Unveiling Security Vulnerabilities in Git Large File Storage Protocol

Yuan Chen[†*], Qinying Wang[†*], Yong Yang[†], Yuanchao Chen[‡], Yuwei Li[‡], Shouling Ji[†(✉)]
[†]*Zhejiang University* [‡]*National University of Defense Technology,*
E-mails: {chenyuan, wangqinying, yangyong2022}@zju.edu.cn, {chenyuanchao, liyuwei}@nudt.edu.cn, sji@zju.edu.cn

***Abstract*—As an extension to the Git version control system that optimizes the handling of large files and binary content, Git Large File Storage (LFS) has been widely adopted by nearly all Git platforms. While Git LFS offers significant improvements in managing large files, it introduces new security implications that remain largely unexplored. This paper presents the first comprehensive security analysis of Git LFS, identifying 11 critical security properties that LFS servers must uphold. Building on our analysis of these property violations, we propose four new attack vectors: Private LFS File Leakage, LFS File Replacement, Quota-based Denial of Service (DoS), and Quota Escape. These attacks exploit weaknesses in practical LFS server implementations and can lead to serious consequences, including unauthorized access to sensitive files, malware injection, denial of service affecting all public repositories, and resource abuse. To evaluate the security of LFS implementations, we develop a semi-automated black-box testing tool and apply it to 14 major Git platforms. We uncover 36 previously unknown vulnerabilities and have responsibly disclosed them to the respective platform maintainers, receiving positive feedback and over $1800 in bug bounty rewards.**

1. Introduction

Git Large File Storage (LFS) [1] is an extension to the Git version control system specifically designed to optimize the management of large files and binary content, such as images, videos, and AI model files. Traditional Git workflows struggle with these large files primarily because Git stores complete file histories, and binary files cannot be efficiently diffed or compressed [2], [3]. Git LFS tackles this problem by replacing large files in the repository with lightweight pointer files and uploading actual files to the LFS server. This approach significantly reduces the repository size, improves performance, and ensures that large file content is only retrieved when necessary. Nowadays, Git LFS is widely adopted across most Git hosting platforms, from mainstream services like GitHub [4] to lightweight solutions like Gogs [5]. Even AI-focused platforms, such as Hugging Face [6], rely on Git LFS to store and distribute large datasets and model files efficiently.

However, the introduction of Git LFS expands the system's attack surface through the addition of the LFS server component. This new component necessitates robust access control mechanisms to prevent unauthorized file uploads and protect private user files from being leaked. For instance, CVE-2019-6786 [7] highlights a GitLab vulnerability that allowed attackers to steal private LFS files. More alarmingly, inadequately implemented LFS servers can be susceptible to file overwrite attacks, which compromise data integrity and could lead to severe supply-chain attacks. Such attacks enable indirect compromise of organizations through trusted third-party dependencies. This risk is particularly pronounced in the realm of AI, where large datasets and model files are commonly stored using LFS. Without proper permission checks and content verification, attackers can covertly embed backdoors into model files, facilitating a range of trojaning and poisoning attacks [8], [9], [10]. Additionally, weaknesses in quota mechanisms may allow adversaries to bypass storage restrictions or conduct denial-of-service (DoS) attacks by exhausting victims' storage quotas. This can severely disrupt their service usage. Moreover, the financial impact of quota bypasses could be substantial. For instance, an attacker uploading 10TB of data could potentially misappropriate storage worth $12,288 annually on GitHub or $115,302 on TencentCloud [11] without being charged.

To the best of our knowledge, no comprehensive security analysis of Git LFS has been conducted, with prior work limited to documented CVEs targeting LFS client [12], [13], [14] and GitLab server [7], [15], [16], [17] vulnerabilities, discovered through ad-hoc efforts. The broader security implications of LFS implementations across different Git providers remain largely unexplored and face three key challenges:

Challenge 1: Protocol Complexity. In contrast to traditional Git's simple client-server architecture, Git LFS introduces additional layers of complexity by incorporating multiple components, including the client, Git SSH server, LFS server, and storage server. This complexity hampers comprehensive security assessments by expanding the attack surface and increasing the difficulty of analyzing vulnerabilities across the system.

Challenge 2: Compositional Intricacy. The interplay between Git LFS and auxiliary features, such as repository archiving and forking, introduces subtle and often overlooked security risks. For instance, when a repository is archived (marked as read-only), LFS may still allow new uploads without updating the quota usage, potentially enabling a quota escape attack. These nuanced interactions highlight the need for rigorous investigation into possible

*The authors contributed equally to this work.
Shouling Ji is the corresponding author.

exploitation vectors within this complex system.

Challenge 3: Infrastructure Heterogeneity. The varied implementation approaches, including cloud storage integration, verification API call enforcements, and quota enforcement, significantly impede systematic vulnerability detection across different platforms.

