Journal of Computer Security 00 (20xx) 1–32
DOI 10.3233/JCS-210098
IOS Press

Towards verifiable web-based code review systems

Hammad Afzali [a], Santiago Torres-Arias [b], Reza Curtmola [a,*] and Justin Cappos [c]
[a] *Department of Computer Science, New Jersey Institute of Technology, NJ, USA*
E-mails: ha285@njit.edu, crix@njit.edu
[b] *Department of Electrical and Computer Engineering, Purdue University, IN, USA*
E-mail: santiagotorres@purdue.edu
[c] *Tandon School of Engineering, New York University, NY, USA*
E-mail: jcappos@nyu.edu

Abstract. Although code review is an essential step for ensuring the quality of software, it is surprising that current code review systems do not have mechanisms to protect the integrity of the code review process. We uncover multiple attacks against the code review infrastructure which are easy to execute, stealthy in nature, and can have a significant impact, such as allowing malicious or buggy code to be merged and propagated to future releases. To improve this status quo, in this work we lay the foundations for securing the code review process. Towards this end, we first identify 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 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. We implement SecureReview as a Chrome browser extension for GitHub and Gerrit. Our security analysis shows that SecureReview is effective in mitigating the aforementioned attacks. An experimental evaluation shows that the SecureReview implementation only adds a slight storage overhead (*i.e.*, less than 0.0006 of the repository size).

Keywords: Code review policy, verifiable code review process, review unit, browser extension, GitHub, Gerrit

1. Introduction

Code review is an integral part of the software development cycle [48,60]. It plays an essential role for software quality assurance by ensuring that new code changes are integrated into the codebase only after being vetted by a number of reviewers [16,52]. Modern code review systems such as Gerrit [19], Collaborator [11], Crucible [15], and ReviewBoard [54] 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 used by Google for code review in open-source projects like Go, Chromium, and Android [8]. Also, GitHub hosts over 240 million repositories and is used by Microsoft to host thousands of projects and perform many development tasks [7,12,43]. Our analysis of the top 50 most starred GitHub projects shows that 45 out of 50 projects are using code review features, demonstrating their value to software engineers.

Despite providing a rich set of features, modern code review systems do not incorporate safeguards against tampering with the code review process. Even if a project owner defines a flawless code review

*Corresponding author. E-mail: crix@njit.edu.

policy, there are no guarantees that the intended code review process was indeed followed. As a result, end users and any independent auditor are left no choice but to assume that the code review server faithfully followed the code review policy. The lack of a secure code review step is the cause behind a significant percentage of attacks [44]. A compromised code review system can cause great damage [39]. Moreover, several important threats are associated with the code review system [53,58]. In the current threat landscape, attacks against the software supply chain have become a common occurrence with devastating impacts [6,13,65]. As such, it is paramount to ensure that the various steps of the software development chain have adequate protection.

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 or limited access to the code review information in subsequent steps of the development chain (*e.g.*, build) 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 what occurred in the code review 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. However, 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 [33], BitBucket [3], GerritHub [24], Crucible [15], and Phabricator [50]). 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 quality 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. This basic guarantee must be provided if we are to hope the code review step is secure.

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 [14,59]. With our defense in place, the bar for attackers is significantly raised, making it difficult to execute attacks while remaining undetected. Specifically, we make the following contributions:

(1) Based on a thorough understanding of the code review process subtleties on different code review platforms, we perform a comprehensive analysis of the attack surface in relation to the components

of a code review system. As a result, we are the first to uncover a large attack surface by identifying attacks that tamper with the integrity of the code review process. These attacks are mostly related to specific details of the code review systems, easy to execute, stealthy in nature, and can have a significant impact. For instance, the attacks can lead to introducing a vulnerable or backdoored piece of code into the software product.

(2) This analysis enables us to identify a set of key design principles necessary to secure the code review process. We 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, and have released it as free and open-source software [56]. Our solution features several advantages that can facilitate its practical 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. SecureReview can sign and store code reviews in less than half a second and can merge changes between half a second (on Gerrit) and two seconds (on GitHub). Moreover, SecureReview adds at most 2 KB per Git commit which represents less than 0.0006 of the repository size even for small repositories.

2. Background

This section provides background on the code review process. After a general overview, we look at two popular code review systems, GitHub [27] and Gerrit [19].

2.1. The 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 [49]. 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 [52,55]. For instance, Microsoft, Google, Facebook, and VMware perform the review process using CodeFlow [9], Gerrit [19], Phabricator [50], and ReviewBoard [54], 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 [27], GitLab [33], Bitbucket [3]) and code review systems such as Gerrit [19] adhere to this model.

2.2. A code review workflow

As shown in Fig. 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).

We provide next an overview of the code review workflow on GitHub and Gerrit, pointing out which steps in these code review systems correspond to the four steps of the generic code review workflow presented in Section 2.2.

