
20:       MergePolicy ← "Rebase"
21:       MRC ← MRC ⋃ getRebaseCommits(Commit)
22:     end if
23:   else
24:     // A squash and merge commit
25:     MergePolicy ← "SquashAndMerge"
26:   end if
27: else
28:   // Extract commits for a merge commit
29:   MRC ← MRC ⋃ getPRCommits(Commit)
30: end if
31: return MRC, MergePolicy
```

As a result, there are three options: (1) The commit has no code reviews, which means the commit was pushed directly to the codebase without being reviewed (lines 13-16); (2) The commit has only one code review; therefore, the merge request's commits are extracted by calling the getRebaseCommits procedures (line 17-22); (3) The commit has more than one code review, which means two or more commits are squashed. Thus, the merge strategy is "SquashAndMerge" (line 23-26).

PROCEDURE: Extract_Merge_Request_Commits_Gerrit
Input: Commit
Output: MergeRequestCommits

```
1: p ← The number of Commit's parents
2: if p == 0 then
3:   // The first commit in the repository
4:   return
5: else if p == 1 then // One commit is added to the codebase
6:   return Commit
7: else// Two commits are added to the codebase
8:   return {Commit, Commit's second parent}
9: end if
```

Gerrit. `PolicyChecker` extract the merge request commits on Gerrit by leveraging the fact that the number of commits in each merge request is at most two. Actually, the first commit is always the latest patch set in the change, and the second one is an additional merge commit which is created due to the merge strategy. As a result, the Extract_Merge_Request_Commits_Gerrit procedure is simpler than the GitHub version. `PolicyChecker` first checks the number commit's parents (line 1), In one parent's case, the only commit in the merge request is the current commit (lines 5-6). Otherwise, there are two commits in the merge request: the current commit, and its second parent corresponding to the latest patchset in the code change.

4.5 Implementation

To validate the code review process performed on a GitHub or Gerrit repository, the auditor must clone the repository on a local machine and then run the `git validate-reviews` command. With `PolicyChecker` in place, an independent auditor can verify the integrity of the code review process if she has the code reviews embedded in the source code repository along with the signed code review policy. We implemented `PolicyChecker` as a git command. The current version consists of a total of 1,108 lines of Python code and has been released as free and open-source software [41].

Given four parameters, `PolicyChecker` can validate the code review process: branch (the repository branch name), format (the code review format), key (the path to the public keys), and CRP (the code review policy)[2]. Once `PolicyChecker` interprets the code review policy (as described in Section 4.4.2), it can perform the validation process through three major steps:

- **Extract the Code Reviews**: Parse the sequence of Git commits to find all code reviews for each merge request.
- **Validate the Code Review Process**: Perform two checks: (1) The integrity and the authenticity of the code review policy, Git commits, and code reviews. (2) A sequence of code reviews that led to merge a code change respects the intended code review.

In the rest of this section, we describe how we perform each of these three step for GitHub and Gerrit.

4.5.1 Extract the Code Reviews

To extract code reviews from a repository, we first detect the boundary of each merge request – find the first and the last commit in the merge request. For which, we evaluate three things: the merge strategy, the number of parents for the head of the branch, the number of code reviews embedded in the head of the branch. Below, we explain how these three pieces of information help to reach our goal.

GitHub. We examine the head of the branch to find the number of parents of the branch's head. Depending on the number of parents, we can extract all commits (and code reviews) in one merge request as follows.

- Two parents: The *Merge* strategy is used to integrate the merge request. Thus, we first the common ancestor of two parents (*i.e.*, the parent of the first commit in the merge request branch). Then any commit from the common ancestor to the merge request branch's head is considered the merge request's commits.

[2]If the code review policy is not provided, `PolicyChecker` can be instructed to download the code review policy from the server. Besides, `PolicyChecker` helps the project owners to create and sign a code review policy. That could help project owners that tend to share the code review policy through an offline channel.

- One parent: The merge strategy is either *Squash and Merge* or *Rebase.* If the head commit contains more than one embedded code review, the merge strategy is *Squash and Merge.* Because each commit can not have more than one review unit unless the commits in the merge request branch are squashed together in one commit. If there is only one review unit embedded in the head commit, we determine the merge strategy by checking the commit's timestamp where the same timestamp for author and committer means the *Squash and Merge* strategy.

Gerrit. In the case of Gerrit, the code review extraction is quite easier than GitHub. On Gerrit, any modification to the merge request (*i.e.*, code change) does not result in a brand new commit since each commit amends the previous one in the merge request branch. Thus the head of the merge request branch is equivalent to the entire branch. That said, merging a code change on Gerrit results in one or at most two new commits: the head of the merge request branch and a new merge commit (when the *Merge Always* or *Merge If Neccessary* is in place).

Once we extract commits for the latest merge request, we repeat the above procedure for the remaining commits in the branch until we extract code reviews for all merge requests. Worthy of note that if we find a merge request with no code reviews (*i.e.*, a *Direct Push*), we conclude that the corresponding code changes got merged without going through the code review process. We will later validate these commits to ensure that the committer has the permission to push commits with no reviews (*i.e.*, permission to perform a *Direct Push*).

4.5.2 Validate the Code Review Process

To validate the code review process, `PolicyChecker` first checks the integrity and the authenticity of all inputs. It then checks code reviews against the rules – ensure if all code changes followed the intended code review policy. Note that the former step is the same for both Gerrit and GitHub. However, in the second step, the details can vary greatly depending on many factors such as the code review workflows, the system settings, and code review policies.