To address Challenge 1, we adopt a feature-based, property-driven approach to systematically formalize Git LFS security analysis. We thoroughly examine the protocol's interactions and complexities across diverse infrastructures. Through this comprehensive analysis, we distill 11 critical security properties that LFS servers must uphold, with special attention to functional compositions and identifying new attack surfaces, thereby resolving Challenge 2. We then propose four new attack vectors that may arise if these security properties are violated:

- **Private LFS File Leakage**: This attack allows an adversary to download sensitive files from private repositories, compromising user confidentiality.
- **LFS File Replacement**: Without proper content verification, an attacker can replace uploaded files and, even more severe, cross-user files, thereby bypassing malware scans and executing supply-chain attacks, which poses a particular risk in AI model sharing contexts.
- **Quota-based DoS Attack**: This attack targets public repositories by consuming their LFS quota, rendering the service unusable to legitimate users.
- **Quota Escape**: This attack permits an attacker to upload and download large amounts of data without incurring associated costs, leading to severe resource abuse.

In addition to identifying these attacks in the wild, we develop a semi-automated tool that employs a black-box testing approach to detect vulnerabilities in LFS server implementations. In response to Challenge 3, we design our tool with a modular architecture that adapts to the distinct LFS features of each platform, ensuring thorough security assessments across diverse environments. Utilizing our tool, we systematically evaluate the security of 14 major Git platforms and have uncovered 36 previously unknown vulnerabilities. The evaluation results highlight the effectiveness of the proposed security properties and underscore the critical need for robust access control, file integrity verification, and systematic quota management. All vulnerabilities were responsibly disclosed to the respective platform maintainers, who have acknowledged the issues and provided positive feedback. In recognition of our efforts, we also received bug bounty rewards totaling over $1800.

The key contributions are summarized as follows.

- We conduct the first comprehensive security analysis of Git LFS servers, establishing 11 essential security properties that platforms should adhere to. Based on these security properties, we propose four novel attacks, including LFS File Leakage, LFS File Replacement, Quota-based DoS Attack, and Quota Escape.
- We develop a semi-automated testing framework to facilitate vulnerability detection in Git LFS, designed to be modular and extensible. We release it to facilitate future research at https://github.com/NESA-Lab/LFSonar.
- Through systematic analysis, we have identified 36 previously unknown vulnerabilities—including private file leakages, malware scanning bypasses, and various quota escape weaknesses—across the 14 Git platforms. Our findings have been validated through responsible disclosure to the affected platforms.

2. Background

2.1. LFS Protocol

This section provides an overview of the LFS protocol interactions. When a user has installed Git LFS extension by `git lfs install`, and marked a file as tracked by LFS using `git lfs track filename`, the LFS client will be triggered during `git push` and `git pull/git clone` operations. The client interacts with the LFS server to upload or download the relevant large files. As illustrated in Figure 1, we outline the steps involved in uploading an LFS object according to the LFS protocol.

Step 1: Authentication via SSH to Obtain an Authorization Token. In a typical LFS setup, LFS clients get an authentication token by connecting to the SSH server and executing *git-lfs-authenticate*. This command allows the server to generate a token, enabling secure HTTP-based interactions for LFS operations. Example:

```
$ ssh git@github.com git-lfs-authenticate
↪  user/repo.git upload
{ "href":
↪  "https://github.com/user/repo.git/info/lfs",
  "header": {
    "Authorization": "Bearer <token>"
  },
  "expires_in": 3600}
```

Alternatively, if the SSH server lacks LFS support or the Git platform only uses the HTTP protocol, users can authenticate by providing their username and password (or access token) via Basic Authentication. In this case, the `Authorization` header used subsequently will follow the format: `Basic base64encode(username:password)`.

Step 2: Batch API Interaction for Getting Actions. With the authorization token in hand, the client proceeds to interact with the LFS server's Batch API. The API allows clients to request actions for multiple files in a single HTTP request, optimizing network efficiency and reducing latency. For file uploads, the client sends a POST request to the Batch API endpoint with a JSON payload listing the objects (`oid` and `size`) to be uploaded:

```
POST /user/repo.git/info/lfs/objects/batch
Authorization: Bearer <token>
Content-Type: application/vnd.git-lfs+json
Accept: application/vnd.git-lfs+json

{ "operation": "upload",
  "objects": [{
      "oid": "<object ID (file SHA-256)>",
      "size": <file size in bytes>
    },...] }
```

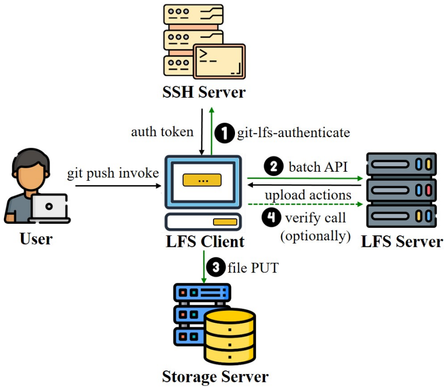


Figure 1: LFS protocol interactions for uploading a file.