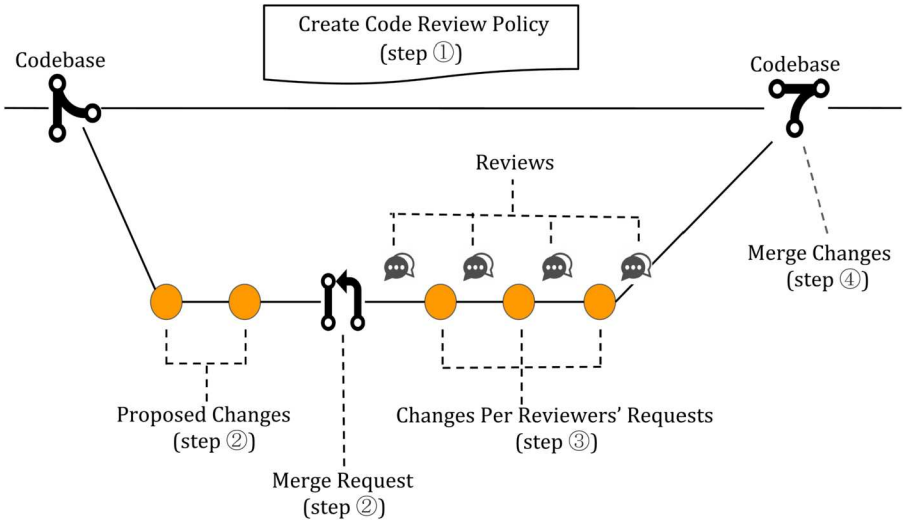

Fig. 1. A typical code review workflow.

2.3. The code review on 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 [28], which represents a growth of 28% compared to 2018. We provide next an overview of the GitHub's code review process.

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

2.3.2. GitHub code review policies

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 (Step ①). By default, code review is disabled, but can be enabled from a configuration option called "Branch Protection Rules" [29]. 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 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

[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.

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

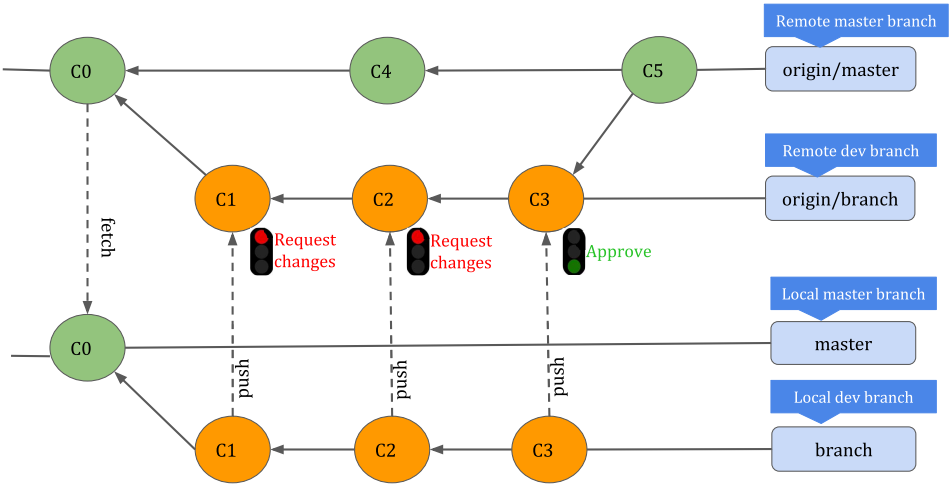

Fig. 2. The typical lifecycle of a pull request on GitHub.

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.

2.3.3. *Creating and merging 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 (Step ②). If reviewers request for changes or the developer herself decides to submit additional changes, she can update the pull request by submitting new commits (Step ③). Finally, when the pull request is approved (per the code review policy), it can be merged into the codebase (Step ④).

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 Fig. 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.

2.4. *The code review on Gerrit*

Gerrit, a popular code review system used in big open-source projects like Go, Chromium, and Android [8], is a highly configurable tool that was designed specifically to support the code review process. A merge request, referred to as a "change", allows code changes to be reviewed before being integrated into the codebase.

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 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, Gerrit users can perform reviews on a change. All review information is stored by Gerrit in a dedicated database, which is stored separately 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.

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.

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 *Maintainers* 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.

2.4.3. Gerrit code review policies

The owner of a Gerrit project can define a code review policy using different components (Step ①). The main component is the *submit rule*, 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 $+2$ [22]. 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).

2.4.4. Creating changes

In a typical code review workflow on Gerrit, developers create a change by uploading a new commit to the Gerrit server which is automatically sent to the `pending changes` location to be reviewed (Step ②). During the course of the review process, reviewers discuss the change and might ask for improvements (Step ③).

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.

2.4.5. Merging changes

When enough reviewers approve the change, the last patch set will be merged into the `authoritative` repository (Step ④). For example, consider the change shown in Fig. 3. 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.