4.5.2.1 Input Validation.

`PolicyChecker` verifies digital signatures for the inputs (*i.e.*, code review policy, Git commits, and code reviews) as follows.

- **Code review policy**: The code review policy's signature is validated using the project owner's public key. We assume that the code review policy is signed by the project owner and is accessible to any independent auditor. In case that the code review policy is not provided locally, the auditor may instruct `PolicyChecker` to retrieve the code review policy and its signature from the server.
- **Git commits**: The GPG signatures on commits are verified using the "`git verify-commit`" command.
- **Code reviews**: The integrity of code reviews is validated using the GPG signatures for individual code reviews and the chaining between code reviews (*i.e.*, a sequence of code reviews embedded in a merge request). Indeed, `PolicyChecker` checks that each code review includes a valid GPG signature computed over the previous code review in the merge request. That helps to ensure that unauthorized code review changes did not happen in the code review process.

4.5.2.2 Checking Code Reviews Against Rules.

Recall from 4.4.2 that `PolicyChecker` extracts different components of the code review policy on GitHub and Gerrit (*i.e.*, find the review rules and user permissions). It then performs multiple checks to ensure that the review process was followed as intended by the project owner.

GitHub. `PolicyChecker` extracts protection rules, code owners, and collaborators to perform the following checks:

- Authorized Merge Option: If the code review policy requires a linear history, merge commits can not be pushed to matching branches. Therefore, a merge request can be merged only under the "squash and merge" or "rebase" merge strategy.
- Authorized Merger: The merge committer must have the "write" permission to the branch. However, if the protection rules restrict who can push to the branch, the committer must be among some specified users who can push into the branch.

- Authorized Author: The author of any code changes in the merge request must have "read" permission to the branch.
- Authorized Reviewer: The author of any code review must have the "read" permission to the branch.
- Authorized Direct Push: If the merge request was merged directly, the merge committer must have the "admin" permission to the repository and the "push" permission to the branch. Finally, the protection rules must exclude the administrators from following the code review process.
- Minimum Approvals: If the protection rules ask for ignoring the stale reviews, any reviews before the latest code change in the merge request will be dismissed. The remaining reviews must satisfy the minimum number of approving reviews from the code owners.

Gerrit. `PolicyChecker` first performs the following checks for Gerrit repositories:

- Authorized Merge Option: The code changes must be merged based on the authorized "submit type" as described in Section 4.2.3.
- Authorized Merger: The merge committer must be a member of a group who has the "submit" permission to the branch.
- Authorized Author: The author of any code changes in the merge request must have the "push" permission to the "pending changes" location (*i.e.*, "refs/for/refs" namespace).
- Authorized Reviewer: The author of any code review must have "Code-Review" permission on the branch. By default, she must be either a project owner, an administrator, or a registered user.
- Authorized Direct Push: If the code change was merged without going through the review process, the committer must have "push" permission to the branch.
- Valid Scores: Any score must come from an authorized reviewer. The max positive and max negative scores can be given only by users who have the "Code-Review" permission with the highest and the lowest scores. By default, any score must range from -2 to +2. Finally, the highest (+2) and the lowest (-2) scores must be given by administrators or project owners.
- Submittable Merge: If there are no customized rules (*i.e.*, the rules.pl file is empty), the submit rule is determined by rules extracted the `project.config` and `groups` files. Otherwise, `PolicyChecker` performs checks for customized rules extracted from the `rules.pl` file. In the former case, `PolicyChecker` follows Gerrit internal workflow [19]. We first capture votes only on the latest patchset and then checks for:

– Authorized Review: Check if the code review is required.
– Authorized Block: Check if giving the lowest score is authorized, and the reviewer is allowed to give the lowest score (*i.e.*, block the change).
– Authorized Block: Check if giving the highest score is authorized, and the reviewer is allowed to give the highest score (*i.e.*, approve the change).

We note that the procedure depicted above allows `PolicyChecker` to handle default, semi-customized, and an important set of customized rules (as described in Section 4.4.2).

4.6 Evaluation

In this section, we evaluate the performance of `PolicyChecker` by investigating the time needed to verify the code review process on a Git repository. For our benchmark, we picked five popular repositories from GitHub [3] and five popular repositories from Gerrit Google Source [21]. We chose repositories of different history sizes, file counts, and file sizes (as shown in Table 4.1[4]). For each repository, we show the size of the master branch, the number of files, the average file size, and the number of commits for each repository. We conducted our experiments on a client system with an Intel Corei7 CPU at 2.70 GHz and 16 GB RAM. The client software consisted of Linux 5.3.7-301.fc31.x86_64 with git 2.23.

We measured the verification time for one merge request and one software release. In the former case, we chose a merge request of four commits with three code reviews. In the latter case, we picked 14 merge requests (each of four commits and three three code reviews). In both cases, we measured the average time over 30 independent runs. Speaking of the code review policy, we chose the default code review policy for Gerrit repositories (*i.e.*, a code change is mergeable if it receives at least a review with the highest score and no reviews have the lowest score). For

[3]Popularity is based on the "star" ranking by GitHub users, which reflects their level of interest in a project as of April 1, 2021.
[4]The top five are GitHub repositories, and the bottom five are Gerrit repositories.