The LFS server processes the request and returns action details for each object, typically including an `upload` action that specifies the upload URL along with the required headers for the LFS client. In certain LFS server implementations, a `verify` action is also provided, which the LFS client should call upon completion of the file upload. If the object already exists on the repository, no actions are returned. For download requests, the `actions` field in the response provides the download link and necessary headers, assuming the file exists and passes permission checks.

```
{ "transfer": "basic",
  "objects": [{
    "oid": "<object ID>",
    "size": <file size>,
    "actions": {
      "upload": {
        "href": "https://<upload url>",
        "header": {
          "Authorization": "<upload-token>",
          ... <more headers>
     }},"verify": {
        "href": "https://<verify api url>",
        "header": {
          "Authorization": "<verify-token>"
}}}}, ... <more objects>]}
```

For example, GitHub LFS Batch API returns an AWS S3 [18] URL as the upload `href`. Besides, the response also includes headers like `Authorization` (AWS4-HMAC-SHA256 Signature), `x-amz-content-sha256` and `x-amz-date`. These headers ensure that AWS S3 verifies the uploaded content against the requested SHA-256 hash. Additionally, GitHub returns the verify `href` in the format `https://lfs.github.com/<user>/<repo>/objects/<sha256>/verify`, with an `Authorization` header in the format `RemoteAuth gitauth-v1-xxx`, a custom authentication mechanism by GitHub.

Step 3: Uploading the Actual File Content via PUT. Utilizing the information (`href` and `headers`) from the Batch API response, the client uploads the file content to the specified storage server. The client performs an HTTP PUT request to the upload URL:

```
PUT https://<upload url>
Authorization: <upload-token>
Content-Length: <file size>
Content-Type: application/octet-stream
... <more headers>

<file content>
```

Upon a successful upload, the storage server responds with an HTTP 200 OK status, indicating that the file has been received and stored.

Step 4: Verification via the Verify API Call (Optional). To ensure the integrity and completeness of the uploaded files, the LFS protocol includes an optional verification step. Our empirical study shows that this step is commonly implemented by Git platforms using cloud-based storage to notify the LFS server once the file has been uploaded to the cloud. If the `verify` action is present in the Batch API response, the client will send a POST request to the verify API (`href` value in the `verify` action) after uploading the file, which contains the `oid` and `size`. The LFS server is responsible for confirming the existence and integrity of the uploaded file and will return an HTTP 200 OK status upon successful verification. If the verification fails like size mismatch, the server will return an error response, prompting the client to retry the upload.

LFS Pointer File Format. After successfully uploading the LFS object to the LFS server, the LFS client generates a pointer file containing metadata about the actual file. This pointer file is committed to the Git tree and subsequently pushed to the Git server. The pointer file is in the following plain text format:

```
version https://git-lfs.github.com/spec/v1
oid sha256:<SHA-256 hash of the actual file>
size <file size in bytes>
```

2.2. Git Platforms

Based on the functionality and open-source availability, Git platforms can be categorized into four types: Git-centric Platforms, Self-hosted Git Providers, Cloud Services, and AI Platforms.

- **Git-centric Platforms** (e.g., GitHub) focus exclusively on providing Git services.
- **Self-hosted Git Providers** (e.g., GitLab) offer open-source solutions that organizations can deploy on their own infrastructure.
- **Cloud Services** (e.g., Aliyun Codeup) integrate Git functionality into broader DevOps platforms tailored for enterprise needs.
- **AI Platforms** (e.g., Hugging Face) leverage Git LFS for storing and sharing AI models and datasets.

All Git platforms support hosting LFS files in Git repositories, though features and quotas vary[1]. Git-centric plat-

[1] For instance, GitHub provides 1GB free storage with a 50GB upgrade option for $5 monthly [19], while TencentCloud Coding charges $563 annually for an additional 50GB LFS space along with extra computational resources [11].

forms, for instance, often emphasize open source collaboration and therefore are more likely to support direct LFS file downloads from web pages. In contrast, Cloud services such as Aliyun Codeup do not support hosting public repositories. Similarly, AI platforms often lack support for the usage queries, deploy keys, and fork functionality.

This paper focuses on features pertaining to LFS capabilities, permission checking, file integrity, and quota accounting. Below, we provide a brief overview of these relevant features.

- **Public Repository**: allows users to designate a repository as public, thereby granting read-only permission to anonymous users.
- **Fork Public Repository**: enables users to fork repositories with shared Git histories, often used in open-source collaboration.
- **LFS Usage Query**: allows monitoring of current LFS storage usage.
- **LFS File Management**: enables viewing and deletion of LFS objects to manage storage effectively, as LFS protocol does not natively support file deletion [20].
- **LFS SSH Support**: allows users to use LFS with an SSH remote URL seamlessly, avoiding the need to input an account password for LFS authentication.
- **Deploy Key**: adds an SSH key for read-only repository access, commonly used in CI/CD scenarios.
- **Archive Repository**: enables users to mark a repository as read-only, thereby preventing further modifications.
- **Cloud Storage Direct Upload**: utilizes cloud services as the storage server, returning a pre-signed upload link in the Batch API call (Step 2 of the LFS protocol).
- **Predictable LFS Endpoint**: allows clients to infer the upload/download URL that would be returned by the Batch API. While this does not inherently constitute a security vulnerability, it may enable extra attack vectors.