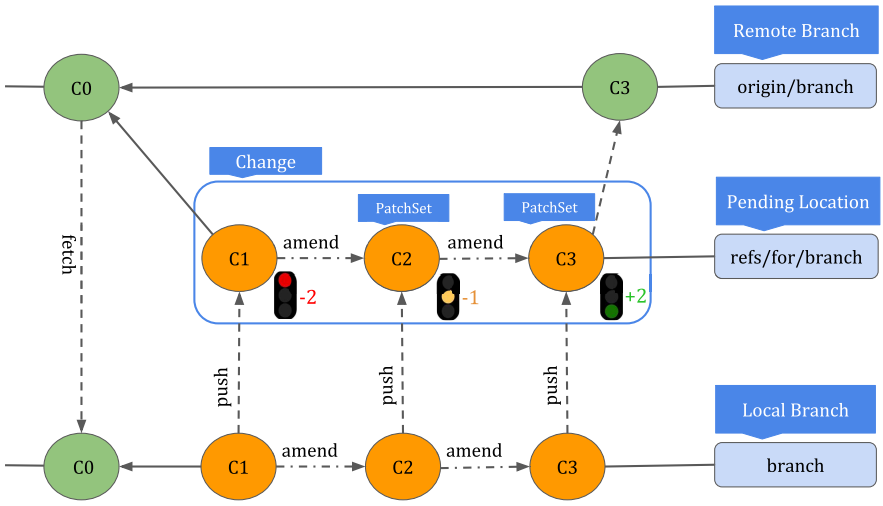

Fig. 3. The typical lifecycle of a code change in Gerrit under the default "merge if necessary" strategy.

Consider the repository shown in Fig. 3, 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.

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.

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 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 directly through a compromised or malicious code review server. Such attacks have been on the rise recently [17,40,47,51,53,57,64]. It can also happen indirectly through MITM attacks, such as government attacks against GitHub [36,42].

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 [1,2] 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 bypass the code review process and 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 while remaining undetected. As such, an unscrupulous server or a malicious reviewer cannot simply manipulate individual commits (*e.g.*, inject arbitrary new commits or modify existing commits) without self incriminating themselves. 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 [62].

We focus on attacks that tamper with the integrity of the code review process (specific attacks are described in Section 4). A limitation of our work is that attacks that tamper with code changes (*i.e.*, commits) performed by the user via the web UI are not addressed. For such attacks, we refer the reader to a comprehensive list of attacks and defenses [2].

We trust the users who are authorized to update the source code repository associated with the code review system. That includes users who are authorized to propose code changes (i.e., developers) or to merge code changes (*e.g.*, the project owner or a maintainer). We assume that 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 perform attacks that bypass the code review policies, as long as they do not self incriminate (*i.e.*, remain undetected). 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 paper:

- 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 that may have been introduced by malicious developers. Instead, we seek to protect the integrity of the code review process (*i.e.*, ensure that the prescribed review policy is being followed correctly).
- 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 dismissing stale approving reviews. We do not seek to detect such review policy flaws.

3.1. Security guarantees

To address this threat model, the goal of a secure code review system should be to enforce the following security guarantees:

- **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 independently verify the correctness of the code review process even after the code review phase.

4. Attacks

In this section, we identify new attacks against a typical code review workflow. Common to these attacks is the fact that an unscrupulous code review server can manipulate arbitrarily code review information. 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.

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,

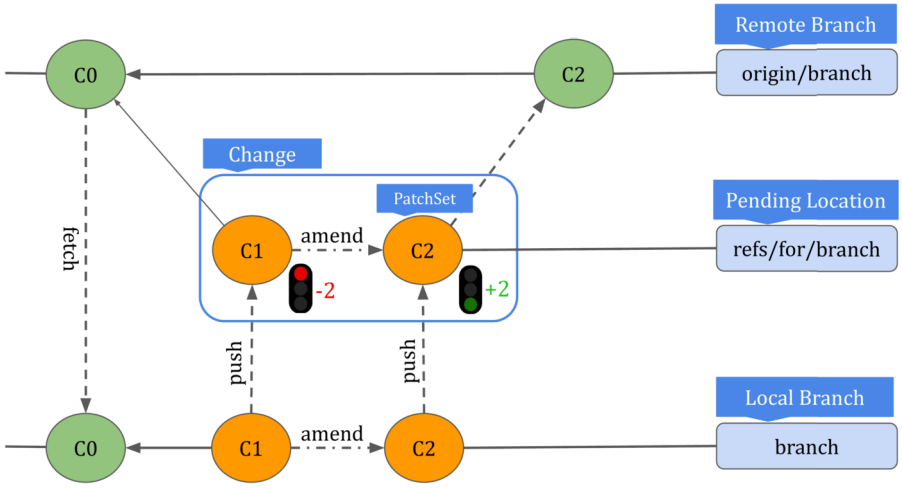

Fig. 4. Review manipulation attack.