Table 4.1 Repositories Chosen for the `PolicyChecker`'s Evaluation

Repository	Size (MB)	File Count	File Size (Bytes)	History Size (# of commits)
javascript	4	55	6,806	1,867
puppeteer	15	413	6,260	2,109
deno	67	1,375	20,189	5,344
terminal	148	3,257	13,854	2,116
django	279	6,453	10,423	29,431
gitfs	1	43	4,078	123
reviewit	7	529	1,902	84
homepage	26	262	27,949	1,224
jgit	55	2,514	5,888	8,289
gerrit	343	5,043	6,837	49,092

GitHub repositories, we assumed a code review policy in which a merge request is mergeable if it receives two approving reviews from code owners, one from another authorized reviewer and stale reviews are dismissed. Finally, we assumed that the code review policy and the public keys are provided at the verification step. We point out that our experimental setup is driven by our findings of the top 50 popular GitHub repositories: (1) the average number of commits per release is 68.7 commits, (2) the average number of commits per merge request is 3.3 commits. (3) the number of reviews per merge request is often less than 2, very rarely more than 3.

Table 4.2 shows the execution time to validate the code review process using `PolicyChecker`. Regardless of the repository size or the repository structure, on average, it takes 0.11 seconds to verify one merge request. However, the execution time to validate a software release (*i.e.*, 68 commits) varies from 1.38 to 1.62 seconds (with an average of 1.49 seconds). Four operations influence the execution time:

Table 4.2 Execution Time for Validating Code Review Process (in seconds)

Repository	One Merge Request	One Software Release
javascript	0.11	1.53
puppeteer	0.11	1.40
deno	0.10	1.43
terminal	0.10	1.52
django	0.11	1.60
gitfs	0.11	1.43
reviewit	0.11	1.38
homepage	0.11	1.59
jgit	0.11	1.44
gerrit	0.12	1.62

(1) traverse the repository to find all commits involved in a merge request or a software release; (2) read the commit objects to eaxtract necessary information (*i.e.*, author, committer, commit message, commit signature); (3) extract the review units from the commit messages; (4) verify the commits' signatures, the code reviews' signatures, and the review chain. From which the first and last operations play a major role in the verification. Since the number of signature verifications is the same for all experiments, the time to traverse a repository causes different execution times. That is why the execution time to verify a software release (which includes traversing the commit history) takes a bit more time (*i.e.*, 0.20 seconds on average) on a huge repository such as gerrit and djando. On the other side, validating just one merge request exhibits similar performance on different repositories.

4.7 Related Work

The code review process has been largely overlooked by security researchers. A wast majority of efforts have focused on finding security related weaknesses (vulnerabilities) [39, 127] in the source code rather than securing the code review process itself. To the best of our knowledge, `PolicyChecker` is the first tool that attempts to verify the correctness of the code review process automatically. In this section, we briefly study the relevant work to the secure code review process.

A variety of code review systems (such as GitHub [23], GitLab [28], Gerrit [16], Bitbucket [4], Collaborator [9], Crucible [11], ReviewBoard [43], Rhodecode [44], Helix Swarm [30], gitea [22], CodeFlow [7], Visual Expert [49], and Upsource [48]) have helped developers to improve the quality of the code reviews by automating this process – reducing the amount of time developers must spend to review a code change. Moreover, services like GitHub provide seamless integrations with additional tools (*e.g.*, DeepSource [12] and AccessLint [1]) to find and fix bugs with lower inspection effort [91]. In particular, vulnerability scanners such as GitHub code scanning tools [92] allows developers to find security vulnerabilities during the code review process.

Several previous studies explored approaches that recommend proper code reviewers for code changes. Work by Yu *et al.* [168] looks into the history of reviewers on GitHub (*e.g.*, the time of comments by each reviewer) to propose a comment networks (*i.e.*, CN-based) approach which recommends reviewers for new pull requests on GitHub. A similar work by Balachandran [59] relies on automatic static analysis and reviewer recommendation to propose a software defect detection technique. RevFinder [158] helps developers finding appropriate code reviewers for their projects. Fistbump [113] and proposes a web-based tool integrated with GitHub to facilitate discussion management and issue tracking during the code review process.

Another research avenue to improve the quality of the code reviews focuses on secure coding. Work by Yang [167] proposes a vulnerability prediction tool based on the CERT-C Secure Coding Standard aiming to eliminate security vulnerabilities. Kang *et al.* [114] present a secure-coding checking system to reduce the weaknesses of open source software products using white box and black box testing and smart fuzzing technology. Finally, there have been many efforts such as OWASP Code Review Guide [39] and MITRE Secure Code Review [127] to standardize a set of secure coding guidelines that helps developers identifying security bugs early in the software development.

CHAPTER 5
CONCLUSION

In this dissertation, we enhanced the security of several individual steps in the software supply chain. In Chapter 2, we revealed novel attacks that can be performed stealthily in conjunction with several common web UI actions on GitHub. Common to all these attacks is the fact that commits created by the server do not reflect the user's actions. The impact can be significant, such as removing a security patch, introducing a backdoor, or merging experimental code into a production branch.

To counter these attacks, we devised `le-git-imate`, a defense scheme that provides security guarantees comparable and compatible with Git's standard commit signing mechanism. With our solution in place, users can take advantage of GitHub's web-based features without sacrificing security. `le-git-imate` does not require any changes on the server side and can be used today with existing web UI deployments. Our experimental evaluation and user study show that `le-git-imate` incurs a reasonable performance overhead and presents a minimal usability burden to Git web UI users.