2.3. Motivating Example

CVE-2019-6786 [7] is a vulnerability with a CVSS score of 6.5 affecting GitLab Community and Enterprise Editions prior to versions 11.5.8, 11.6.6, and 11.7.1. Due to incorrect access control, it allows unauthorized users to access a Git LFS object's content if they know the file's oid (SHA-256 hash) and size. According to the issue page [21], an attacker can exploit this vulnerability by uploading an empty file without a Content-Type header in Step 3 of the LFS protocol, bypassing the need for the actual file content. Since the affected version of GitLab relies on the `Content-Type` header set to `application/octet-stream` to identify client LFS API requests, the attacker is able to bypass the LFS handling mechanism, including permission checks and file hash validation. Consequently, the request triggers the internal API endpoint `PUT Projects::LfsStorageController #upload_finalize`, which verifies the existence of the LFS object in global shared storage based on its oid and size, and then creates a database record linking the LFS object to the attacker's repository, ultimately granting the attacker access to the sensitive file content.

This vulnerability exposes how seemingly minor implementation flaws can lead to significant security breaches. The intricate interactions among various components (client, Git SSH server, LFS server, and storage server) further complicate authentication and access control. Moreover, given the widespread adoption of Git platforms with diverse configurations, including cloud infrastructure integration, it is highly probable that similar vulnerabilities exist in other implementations, and that other components, such as quota management, may harbor distinct types of bugs. This motivates us to propose a rigorous security analysis and bug detection framework tailored to LFS implementations, aimed at uncovering latent flaws that could compromise access control mechanisms or other critical components.

3. LFS Security Analysis

In this paper, we focus on issues related to LFS servers in their implementation of the Git LFS protocol, while vulnerabilities in LFS clients or low-level binary issues (such as buffer overflows) are beyond the scope of our analysis. In this section, we present a systematic analysis of the LFS protocol structured around three fundamental dimensions of file storage security: confidentiality, integrity, and availability. These dimensions guide the identification of three critical aspects of the LFS protocol: access control (Section 3.1), file integrity (Section 3.2), and quota management (Section 3.3), each of which directly contributes to achieving these security goals. Access control ensures that only authorized users can store and retrieve files, thereby preserving confidentiality. File integrity guarantees that files stored on the LFS server correspond to their respective Git commits, ensuring data integrity. Quota management enforces limitations on storage usage, protecting server availability by preventing potential abuse.

To rigorously evaluate these aspects, we formulate a series of targeted questions based on widely accepted security principles and informed by vulnerabilities observed in similar protocols. We then conduct a thorough analysis of these questions, systematically examining each one to identify potential security issues and vulnerabilities. Based on this analysis, we formalize our findings into clearly defined **Security Properties** (SPs), to encapsulate key security requirements for a robust LFS server implementation.

3.1. Access Control

Who should be able to read a private LFS object? In a public repository, it is evident that all users can access LFS objects within that repository. However, in a private repository, access control must be strictly enforced, treating all files as confidential. Therefore, only the owner and collaborators with granted permissions to access the Git repository should be able to read the private LFS objects.

Users of the current repository should not be able to access LFS objects that are stored in other repositories. Even if the same object is already on the storage server, the upload process (Step 3 of the protocol) must still enforce strict file content verification[2]. As demonstrated by CVE-2019-6786, bypassing the upload step or using weak verification logic can allow attackers to exfiltrate sensitive LFS files by uploading empty content, leading to breaches of file confidentiality and user privacy.

In light of the above discussion, we propose the following two SPs:

SP1: Only users with read access to the Git repository should be authorized to download LFS objects that have been uploaded to the same repository.
SP2: File content must be verified during uploads before any LFS object is made accessible for reading, regardless of whether an identical LFS object already exists in other repositories.

Who can upload an LFS object to the repository? Uploading an LFS object can be considered as a write operation to the corresponding Git repository. If another user, whether logged in as a non-collaborator or as an anonymous user, is able to upload a new LFS file, the uploaded LFS object would consume the victim's LFS quota, leading to abuse and potential denial-of-service (DoS) attacks. Therefore, only authorized users with write access to the repository, authenticated with appropriate credentials such as the owner's SSH key or Basic Authentication (username/password), should be permitted to upload LFS objects. Furthermore, if coupled with file integrity violations, such unauthorized uploads could result in the overwriting of existing files, exacerbating the security risks and potentially leading to a supply-chain attack.

In addition to users, deploy keys may also have access to the repository. It is crucial to ensure that a deploy key with only read access cannot perform uploads. If this restriction is violated and the deploy key is compromised (e.g., through a third-party CI/CD platform breach), an attacker could exploit the unintended write permissions, escalating the threat from mere code leakage to a supply chain attack. This risk is particularly severe when users of the vulnerable Git platform rely on the assumed read-only nature of the deploy key.