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 manually.

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 Fig. 3 uses a review policy in which a change is mergeable if there is at least one review with the 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 Fig. 4 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.

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 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.

4.2.1. Bypassing 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 example, consider a GitHub code review policy that requires 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.

4.2.2. Counting 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 systems 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.

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.

4.2.3. Excluding 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 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.

4.2.4. Counting 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 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.

4.2.5. Merging 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.

4.2.6. Misusing 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.

4.2.7. Accepting 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.

4.2.8. Modifying project settings

A malicious server can modify different project settings to achieve AG1 and AG2. The server can simply disable the commit signing requirement. This would weaken significantly the project's security because then code could be arbitrarily manipulated in an undetectable fashion. 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 change maliciously. Moreover, an unscrupulous GitHub server may alter the `gitattributes` 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.

5. Solution: Securing the code review process with SecureReview

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.

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 or limited 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. In services like GitHub that allow managing the entire software lifecycle, only an authorized user can access the code reviews. For instance, consider a GitHub user who performs the deployment step in a project and wants to verify the code review process in that project. The user has to either blindly trust the GitHub server that the code review process followed the intended code review policy, or the user should obtain the authorization to check the code review information manually. In services like Gerrit that are dedicated to code review, the code reviews are separated from the source code (*i.e.*, stored in a different system) and are not accessible in other steps of the software development. Thus, users should go through different systems to access the code review information. For example, if a user who performs the build step wants to validate the integrity of the new code changes that are reviewed on the Gerrit server, they should leave the build system, obtain the authorization to access the Gerrit server, and manually go through the code review steps to ensure that the code changes were merged correctly.
- **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 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 principles 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 may be available either in partial form (*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: Reducing 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: Minimize 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.

5.2. *Strawman solutions*

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 we can address this issue with straightforward approaches.

Sign Commits. Signing code commits provides protection against tampering with the code but does not protect against manipulation of the code review process itself. As such, none of the attacks described in Section 4 can be detected.

Sign Code Review Policy. Having the project owner digitally sign the code review policy is a necessary step to ensure the integrity of the code review process. However, we argue that is it not sufficient. This approach can mitigate only a few attacks presented in Section 4 (*i.e.*, only attacks that incorporate changes from unauthorized users or modify the project settings). We note that 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.

5.3. *SecureReview design*

In a typical code review workflow (as described in Fig. 1 of Section 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 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) Creating and storing a signed code review policy, (2) Creating and storing signed reviews, and (3) Merging changes. We describe next these three components.

5.3.1. *Creating and storing a signed code review policy*

Popular code review systems allow users to define a code review policy, *e.g.*, GitHub's Branch Protection Rules [29], GitLab's Merge Request Approvals [34], or Gerrit's Submit Rules [22].

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` and `gitattributes` files 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 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.

5.3.2. *Creating and storing signed reviews*

SecureReview encapsulates each review in a *review unit* as shown in Fig. 5. 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 Fig. 6, 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

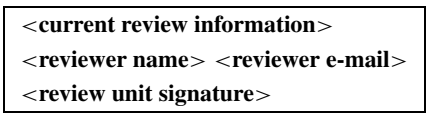

Fig. 5. The *review unit*'s format. Each review unit is computed by the reviewer who performed the review.

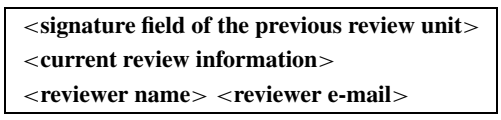

Fig. 6. The data over which the signature field in a review unit is computed. Note that the signature for the first review in a `Merge Request` omits the first field, as there is no previous review unit.

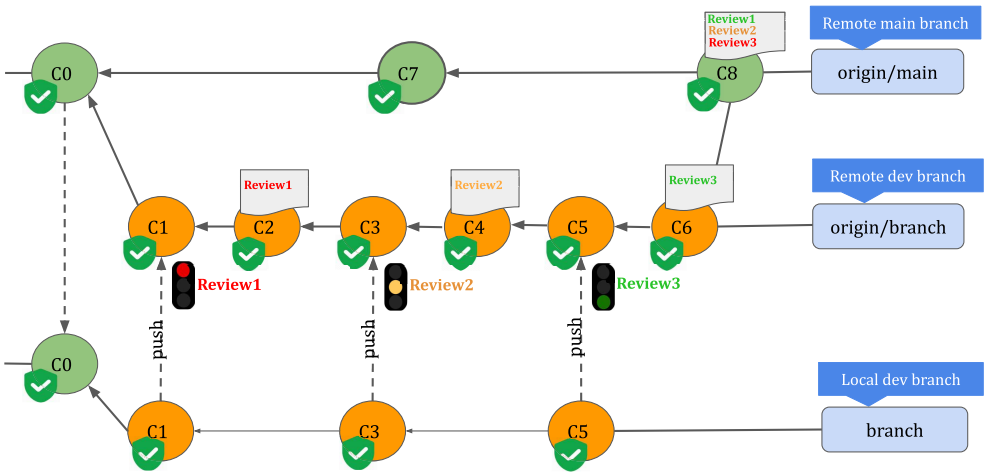

Fig. 7. Create signed code reviews.

that requires no (or minimal) server-side changes in current code review systems? To tackle this challenge, 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 [23,30,35]. 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.