`le-git-imate`'s current design provides limited protection against web UI attacks. As future work, we plan to develop a more comprehensive defense mechanism against UI attacks, especially through a more tight integration with the provider of the web-based Git repository hosting service. Adapting `le-git-imate` to other web-based repository hosting services will require a degree of manual work that depends on the specifics of the service's UI; however, we found that the same general principles used for GitHub/GitLab are applicable to a wide variety of similar services.

In Chapter 3, we introduced `SecureReview`, a mechanism that can be applied on top of code review systems to provide verifiable guarantees about the integrity of the code review process. `SecureReview` lays the foundations for securing the code review step, but more future work is needed. In particular, some code review systems allow the server to automatically merge changes, which raises additional security concerns. Project owners may be reluctant to publicize code review information due to privacy concerns. `SecureReview` can be extended to replace the review content with a hash of the review (salted to prevent dictionary attacks), and only reveal the content on demand.

In Chapter 4, we introduced `PolicyChecker`, a tool that enables automatic verification of the code review process in web-based core review systems such as GitHub and Gerrit. This tool can be useful prior to and after the code merging step to check if the code review process followed the intended code review policy. Finally, more work is needed to support arbitrarily customized review policies. In particular, we plan to integrate a Prolog interpreter into `PolicyChecker` to support all customized rules on the Gerrit server.

APPENDIX
PROLOG INTERPRETER: A WORK IN PROGRESS

An ideal version of `PolicyChecker` can handle arbitrary code review policies for Gerrit repositories. Arbitrary policies can be defined in `rules.pl` using the Prolog language [17]. As detailed in Section 4.4.2, the current version of `PolicyChecker` can handle a limited set of customized rules on Gerrit. In order to implement the ideal version of `PolicyChecker`, we have explored two approaches: (1) implement a light Prolog interpreter dedicated to Gerrit rules; (2) integrate `PolicyChecker` with existing Prolog interpreters.

Implement a light Prolog interpreter. We first implemented a simple Prolog interpreter with two basic operators: (1) remove comment lines, (2) extract facts and rules. Then we improved our interpreter to create a database of rules and facts using Gerrit's default code review policy (`groups` and `project.config`). As future work, we want to convert code reviews and code review policies to Prolog queries on our database.

Integrate `PolicyChecker` with existing Prolog interpreters. We have worked on an integration with PySwip [42], a Python interface that enables us to query SWI-Prolog [46] in a Python program. For which, we first created a simple database and performed basic Prolog queries. Then we implemented an interface to easily work with the PySwip library. That helped us to create the database of code review rules in Prolog and then to perform queries in the database. This functionality must be extended to (1) import any customized Gerrit rules into the Prolog database, (2) perform queries with complex and advanced Prolog syntax.

REFERENCES

[1] *AccessLint.* https://github.com/marketplace/accesslint, last accessed on 04/01/2021.

[2] *Apso - Summary.* https://savannah.nongnu.org/projects/apso, last accessed on 04/01/2021.

[3] *Assembla.* https://www.assembla.com, last accessed on 04/01/2021.

[4] *Bitbucket.* https://bitbucket.org, last accessed on 04/01/2021.

[5] *Chrome Extensions.* https://developer.chrome.com/docs/extensions/, last accessed on 04/01/2021.

[6] *Code Reviews at Google.* https://www.michaelagreiler.com/code-reviews-at-google/, last accessed on 04/01/2021.

[7] *CodeFlow.* https://codeflow.co, last accessed on 04/01/2021.

[8] *Codereviewhub.* https://www.codereviewhub.com/, last accessed on 04/01/2021.

[9] *Collaborator.* https://smartbear.com/product/collaborator/overview, last accessed on 04/01/2021.

[10] *CrowdStrike. Securing the supply chain.* https://www.crowdstrike.com/resources/wp-content/brochures/pr/CrowdStrike-Security-Supply-Chain.pdf, last accessed on 04/01/2021.

[11] *Crucible.* https://www.atlassian.com/software/crucible, last accessed on 04/01/2021.

[12] *DeepSource.* https://github.com/marketplace/deepsource-io, last accessed on 04/01/2021.

[13] *es-git.* https://github.com/es-git/es-git, last accessed on 04/01/2021.

[14] *Flask.* http://flask.pocoo.org, last accessed on 04/01/2021.

[15] *FlowCrypt.* https://flowcrypt.com, last accessed on 04/01/2021.

[16] *Gerrit.* https://www.gerritcodereview.com, last accessed on 04/01/2021.

[17] *Gerrit Code Review - Prolog Submit Rules.* https://gerrit-review.googlesource.com/Documentation/prolog-cookbook.html, last accessed on 04/01/2021.

[18] *Gerrit Code Review - REST API.* https://gerrit-review.googlesource.com/Documentation/rest-api.html, last accessed on 04/01/2021.

[19] *Gerrit default sumit rules.* https://github.com/GerritCodeReview/gerrit/blob/master/java/com/google/gerrit/server/rules/DefaultSubmitRule.java, last accessed on 04/01/2021.

[20] *GerritHub.* http://gerrithub.io, last accessed on 04/01/2021.

[21] *Git repositories on gerrit.* https://gerrit.googlesource.com/, last accessed on 04/01/2021.

[22] *gitea.* https://github.com/go-gitea/gitea, last accessed on 04/01/2021.

[23] *GitHub.* https://github.com, last accessed on 04/01/2021.

[24] *The GitHub Blog.* https://github.com/blog, last accessed on 04/01/2021.