Based on the preceding analysis, we formulate the following two SPs:

SP3: Only users with write access to the Git repository should be authorized to upload LFS objects.
SP4: A deploy key with only read access should not be able to upload LFS objects.

3.2. File Integrity

Can the file contents be maliciously tampered with? The SHA-256 hash serves as an immutable identifier shared between the Git repository and the LFS server, so it's required that the LFS server maintain the corresponding stored file unchanged to preserve data integrity. If an attacker can upload or overwrite an existing LFS file with altered content, this would trigger an error during subsequent download attempts by the Git LFS client. As the client checks whether the content of the downloaded object matches its corresponding object identifier (the file's SHA-256 hash), any mismatch would result in an exception, effectively blocking the execution of commands such as `git clone` or `git pull`, leading to a DoS attack.

Furthermore, victim users may unknowingly download the tampered file through the web interface, which often lacks content verification mechanisms, creating a significant risk of supply-chain attacks as replaced files with stealthy injected backdoors would not trigger any alert. Such a threat is particularly concerning in AI scenarios, where model files are commonly stored using LFS and are frequently downloaded via the web interface or API rather than through Git commands. If overwrite attacks are possible due to insufficient permission checks and content verification, attackers could stealthily inject backdoors into model files and datasets, posing severe security risks such as trojaning and poisoning attacks [8], [9], [10]. Notably, we have confirmed that the latest version (v0.26.2) of the Huggingface Hub Python SDK [22] does not verify the SHA-256 hash of downloaded LFS files, exacerbating this threat.

As demonstrated by the prior discussion, we propose the following SP:

SP5: Uploaded LFS objects must be verified to ensure their content matches the claimed SHA-256 hash, to prevent the upload of tampered files.

3.3. Quota Management

When should quota usage be increased? Quota usage should remain unchanged in the case of a download request, as this action is merely a read operation. However, if the LFS server is improperly implemented, it might allow the downloading an LFS object that exists globally but is unknown to the current public repository, which would also violate SP1. Under such circumstances, the server might automatically generate records that link the object to the repository, thereby consuming the repository's quota. This flawed design, where download requests can increase quota usage, enables potential DoS attacks against all public repositories, threatening platform-wide availability.

For upload requests, ideally, the usage should be updated immediately upon successful completion of the file upload. Nevertheless, implementing such a mechanism can be challenging, particularly for Git platforms employing object storage services (e.g., AWS S3) as the storage server (as shown in Figure 2). In these cases, the uploaded file is directly transferred to the storage server without notifying the LFS server to update the quota usage. To address this challenge, the LFS protocol includes an optional verify API

[2] To optimize performance, the file upload process may be safely skipped if and only if: (a) the LFS object exists in at least one public repository, or (b) the current user has read access to at least one private repository that contains the object.

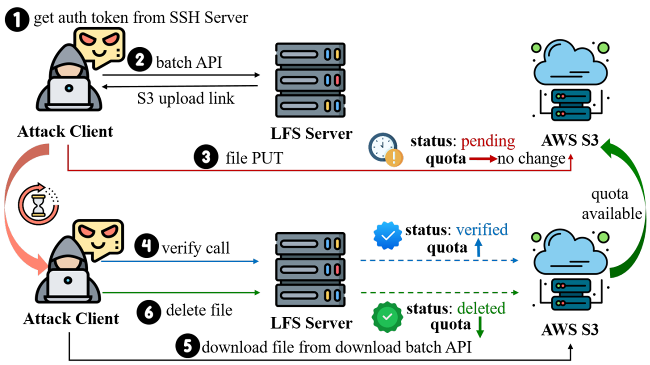


Figure 2: Delay attack against quota mechanism.

call (Step 4), requiring the client to send a POST request to the LFS server after completing the PUT request. If the LFS server relies on this verification step to increase the quota usage, but still permits the downloading of objects that bypassed this verification step, abuse becomes possible. An adversary could easily modify the LFS client to omit the verification step, thus bypassing the quota limitation and uploading an unlimited number of files. Such a flaw could result in substantial quota abuse, thereby affecting both the stability and reliability of the platform.

Upon further reflection, even if LFS objects are restricted from being downloaded without invoking the verify API, the risk of abuse persists if uploaded objects are stored without an expiration mechanism and the verify step can be delayed. This security gap stems from the exclusion of pending objects from quota calculations. As illustrated in Figure 2, an attacker can first upload a large number of files without invoking the verify API. Once the attacker needs to download the files after an extended period, they can finalize the upload by invoking the verify API (Step 4). This step remains valid because pending files are not subject to garbage collection and will temporarily increase quota usage. The attacker can then obtain a download URL via the Batch API (Step 5). After downloading, by deleting files (Step 6) to reclaim quota and repeating the upload process (Steps 1–3), the attacker can effectively bypass quota limits. While this requires some effort, continuous deletion and re-uploading allow the attacker to make additional files downloadable. It is important to note that the "pending" state is only conceptual since the LFS server is unaware of the files until the verify API is called. This vulnerability is particularly relevant in cold data backup scenarios, where large volumes of data are stored long-term with infrequent access.