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 object. This new commit is then pushed to the code review server (as described in Section 6). Thus, each code review results in a new signed commit object in the `Merge Request` as depicted in Fig. 7.

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

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 paper.

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 1–4), it checks if a direct push (*i.e.*, no reviews) is detected (lines 5–10). Then

[3] We note that at Google, each code change receives a peak of 12.5 reviews for changes of 1250 lines [55].

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

if the merger is authorized (lines 11–15), the most common code review policy rules are checked (lines 16–18) 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.

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.

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

```
1: Reviews ← getReviewUnits(Commits)
2: // Check if review units have valid signature and the chain of review units is valid.
3: if Validate_Review_Units(Reviews, ReviewPolicy) == false then
4:   return fail
5: ru ← length(Reviews)
6: if ru == 0 and FirstCommit(Commits) == false then
7:   // Check if the committer has the direct push access
8:   if DirectPush(Commits, ReviewPolicy) == false then
9:     // Unauthorized direct push is detected
10:    return fail
11: else
12:   // Check if the merger has the permission
13:   if AuthorizedMerger(Commits, ReviewPolicy) == false then
14:     // Unauthorized merge is detected
15:     return fail
16:   // Check the common rules are followed.
17:   if CommonRules(Reviews, ReviewPolicy) == false then
18:     return fail
19: return success
```

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, and have released it as free and open-source software [56]. 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 [41] and FlowCrypt [18]. The extension extracts information about the `Merge Request` from the web UI and from the code review server via REST APIs [23,30,35]. 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 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 (Minimum impact on the user's experience), SecureReview is implemented exclusively in JavaScript and does not require users to leave the browser. For this, we leverage prior work that fetches select information from the Git repository without retrieving the whole repository and without creating a working directory on the client side [2]. 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 commit, amend, merge, and push.

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.

In the following, we highlight SecureReview's main components when deployed on top of GitHub and Gerrit.

6.1. SecureReview for GitHub

In this section, we describe how SecureReview adds verifiability code review process on GitHub 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 (*i.e.*, `Branch Protection Rules`), the `CODEOWNERS` file (which contains a list of individuals that can provide approving reviews), and the `gitattributes` file (which can be used to exclude certain files from being displayed in the pull request's diff).

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) [31,32], 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 [30] 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 this procedure, we consider the example shown in Fig. 2. The reviewer makes three reviews before the `Pull Request` can be approved. With SecureReview in place, all review information is securely captured as shown in Fig. 7. After reviewing commit C1, the reviewer requests changes by making *Review1*. As a result, a new commit (C2) is created on top of C1. SecureReview helps the reviewer to embed the review in the GitHub repository as follows (1) form a *review unit* to store *Review1*; (2) retrieve the latest commit from the server to extract the commit hash and the "tree hash"; (3) compute a signed commit object (*i.e.*, C2) and embed *Review1* in its commit message; (4) push C2 to the server. In a similar manner, the reviewer uses SecureReview to embed *Review2* and *Review3* in the repository.

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 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.

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) [20,21]. 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 [23]. 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 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.

7. Security analysis

In this section, we first show how SecureReview achieves the security guarantees defined in Section 3.1. Then, we analyze SecureReview's ability to mitigate the attacks described in Section 4.

7.1. Achieving the security guarantees

SG1: Prevent unauthorized modification of stored code reviews and code review policy. SecureReview allows the project owner to protect the code review policy using a digital signature. Thus, any tampering with the review policy will be detected during the verification procedure. This includes, for example, tampering with the configuration files that are relevant for the code review policy, such as the list of users authorized to submit code changes. It also includes tampering with the policy rules, for example, disabling the commit signing requirement, or simply reducing the minimum number of approving reviews that is required. The review of a proposed change consists of a chain of signed review units. Each review unit includes a hash of the previous review unit and is protected by a digital signature computed by the respective reviewer. This means that any tampering with the sequence of review units or with individual review units will also be detected during the verification procedure. Thus, under the considered threat model, SecureReview achieves security guarantee SG1.