[25] *GitHub Enterprise 2.1.0 Update Released.* https://enterprise.github.com/releases/2.1.0/notes, last accessed on 04/01/2021.

[26] *git.js.* https://github.com/danlucraft/git.js, last accessed on 04/01/2021.

[27] *gitkit-js.* https://github.com/SamyPesse/gitkit-js, last accessed on 04/01/2021.

[28] *GitLab.* https://gitlab.com, last accessed on 04/01/2021.

[29] *Gogs.* https://gogs.io, last accessed on 04/01/2021.

[30] *Helix Swarm.* https://www.perforce.com/products/helix-swarm, last accessed on 04/01/2021.

[31] *Horde Groupware contains backdoor.* http://www.h-online.com/security/news/item/Horde-Groupware-contains-backdoor-1433972.html, last accessed on 04/01/2021.

[32] *How Gerrit Works.* https://gerrit-review.googlesource.com/Documentation/intro-how-gerrit-works.html, last accessed on 04/01/2021.

[33] *isomorphic-git.* https://isomorphic-git.org, last accessed on 04/01/2021.

[34] *Jira.* https://www.atlassian.com/software/jira, last accessed on 04/01/2021.

[35] *js-git.* https://github.com/creationix/js-git, last accessed on 04/01/2021.

[36] *Keybase.* https://keybase.io, last accessed on 04/01/2021.

[37] *Mailvelope.* https://www.mailvelope.com/en, last accessed on 04/01/2021.

[38] *OpenPGP.js.* https://openpgpjs.org, last accessed on 04/01/2021.

[39] *OWASP Code Review Guide.* https://owasp.org/www-project-code-review-guide/, last accessed on 04/01/2021.

[40] *Phabricator.* https://www.phacility.com, last accessed on 04/01/2021.

[41] *PolicyChecker.* `https://github.com/thesecurereview/policychecker`, last accessed on 04/01/2021.

[42] *PySwip.* `https://github.com/yuce/pyswip`, last accessed on 04/01/2021.

[43] *ReviewBoard.* `https://www.reviewboard.org`, last accessed on 04/01/2021.

[44] *RhodeCode.* `https://rhodecode.com`, last accessed on 04/01/2021.

[45] *SourceForge.* `https://sourceforge.net`, last accessed on 04/01/2021.

[46] *SWI Prolog.* `https://www.swi-prolog.org/`, last accessed on 04/01/2021.

[47] *The Heartbleed Bug.* `http://heartbleed.com`, last accessed on 04/01/2021.

[48] *Upsource.* `https://www.jetbrains.com/upsource/`, last accessed on 04/01/2021.

[49] *visual-expert.* `https://www.visual-expert.com/`, last accessed on 04/01/2021.

[50] *What Are Dark Patterns?* `https://darkpatterns.org`, last accessed on 04/01/2021.

[51] Catalin Cimpanu. *Hacker gains access to a small number of Microsoft's private GitHub repos.* `https://www.zdnet.com/article/hacker-gains-access-to-a-small-number-of-microsofts-private-github-repos/`, last accessed on 04/01/2021.

[52] Lillian Ablon and Andy Bogart. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits.* Rand Corporation, Santa Monica, California, 2017.

[53] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.

[54] Hammad Afzali, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos. le-git-imate: Towards verifiable web-based git repositories. In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS 18)*, pages 469–482, 2018.

[55] Hammad Afzali, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos. Towards adding verifiability to web-based git repositories. *Journal of Computer Security*, 28(4):405–436, 2020.

[56] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 103–112, 2014.

[57] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, et al. DROWN: Breaking TLS Using SSLv2. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, 2016.

[58] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the IEEE International Conference on Software Engineering*, page 712–721, 2013.

[59] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 35th IEEE International Conference on Software Engineering (ICSE)*, pages 527–534, 2013.

[60] Karthikeyan Bhargavan and Gaetan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on http over tls and openvpn. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467, 2016.

[61] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[62] Amy Blackshaw. *Kingslayer-A Supply Chain Attack*. `https://www.rsa.com/en-us/blog/2017-02/kingslayer-a-supply-chain-attack`, last accessed on 04/01/2021.

[63] Amiangshu Bosu and Jeffrey C Carver. Peer code review to prevent security vulnerabilities: An empirical evaluation. In *Proceedings of the 7th IEEE International Conference on Software Security and Reliability Companion*, pages 229–230, 2013.

[64] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 257–268, 2014.

[65] Edmund Brumaghin, Ross Gibb, Warren Mercer, Matthew Molyett, and Craig Williams. *CCleanup: A Vast Number of Machines at Risk*. `https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html`, last accessed on 04/01/2021.

[66] Ramaswamy Chandramouli, Michaela Iorga, and Santosh Chokhani. *Cryptographic Key Management Issues & Challenges in Cloud Services.(National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 7956*, 2013. `https://doi.org/10.6028/NIST.IR.7956`, last accessed on 04/01/2021.

[67] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual ec incident. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 468–479. ACM, 2016.

[68] Anton Cherepanov. *Analysis of TeleBots' cunning backdoor.* `https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor`, last accessed on 04/01/2021.

[69] Sonia Chiasson, Alain Forget, Robert Biddle, and Paul C Van Oorschot. User interface design affects security: Patterns in click-based graphical passwords. *International Journal of Information Security*, 8(6):387, 2009.