Given the discussion above, we outline the following three SPs:

SP6: Download requests must not increase quota usage.
SP7: Files excluded from quota management should not be downloadable.
SP8: Pending objects should be freed regularly.

How much should quota usage be increased? Quota management should accurately track the size of the uploaded file, rather than relying on the size parameter provided by a malicious client. Similar to cloud storage services (e.g., Dropbox), any manipulation of the file size parameter can pose a significant risk to the provider, as it can enable abuse attacks. Therefore, in addition to verifying the file content against its claimed SHA-256 (SP5), the actual size of the uploaded file must also be verified against the provided size parameter at all stages.

We summarize the above discussion into the following SP:

SP9: Quota usage should increase by the exact size of the uploaded file, and requests with incorrect size parameters must be rejected.

To whom should quota usage be attributed? While it may seem logical to attribute quota usage to the repository where a file is uploaded, this becomes complex with forked repositories. GitHub's policy, as documented in [19], attributes both bandwidth and storage usage from fork repositories to the root repository of the network. This design choice leaves public repositories vulnerable to both resource abuse and DoS attacks through their forks. We are currently discussing this issue with GitHub to address the underlying design flaw.

The above discussion leads to the following SP:

SP10: Quota usage of uploaded files should be attributed to the current repository.

When should quota usage be decreased? There are two primary methods to delete LFS objects: managing them via the web interface or deleting the entire repository. However, decreasing quota does not necessarily imply that the file is no longer accessible, leaving the potential for abuse. For instance, many Git platforms do not delete LFS objects when a repository is removed, likely to provide an option for recovery within a specific timeframe. If these files remain accessible, such as being downloadable from a newly recreated repository with the same name, it could lead to a quota limitation bypass, as these files would no longer count toward the user's quota but would still be available for download.

The preceding analysis results in the following SP:

SP11: Quota usage should only be decreased after LFS objects are properly deleted.

To summarize this section, we present a systematic security analysis of the LFS protocol across confidentiality, integrity, and availability dimensions, examined through access control, file integrity, and quota management. The analysis yields 11 Security Properties: access control properties (SP1-4) governing read and write permissions, file integrity property (SP5) mandating content verification, and quota management properties (SP6-11) preventing resource abuse. These properties collectively establish essential security requirements for robust LFS server implementations.

4. Threat Model and New Attacks

In this section, we outline our threat model and introduce four novel attacks targeting Git LFS servers. These attack vectors exploit various vulnerabilities in the LFS mechanism, which have not been addressed in prior research.

4.1. Adversarial Capabilities

We assume the adversary has the following capabilities:

- Create accounts and repositories on the Git platform.
- Upload files to his own repositories.
- Read and fork public repositories, but cannot access private repositories belonging to the targeted entity.
- Possess knowledge of the file metadata (SHA-256 hash and size) of the target sensitive files.

The first three capabilities reflect the standard privileges available to any user of a Git platform that supports public registration. The fourth capability, while more advanced, is still plausible in several real-world scenarios. For instance, a user may inadvertently leak a downloaded archive of a Git repository that contains LFS object pointers, which include the SHA-256 hash and file size of the files. Another realistic scenario involves a company deleting a public repository, possibly for compliance reasons. In such cases, users who previously cloned the repository might attempt to exploit LFS vulnerabilities to retrieve the LFS objects from the deleted repository, which should now be treated as private. Additionally, adversaries might infer or obtain file metadata through man-in-the-middle attacks or other indirect channels, such as leaked build artifacts, logs, or backup files, all of which could expose file hashes or sizes.

4.2. New Attacks

Based on these assumptions and the preceding discussion on the security properties that LFS servers must maintain, we identify four new attack vectors.

4.2.1. Private LFS File Leakage. In this attack, the adversary exploits access control weaknesses or employs other attack techniques to gain unauthorized access to LFS objects stored in private repositories. This breach violates SP1 and SP2, undermining both user privacy and data confidentiality. Further, we consider files from deleted public repositories as private and therefore potential targets for this attack.

4.2.2. LFS File Replacement. This attack exploits the LFS server's failure to verify file contents against their claimed SHA-256 hashes. As a result of violating SP5, the adversary can replace legitimate LFS objects with malicious or altered content. While the LFS client would detect this discrepancy during a git clone or git pull operation, causing the process to fail (DoS) and impacting availability, the manipulated files could still be downloaded through non-Git interfaces. If these interfaces do not issue appropriate warnings, the attack poses a significant supply-chain risk and compromises data integrity. The violations of SP3 and SP4, allowing attackers to upload files to repositories owned by other users, may further escalate damage.