SG2: Ensure the integrity of the code review process. 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 computed by a developer who is authorized to perform reviews, (3) the code review policy has a valid signature computed by the project owner, and (4) the sequence of review units satisfies the intended code review policy, *i.e.*, the policy rules are satisfied. SecureReview was designed so that the actions of malicious reviewers do not affect the integrity of the reviews submitted by honest reviewers. This is because before being merged, the branch that contains the code changes encapsulates each individual review unit in a separate commit object at the repository level. As such, each individual reviewer is allowed to only handle her own reviews and

cannot manipulate the reviews from other reviewers. Even if two reviewers collude with each other, they are not able to tamper with reviews submitted by other honest reviewers. This is simply because in the review workflow imposed by SecureReview each review unit is digitally signed and stored in its own commit object at the repository level and, thus, it does not depend on the actions of other reviewers.

When dealing with a malicious server, these security checks allow SecureReview to detect if the server tampers with the reviews submitted for the code change that is about to be merged (*e.g.*, a server attempting to modify existing code reviews, add unauthorized code reviews, or remove reviews from the middle of the review chain). One strong attack is when the server, at some point before the code changes are merged, discards the last N review units. SecureReview cannot defend against such a powerful attack. However, there are two mitigating factors for this attack. First, the attack has a high probability of being detected by honest reviewers or developers who may notice that their reviews are missing. Second, even though SecureReview cannot handle such an attack, one could deploy a more specialized solution such as a reference state log [62], though at the cost of additional overhead. We conclude that, under the considered threat model, the code review process cannot be manipulated, and SecureReview achieves security guarantee SG2.

SG3: Ensure verifiability of the code review process. 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.

7.2. Mitigating the attacks

In this section, we analyze SecureReview's ability to mitigate the attacks introduced in Section 4.

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 5.3.4) fails in line 4.

Bypassing 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 10 or 18.

Counting 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 18.

Excluding 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). The verification procedure, however, checks that policy rule and will detect the server's misbehavior – the Validate_Reviews procedure fails in line 18.

Counting outdated reviews. The attacker may disable a policy that helps to catch bugs introduced by a patchset by counting outdated reviews 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 18.

Merging changes without addressing comments. A malicious server may prevent discovered code defects from being patched by ignoring an important code review policy (*i.e.*, a code change can not be merged unless all comments are addressed). With the sequence of signed review units, SecureReview can detect that the code change was merged without any approval from reviewers who made comments – the Validate_Reviews procedure fails in line 18.

Misusing 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 15.

Accepting 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 18.

Modifying project settings. A malicious server may modify different project settings to bypass the code review process. For instance, the server may manipulate user permissions, add unauthorized users, or disable the commit signing requirement. Any unauthorized changes in the code review rules or the project settings will be detected during the verification procedure when the code review policy signature is validated – the Validate_Branch procedure fails in line 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 [26]. To cover diverse repository configurations, we chose repositories of different history sizes, file counts, and file sizes, as shown in Table 1.

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 2.30 GHz), 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 in Section 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 three performance metrics: execution time, storage overhead, and verification time. We benchmarked both integration options of SecureReview (*i.e.*, "Full" and "Compact")

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

Table 1

Repositories chosen for the evaluation. We show the size of the working directory of the default branch, the file count, the average file size, and the total number of commits

Repository	Size (MB)	File count	File size (bytes)	History size (# of commits)
		GitHub		
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
		Gerrit		
bazlets	1	43	2,209	377
gitiles	6	185	4,818	933
gitblit	29	1,095	9,997	3,072
jgit	53	2,463	5,791	7,938
gerrit	234	4,717	7,288	44,944

Table 2

Execution time for storing signed reviews and merging changes (in seconds)

Repository	Sign and store	Merge changes
	GitHub	
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
	Gerrit	
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

for all these metrics except for execution time, where the runtime differences were negligible (and thus we only show one metric).

Execution time. Table 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.

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 [23,30] 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).

Storage overhead. Table 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. 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 58 bytes to the merge commit object (and a total of 232 bytes for four review units). For Full integration, the storage overhead for one

Table 3
Storage overhead per merge commit

Repository	Commit size (bytes)	C1	C4	F1	F4
		GitHub			
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%
		Gerrit			
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%

and four review units was 572 bytes and 2288 bytes, 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 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. Therefore, we argue that the storage overhead is negligible and does not affect the user experience. 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.

Verification time. The median execution time to verify a repository branch is 0.91 seconds. To measure the verification time, we first evaluated the top 100 most popular repositories that 1) used GitHub releases and 2) used merge commits in their development (e.g., instead of rebasing and squashing) and found that the average number of commits per release is 69. Then we ran the Validate_Branch procedure for the latest 69 commits of each repository's main branch.

9. Related work

SecureReview is, to the best of our knowledge, the first tool that ensures 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 [45,60]), 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 [4] 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 [5] 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 [46] 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. [37] 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. [38] 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 [10] 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 [66] 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 [63] 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 [25] provides a detailed description of creating and verifying Git signed commits.