[70] Chrome. *Content Scripts.* `https://developer.chrome.com/extensions/content_scripts`, last accessed on 04/01/2021.

[71] Chrome. *Manage Events with Background Scripts.* `https://developer.chrome.com/extensions/background_pages`, last accessed on 04/01/2021.

[72] Catalin Cimpanu. *Two malicious Python libraries caught stealing SSH and GPG keys.* `https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/`, last accessed on 04/01/2021.

[73] Barb Darrow. *Adobe source code breach; it's bad, real bad.* `https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad/`, last accessed on 04/01/2021.

[74] DBpedia. *About: Keybase.* `https://dbpedia.org/page/Keybase`, last accessed on 04/01/2021.

[75] Thai Duong and Juliano Rizzo. Here come the$\oplus$ ninjas. *Unpublished manuscript*, 320, 2011.

[76] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion in code reviews: Reasons, impacts, and coping strategies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 49–60, 2019.

[77] eSecurity Solutions. *Are Software Supply Chain Attacks Replacing Zero Day?* `https://www.esecuritysolutions.com/software-supply-chain-attacks-v-zero-day`, last accessed on 04/01/2021.

[78] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, and Uwe Sander. Helping johnny 2.0 to encrypt his facebook conversations. In *Proceedings of the 8th Symposium on Usable Privacy and Security*, pages 1–17, 2012.

[79] Nargis Fatima, Suriayati Chuprat, and Sumaira Nazir. Challenges and benefits of modern code review-systematic literature review protocol. In *Proceedings of the IEEE International Conference on Smart Computing and Electronic Enterprise (ICSCEE)*, pages 1–5, 2018.

[80] FireEye. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor.* `https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html`, last accessed on 04/01/2021.

[81] Benjamin Fogel, Shane Farmer, Hamza Alkofahi, Anthony Skjellum, and Munawar Hafiz. Poodles, more poodles, freak attacks too: how server administrators responded to three serious web vulnerabilities. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pages 122–137. Springer, 2016.

[82] Brian Fox. *Open Source Software Is Under Attack; New Event-Stream Hack Is Latest Proof.* `https://blog.sonatype.com/open-source-software-is-under-attack-new-event-stream-hack-is-latest-proof`, last accessed on 04/01/2021.

[83] Dennis F Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 5(1):1, 2004.

[84] Gerrit. *Gerrit Customized Labels.* `https://gerrit.opencord.org/Documentation/config-labels.html#label_custom`, last accessed on 04/01/2021.

[85] Gerrit. *Labels.* `https://gerrit.opencord.org/Documentation/intro-project-owner.html#labels`, last accessed on 04/01/2021.

[86] Gerrit. *Prolog Shell.* `https://gerrit-review.googlesource.com/Documentation/pgm-prolog-shell.html`, last accessed on 04/01/2021.

[87] Mike Gerwitz. *A Git Horror Story: Repository Integrity With Signed Commits.* `https://mikegerwitz.com/2012/05/a-git-horror-story-repository-integrity-with-signed-commits`, last accessed on 04/01/2021.

[88] Git-SCM. *Signing Your Work.* `https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work`, last accessed on 04/01/2021.

[89] GitHub. *10 million repositories.* `https://github.com/blog/1724-10-million-repositories`, last accessed on 04/01/2021.

[90] GitHub. *About status checks.* `https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-status-checks`, last accessed on 04/01/2021.

[91] GitHub. *Code review.* `https://github.com/marketplace/category/code-review`, last accessed on 04/01/2021.

[92] GitHub. *Code scanning now available on GitHub Enterprise.* `https://resources.github.com/code-scanning/`, last accessed on 04/01/2021.

[93] GitHub. *GitHub API.* `https://developer.github.com/v3`, last accessed on 04/01/2021.

[94] GitHub. *GitHub Platform Roadmap.* `https://developer.github.com/early-access/platform-roadmap`, last accessed on 04/01/2021.

[95] GitHub. *Git's pack protocol.* `https://github.com/git/git/blob/master/Documentation/technical/pack-protocol.txt`, last accessed on 04/01/2021.

[96] Github. *GPG signature verification.* `https://github.com/blog/2144-gpg-signature-verification`, last accessed on 04/01/2021.

[97] GitHub. *Managing a branch protection rule.* `https://docs.github.com/en/github/administering-a-repository/managing-a-branch-protection-rule`, last accessed on 04/01/2021.

[98] GitHub. *Search more than 224M repositories.* `https://github.com/search`, last accessed on 04/01/2021.

[99] GitHub. *Statuses.* `https://docs.github.com/en/rest/reference/repos#statuses`, last accessed on 04/01/2021.

[100] GitHub. *The state of the Octoverse.* `https://octoverse.github.com`, last accessed on 04/01/2021.

[101] GitLab. *API Docs.* `https://docs.gitlab.com/ee/api/`, last accessed on 04/01/2021.

[102] GitLab. *Merge Request Approvals.* `https://docs.gitlab.com/ee/user/project/merge_requests/merge_request_approvals.html`, last accessed on 04/01/2021.

[103] Dan Goodin. *Kernel.org Linux repository rooted in hack attack.* `http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach`, last accessed on 04/01/2021.

[104] Dan Goodin. *Meet "Great Cannon", the man-in-the-middle weapon China used on GitHub.* `https://arstechnica.com/security/2015/04/meet-great-cannon-the-man-in-the-middle-weapon-china-used-on-github/`, last accessed on 04/01/2021.