Additionally, this file replacement attack can be used to bypass virus scanning. For example, some Git providers, such as Hugging Face [23], integrate with virus scan engines like ClamAV. An attacker could initially upload a benign file to deceive the virus scanner into marking the file as safe, then subsequently overwrite it with malicious content. For this specific attack, we assume the attacker can upload LFS objects to the target repository, either by exploiting an access control vulnerability, or by being granted a collaborator role.

4.2.3. Quota-Based Denial of Service Attack. In this attack, the adversary depletes the allocated storage quota of a public repository by uploading large files or excessive LFS objects. Once the quota is exhausted, the LFS server will deny service to legitimate users of the repository, directly impacting its availability. This attack becomes more severe if the platform lacks support for LFS object management and deletion, leaving the repository owner unaware of the attack. As a result, the owner may be forced to either purchase additional storage, incurring undue financial costs, or delete and recreate the repository—a disruptive and impractical solution. This attack is enabled by breaches of SP3, SP6 and SP10.

4.2.4. Quota Escape. This attack involves the misuse of LFS as a means to bypass the storage quota limitations enforced by the Git platform. By exploiting LFS, the adversary can treat the platform as a form of free network storage, resulting in significant storage and bandwidth costs for the platform provider. Unlike the Quota-Based DoS attack, in which the victim is a specific repository owner, here the platform itself becomes the victim, bearing the financial consequences of storage abuse. This attack involves multiple security properties, including SP3 and SP7 through SP11.

5. Vulnerability Detection Framework

The heterogeneous nature of infrastructures and the diverse functionalities supported across platforms significantly expand the range of potential attack surfaces, thereby complicating the task of universal vulnerability detection. Nevertheless, we observe that despite these differences, platforms consistently share a set of common "Supported Features". This insight guides our approach: by leveraging these supported features, we can systematically identify core security properties that are broadly applicable across platforms. To achieve this, we decompose each security property into a series of targeted checks, thereby constructing a modular detection framework. As illustrated in Figure 3 and detailed in Algorithm 1, our framework begins with initial preparation and feature collection, followed by systematic property checks for each proposed security property. Any violations of these properties are then validated to confirm vulnerabilities and evaluate their potential impact.

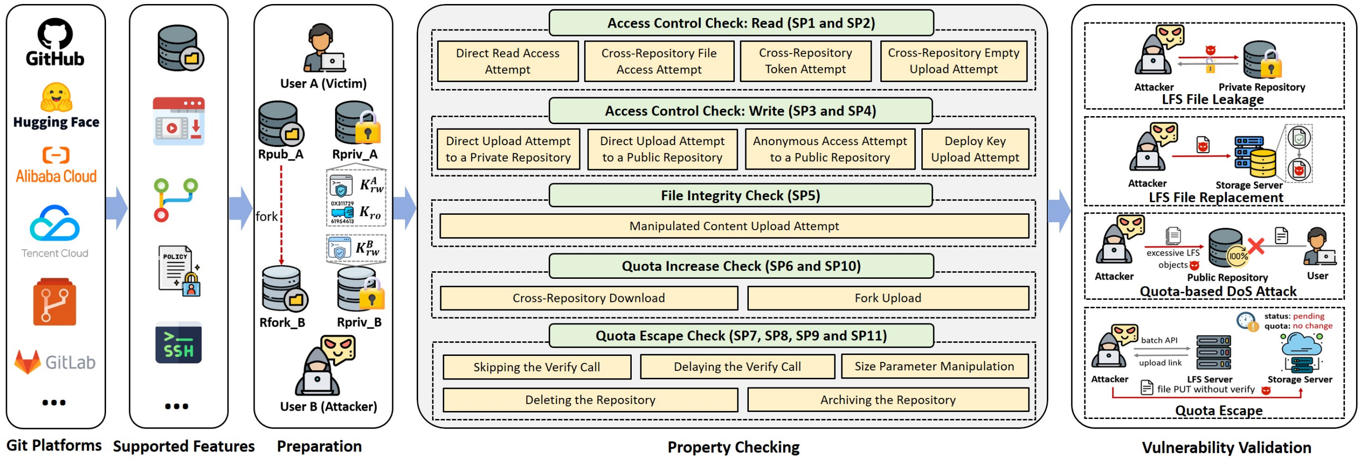


Figure 3: Overview of our work.

Algorithm 1 Security violation and attack detection.

```
Symbol Definitions:
    $AF$: All features (e.g., SSH Keys, Fork Public Repository, Deploy Key)
    $F$: Supported feature set of the platform
    $M$: All property check modules
    $V$: Set of detected property violations
    $A$: Set of detected attacks

Step 1: Preparation and Feature Collection
Preparation (e.g., Creating accounts and repositories)
for each feature $f \in AF$ do
    if CheckSupported($f$) then
        $F \leftarrow F \cup f$
    end if
end for

Step 2: Property Check
for each property check module $M_i \in M$ do
    if CheckApplicable($M_i, F$) then
        $V_i \leftarrow$ DetectViolations($M_i$)
        $V \leftarrow V \cup V_i$
    end if
end for

Step 3: Vulnerability Validation
if $V \neq \varnothing$ then
    $A \leftarrow$ PerformAttacks($V$)
    EvaluateImpact($A$)
end if

Output: Property violations $V$ and Attacks $A$
```