Work by Torres-Arias et al. [62] covers some attack vectors against VCS where a malicious server tampers with Git metadata to trick users into performing unintended operations. The attacks are mitigated by maintaining a cryptographically-signed log of user actions. This solution may be used to protect against complex attacks that seek to tamper with the sequence of reviews (*e.g.*, removing reviews from the end of the review chain). However, such solutions only focus on the consistency of the VCS itself and target attacks that cause developers to have inconsistent views of the branches in a Git repository. In particular, they were not designed to account for the intricacies of the code review policies and cannot defend against the specific attacks identified in this work. Whereas they can complement the solutions proposed in our work, they can not fully ensure the integrity and consistency of the entire process by which code changes are integrated into the VCS.

Perhaps most closely related work to SecureReview is le-git-imate [2], 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.

Code review is one step of the software development cycle. Prior work seeks to secure the integrity of the entire software supply chain [61].

10. Conclusion

In this paper, 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. Although SecureReview lays the foundations for securing the code review process, this area of research looks ripe with open problems. We outline here several directions that could be explored by future work. First, some code review systems allow the server to automatically merge changes, which raises additional security concerns when validating such scenarios. Second, project owners may be reluctant to publicize code review information due to privacy concerns. A promising approach is to extend SecureReview to replace the review content with a hash of the review (salted to prevent dictionary attacks), and only reveal the content on demand. Finally, more work is needed to develop solutions that support dynamically changing review policies and verification of customizable rules.

Acknowledgments

This research was supported by the US NSF (National Science Foundation) under Grants No. CNS 1801430, DGE 1565478, and DGE 2043104.

References

[1] H. Afzali, S. Torres-Arias, R. Curtmola and J. Cappos, le-git-imate: Towards verifiable web-based Git repositories, in: *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS '18)*, 2018, pp. 469–482.

[2] H. Afzali, S. Torres-Arias, R. Curtmola and J. Cappos, Towards adding verifiability to web-based Git repositories, *Journal of Computer Security* **28**(4) (2020), 405–436. doi:10.3233/JCS-191371.

[3] Bitbucket, https://bitbucket.org.

[4] A. Bosu and J.C. Carver, Peer code review to prevent security vulnerabilities: An empirical evaluation, in: *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, IEEE, 2013, pp. 229–230. doi:10.1109/SERE-C.2013.22.

[5] A. Bosu, J.C. Carver, M. Hafiz, P. Hilley and D. 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*, 2014, pp. 257–268. doi:10.1145/2635868.2635880.

[6] A. Cherepanov, Analysis of TeleBots' cunning backdoor, https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor.

[7] C. 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/.

[8] Code Reviews at Google, https://www.michaelagreiler.com/code-reviews-at-google/.

[9] CodeFlow, https://codeflow.co.

[10] Codereviewhub, https://www.codereviewhub.com/.

[11] Collaborator, https://smartbear.com/product/collaborator/overview.

[12] M. Cooper, How we use Git at Microsoft, https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/use-git-microsoft.

[13] Crowdstrike, SUNSPOT: An implant in the build process, https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/.

[14] CrowdStrike, Securing the supply chain, https://www.crowdstrike.com/resources/wp-content/brochures/pr/CrowdStrike-Security-Supply-Chain.pdf.

[15] Crucible, https://www.atlassian.com/software/crucible.

[16] N. Fatima, S. Chuprat and S. Nazir, Challenges and benefits of modern code review-systematic literature review protocol, in: *2018 International Conference on Smart Computing and Electronic Enterprise (ICSCEE)*, IEEE, 2018, pp. 1–5.

[17] 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.

[18] FlowCrypt, https://flowcrypt.com.

[19] Gerrit, https://www.gerritcodereview.com.

[20] Gerrit, Labels, https://gerrit.opencord.org/Documentation/intro-project-owner.html#labels.

[21] Gerrit, Gerrit customized labels, https://gerrit.opencord.org/Documentation/config-labels.html#label_custom.

[22] Gerrit code review – Prolog submit rules, https://gerrit-review.googlesource.com/Documentation/prolog-cookbook.html.

[23] Gerrit code review – REST API, https://gerrit-review.googlesource.com/Documentation/rest-api.html.

[24] GerritHub, http://gerrithub.io.

[25] M. Gerwitz, A Git horror story: Repository integrity with signed commits, https://mikegerwitz.com/2012/05/a-git-horror-story-repository-integrity-with-signed-commits.

[26] Git repositories on Gerrit, https://gerrit.googlesource.com/.

[27] GitHub, https://github.com.

[28] GitHub, The state of the Octoverse, https://octoverse.github.com.

[29] GitHub, Managing a branch protection rule, https://docs.github.com/en/github/administering-a-repository/managing-a-branch-protection-rule.

[30] GitHub, GitHub API, https://developer.github.com/v3.

[31] GitHub, Statuses, https://docs.github.com/en/rest/reference/repos#statuses.

[32] GitHub, About status checks, https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-status-checks.

[33] GitLab, https://gitlab.com.

[34] GitLab, Merge request approvals, https://docs.gitlab.com/ee/user/project/merge_requests/merge_request_approvals.html.