[105] GreatFire. *China, GitHub and the man-in-the-middle.* `https://en.greatfire.org/blog/2013/jan/china-github-and-man-middle`, last accessed on 04/01/2021.

[106] Matthew Green. *Attack of the week: FREAK (for 'factoring the NSA for fun and profit').* `https://blog.cryptographyengineering.com/2015/03/03/attack-of-week-freak-or-factoring-nsa/`, last accessed on 04/01/2021.

[107] Hackread. *Proton malware.* `https://www.hackread.com/hackers-infect-mac-users-proton-malware-using-elmedia-player`, last accessed on 04/01/2021.

[108] Hanno Bock, Juraj Somorovsky, and Craig Young. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 817–849, 2018.

[109] Egor Homakov. *How I hacked GitHub again.* `http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html`, last accessed on 04/01/2021.

[110] Ahmed Ibrahim, Mohammad El-Ramly, and Amr Badr. Beware of the vulnerability! how vulnerable are github's most popular php applications? In *Proceedings of the 16th IEEE International Conference on Computer Systems and Applications (AICCSA)*, pages 1–7, 2019.

[111] Edward Iskra. *Critical Warning.* `https://bitcoingold.org/critical-warning-nov-26`, last accessed on 04/01/2021.

[112] isomorphic-git v0.65.0. `https://github.com/isomorphic-git/isomorphic-git/releases/tag/v0.65.0`, last accessed on 04/01/2021.

[113] Akshay Kalyan, Matthew Chiam, Jing Sun, and Sathiamoorthy Manoharan. A collaborative code review platform for github. In *Proceedings of the 21st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 191–196, 2016.

[114] Jungho Kang and Jong Hyuk Park. A secure-coding and vulnerability check system based on smart-fuzzing and exploit. *Neurocomputing*, 256:23–34, 2017.

[115] Kohsuke Kawaguchi. *Summary Report: Git Repository Disruption Incident of Nov 10th.* `https://www.jenkins.io/blog/2013/11/25/summary-report-git-repository-disruption-incident-of-nov-10th/`, last accessed on 04/01/2021.

[116] Keybase. *Introducing Keybase Chat.* `https://keybase.io/blog/keybase-chat`, last accessed on 04/01/2021.

[117] SS Kulkarni, Aastha Mittal, and Aniket Nayakawadi. Detecting phishing web pages. *International Journal of Computer Applications*, 118(16), 2015.

[118] le-git-imate. `https://le-git-imate.github.io/`, last accessed on 04/01/2021.

[119] Matthew M Lucas and Nikita Borisov. Flybynight: mitigating the privacy risks of social networking. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, pages 1–8, 2008.

[120] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An Analysis of China's "Great Cannon". In *Proceedings of the 5th USENIX Workshop on Free and Open Communications on the Internet (FOCI 15)*, 2015.

[121] Matt Cooper. *How We Use Git at Microsoft*. `https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/use-git-microsoft`, last accessed on 04/01/2021.

[122] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[123] Microsoft. *Open source projects and samples from Microsoft*. `https://github.com/microsoft`, last accessed on 04/01/2021.

[124] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall Joint Computer Conference*, pages 267–277, 1968.

[125] Michael Mimoso. *Hacker Puts Hosting Service Code Spaces Out of Business*. `https://threatpost.com/hacker-puts-hosting-service-code-spaces-out-of-business/106761/`, last accessed on 04/01/2021.

[126] Amber Mishra. *Secure Code Review – A Necessity*. `https://www.kratikal.com/blog/secure-code-review-a-necessity/`, last accessed on 04/01/2021.

[127] MITRE. *Secure Code Review*. `https://www.mitre.org/publications/systems-engineering-guide/enterprise-engineering/systems-engineering-for-mission-assurance/secure-code-review`, last accessed on 04/01/2021.

[128] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *Security Advisory*, 21:34–58, 2014.

[129] Matt Mullenweg. *Passwords Reset*. `https://wordpress.org/news/2011/06/passwords-reset`, last accessed on 04/01/2021.

[130] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[131] Ryan Naraine. *Adobe code signing infrastructure hacked by sophisticated threat actors.* `https://www.zdnet.com/article/adobe-code-signing-infrastructure-hacked-by-sophisticated-threat-actors/`, last accessed on 04/01/2021.

[132] Ryan Naraine. *Open-source ProFTPD hacked, backdoor planted in source code.* `http://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code`, last accessed on 04/01/2021.

[133] Sumaira Nazir, Nargis Fatima, and Suriayati Chuprat. Modern code review benefits-primary findings of a systematic literature review. In *Proceedings of the 3rd International Conference on Software Engineering and Information Management*, pages 210–215, 2020.

[134] Jakob Nielsen. Usability engineering at a discount. In *Proceedings of the 3rd International Conference on Human-computer Interaction on Designing and Using Human-computer Interfaces and Knowledge Based Systems (2nd ed.)*, pages 394–401, 1989.

[135] Nielsen.com. *GLOBAL TRENDS IN ONLINE SHOPPING - A NIELSEN REPORT.* `http://www.nielsen.com/us/en/insights/reports/2010/Global-Trends-in-Online-Shopping-Nielsen-Consumer-Report.html`.

[136] Pushkar Ogale, Michael Shin, and Sasanka Abeysinghe. Identifying security spots for data integrity. In *Proceedings of the 42nd IEEE Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 462–467, 2018.