[35] GitLab, API Docs, https://docs.gitlab.com/ee/api/.

[36] GreatFire, China, GitHub and the man-in-the-middle, https://en.greatfire.org/blog/2013/jan/china-github-and-man-middle.

[37] A. Ibrahim, M. El-Ramly and A. Badr, Beware of the vulnerability! How vulnerable are GitHub's most popular PHP applications? in: *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, IEEE, 2019, pp. 1–7.
[38] A. Kalyan, M. Chiam, J. Sun and S. Manoharan, A collaborative code review platform for Github, in: *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE, 2016, pp. 191–196. doi:10.1109/ICECCS.2016.032.
[39] K. 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/.
[40] S. Khandelwal, Github account of Gentoo Linux hacked, code replaced with malware, https://thehackernews.com/2018/06/gentoo-linux-github.html.
[41] Mailvelope, https://www.mailvelope.com/en.
[42] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert and V. Paxson, An analysis of China's "Great Cannon", in: *Fifth {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 15)*, 2015.
[43] Microsoft, Open source projects and samples from Microsoft, https://github.com/microsoft.
[44] A. Mishra, Secure code review – A necessity, https://www.kratikal.com/blog/secure-code-review-a-necessity/.
[45] MITRE, Secure code review, https://www.mitre.org/publications/systems-engineering-guide/enterprise-engineering/systems-engineering-for-mission-assurance/secure-code-review.
[46] P. Ogale, M. Shin and S. Abeysinghe, Identifying security spots for data integrity, in: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2, IEEE, 2018, pp. 462–467. doi:10.1109/COMPSAC.2018.10277.
[47] C. Osborne, Hijacked ASUS Live Update software installs backdoors on countless PCs worldwide, https://www.zdnet.com/article/supply-chain-attack-installs-backdoors-through-hijacked-asus-live-update-software/.
[48] OWASP Code Review Guide, https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf.
[49] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink and A. Bacchelli, Information needs in contemporary code review, in: *Proceedings of the ACM on Human-Computer Interaction 2 (CSCW)*, 2018, p. 135.
[50] Phabricator, https://www.phacility.com/phabricator.
[51] N. Popov, php.internls: Changes to Git commit workflow, https://news-web.php.net/php.internals/113838.
[52] A. Ram, A.A. Sawant, M. Castelluccio and A. Bacchelli, What makes a code change easier to review: An empirical investigation on code change reviewability, in: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2018, pp. 201–212. doi:10.1145/3236024.3236080.
[53] M. Reiter, Homebrew security incident disclosure, https://brew.sh/2021/04/21/security-incident-disclosure/.
[54] ReviewBoard, https://www.reviewboard.org.
[55] C. Sadowski, E. Söderberg, L. Church, M. Sipko and A. Bacchelli, Modern code review: A case study at Google, in: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ACM, 2018, pp. 181–190. doi:10.1145/3183519.3183525.
[56] SecureReview, https://github.com/thesecurereview/securereview.
[57] A. Sharma, Here's how a researcher broke into Microsoft VS Code's GitHub, https://www.bleepingcomputer.com/news/security/heres-how-a-researcher-broke-into-microsoft-vs-codes-github/.
[58] D. Stuckeyi, Defending infrastructure as code in GitHub enterprise, https://www.sans.org/reading-room/whitepapers/securecode/paper/39380.
[59] Symantec, Internet security threat report 2019, https://docs.broadcom.com/doc/istr-24-2019-en.
[60] Synopsys, Secure code review, https://www.synopsys.com/glossary/what-is-code-review.html.
[61] S. Torres-Arias, H. Afzali, T.K. Kuppusamy, R. Curtmola and J. Cappos, in-toto: Providing farm-to-table guarantees for bits and bytes, in: *28th USENIX Security Symposium (USENIX Security '19)*, 2019, pp. 1393–1410.
[62] S. Torres-Arias, A.K. Ammula, R. Curtmola and J. Cappos, On omitting commits and committing omissions: Preventing Git metadata tampering that (re)introduces software vulnerabilities, in: *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 379–395.
[63] D.A. Wheeler, Software configuration management (SCM) security, http://www.dwheeler.com/essays/scm-security.html.
[64] Z. Whittaker, Linux Mint website hacked to trick users into downloading version with backdoor, https://www.zdnet.com/article/linux-mint-website-hacked-malicious-backdoor-version/.
[65] WIRED, A mysterious hacker group is on a SUpply chain hijacking spree, https://www.wired.com/story/barium-supply-chain-hackers/.
[66] C. Yang, X.-H. Zhang, L.-B. Zeng, Q. Fan, T. Wang, Y. Yu, G. Yin and H.-M. Wang, RevRec: A two-layer reviewer recommendation algorithm in pull-based development model, *Journal of Central South University* **25**(5) (2018), 1129–1143. doi:10.1007/s11771-018-3812-x.