[137] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Human-Computer Interaction*, 2(CSCW):135, 2018.

[138] Idrees Patel. *Janus Vulnerability.* `https://www.xda-developers.com/janus-vulnerability-android-apps`, last accessed on 04/01/2021.

[139] Darren Pauli. *It's 2017 and 200,000 services still have unpatched Heartbleeds.* `https://www.theregister.co.uk/2017/01/23/heartbleed_2017`, last accessed on 04/01/2021.

[140] Jeronimo Pellegrini. Secrecy in concurrent version control systems. In *Proceedings of the Brazilian Symposium on Information and Computer Security (SBSeg 2006)*, 2006.

[141] Nicolas Poggi, David Carrera, Ricard Gavalda, Eduard Ayguadé, and Jordi Torres. A methodology for the evaluation of high response time on e-commerce users and sales. *Information Systems Frontiers*, 16(5):867–885, 2014.

[142] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: an empirical investigation on code change reviewability. In *Proceedings of the 26th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 201–212. ACM, 2018.

[143] Thomas Reed. *Transmission hijacked again to spread malware*, 2016. `https://blog.malwarebytes.com/threat-analysis/2016/09/transmission-hijacked-again-to-spread-malware`, last accessed on 04/01/2021.

[144] Juliano Rizzo and Thai Duong. Practical padding oracle attacks. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, page 1–8, 2010.

[145] Juliano Rizzo and Thai Duong. The crime attack. In *Proceedings of the Ekoparty Security Conference*, 2012.

[146] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th ACM International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190. ACM, 2018.

[147] Reza Curtmola Sangat Vaidya, Santiago Torres-Arias and Justin Cappos. Commit signatures for centralized version control systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 359–373. Springer, 2019.

[148] Intel Security. *BERserk Vulnerability Part 1: RSA signature forgery attack due to incorrect parsing of ASN.1 encoded DigestInfo in PKCS#1 v1.5. Part2: Certificate Forgery in Mozilla NSS (Technical Report)*, 2014.

[149] P. J. Sevcik et al. Understanding how users view application performance. *Business Communications Review*, 32(7):8–9, 2002.

[150] Russell G Shirey, Kenneth M Hopkinson, Kyle E Stewart, Douglas D Hodson, and Brett J Borghetti. Analysis of implementations to secure git for use as an encrypted distributed version control system. In *Proceedings of the 48th IEEE Hawaii International Conference on System Sciences*, pages 5310–5319, 2015.

[151] SmartBear. *The 2019 State of Code Review: Trends and Insights into Collaborative Software Development.* `https://smartbear.com/resources/ebooks/the-state-of-code-review-2019/` , last accessed on 04/01/2021.

[152] Sonatype. *2020 State of the Software Supply Chain.* `https://www.sonatype.com/resources/white-paper-state-of-the-software-supply-chain-2020`, last accessed on 04/01/2021.

[153] Dane Stuckeyi. *Defending Infrastructure as Code in GitHub Enterprise.* `https://www.sans.org/reading-room/whitepapers/securecode/paper/39380`, last accessed on 04/01/2021.

[154] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[155] Symantec. *Internet Security Threat Report 2018.* `https://docs.broadcom.com/doc/istr-23-2018-en`, last accessed on 04/01/2021.

[156] Symantec. *Internet Security Threat Report 2019.* `https://docs.broadcom.com/doc/istr-24-2019-en`, last accessed on 04/01/2021.

[157] Synopsys. *Secure Code Review.* `https://www.synopsys.com/glossary/what-is-code-review.html`, last accessed on 04/01/2021.

[158] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150, 2015.

[159] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, 2019.

[160] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 379–395, 2016.

[161] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of curveswap: Measuring elliptic curve implementations in the wild. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 384–398, 2018.

[162] Steven J. Vaughan-Nichols. *Red Hat's Ceph and Inktank code repositories were cracked.* `http://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked`, last accessed on 04/01/2021.

[163] David A. Wheeler. *Software Configuration Management (SCM) Security.* `http://www.dwheeler.com/essays/scm-security.html`, last accessed on 04/01/2021.

[164] David A. Wheeler. *"The Apple goto fail vulnerability: lessons learned".* `https://dwheeler.com/essays/apple-goto-fail.html` , last accessed on 04/01/2021.

[165] Wikipedia. *SaaS*. `https://en.wikipedia.org/wiki/Software_as_a_service`, last accessed on 04/01/2021.

[166] Cheng Yang, Xun-hui Zhang, Ling-bin Zeng, Qiang Fan, Tao Wang, Yue Yu, Gang Yin, and Huai-min Wang. Revrec: A two-layer reviewer recommendation algorithm in pull-based development model. *Journal of Central South University*, 25(5):1129–1143, 2018.

[167] Joonseok Yang, Duksan Ryu, and Jongmoon Baik. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 115–122, 2016.

[168] Yue Yu, Huaimin Wang, Gang Yin, and Charles X Ling. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In *Proceedings of the 21st IEEE Asia-Pacific Software Engineering Conference*, volume 1, pages 335–342, 2014.

[169] Kim Zetter. *'Google' Hackers had ability to alter source code'*. `https://www.wired.com/2010/03/source-code-hacks`, last accessed on 04/01/2021.

[170] Yue Zhang, Jason I Hong, and Lorrie F Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th ACM International Conference on World Wide Web*, pages 639–648, 2007.