5.1. Preparation

To thoroughly test all the security properties, we outline the following preparations for setting up the testing environment. For each Git platform, two accounts, referred to as User A and User B, are created. User A, acting as the victim, is responsible for creating both a private repository and a public repository, denoted as R^A_{priv} and R^A_{pub}, respectively. Meanwhile, User B, acting as the attacker, creates a private repository, denoted as R^B_{priv}. If the platform supports forking, User B will fork R^A_{pub}, resulting in a forked repository R^B_{fork}. Subsequently, User A generates and adds an SSH key K^A_{rw} as a personal SSH key able to read and write, and configure key K_{ro} as a read-only deploy key for R^A_{priv}. Similarly, User B generates and adds another SSH key K^B_{rw} as his personal SSH key. If the platform does not support SSH keys, we will instead create personal access tokens for the accounts, and employ Basic authentication instead.

After these preparatory steps, we will run our tool to upload a file, denoted as F, to R^A_{priv} using K^A_{rw}, in strict accordance with the protocol, and then download the file and verify it to ensure our client implementation operates as expected. Additionally, we collect information about the platform's supported features, as outlined in Section 2.2. Specifically, we inspect the platform's web interface to verify the availability of various functionalities. Furthermore, by analyzing the traffic generated by the LFS protocol, we confirm whether the platform employs Cloud Storage Direct Upload and whether the LFS endpoints can be predicted.

5.2. Property Checking

Following the preparation phase, we identify violations of the security properties outlined in Section 3. To accommodate the diversity of platforms, we design our property checking process to be modular with scalable test attempts. Moreover, to enhance testing efficiency, we ensure that each property check is enabled only when relevant. The system is also easily extendable, allowing for the integration of new attack methods targeting the same security properties. The majority of these checks are performed automatically by our tool, requiring minimal manual effort, such as quota usage monitoring. If the tool detects a violation of the security properties, such as unauthorized access attempts, we will undertake further vulnerability validation, as detailed in Section 5.3. Below are the detailed modules for property checks and their corresponding test attempts.

5.2.1. Access Control Check: Read.

Given that file F has already been uploaded to R^A_{priv}, we first examine whether User B can access the file content, under the assumption that B possesses the knowledge of the file's SHA-256 hash and size. The following steps outline our approach, which is universally enabled for all Git platforms:

1) **Direct Read Access Attempt**. We use K^B_{rw} to request an access token for reading from R^A_{priv}. If the operation succeeds without raising any errors, it indicates an authentication flaw within the SSH server, which should not grant access tokens for private repositories to unauthorized users.
2) **Cross-Repository File Access Attempt**. We use K^B_{rw} to obtain an access token for R^B_{priv}, then attempt to download from R^B_{priv} using the same SHA-256 hash and size as file F in R^A_{priv}. This test verifies that LFS objects are not shared across repositories, as such sharing would violate SP1 by permitting cross-repository file access without sufficient isolation. Additionally, if the download URL provided by the batch API appears predictable (Predictable LFS Endpoint feature), an additional download attempt will be made by directly accessing the LFS endpoint of R^B_{priv}, allowing for a more thorough investigation of the access control mechanisms of LFS endpoints. This approach has been effective in identifying several leakage vulnerabilities in practice.
3) **Cross-Repository Token Attempt**. We use K^B_{rw} to obtain an access token for R^B_{priv}, and call batch API of R^A_{priv} or directly access the LFS download URL (if predictable) with the acquired token. This test targets misconfigurations in the LFS server that rely solely on the SSH Server to validate user permissions.
4) **Cross-Repository Empty Upload Attempt**. Inspired by the GitLab file leakage vulnerability highlighted in Section 2.3, we use K^B_{rw} to attempt uploading an empty file to R^B_{priv}, specifying the same object ID (oid) as file F. Additionally, we experiment with various `Content-Type` header settings during the upload process, such as omitting the header or providing invalid values, to observe the LFS server's behavior under these conditions, thus assessing compliance with SP2.

5.2.2. Access Control Check: Write.

This section examines whether unauthorized users can upload LFS objects to repositories they do not own. The following steps outline our process:

1) **Direct Upload Attempt to a Private Repository**. Using K^B_{rw}, we request an access token to upload an LFS object to R^A_{priv}. This token is then utilized to call the batch API of R^A_{priv} or directly upload LFS objects if the upload URL is predictable. Additionally, if the SSH server provided a read access token during the prior read attempt, we also use this token to attempt an upload.
2) **Direct Upload Attempt to a Public Repository**. Similarly, we use K^B_{rw} to request two access tokens, one for download and one for upload, to R^A_{pub}. Since public repositories are expected to allow read access, it is anticipated that User B can obtain a read token. However, if the same token also permits file uploads, this would constitute a violation of SP3 and expose an access control vulnerability.
3) **Anonymous Access Attempt to a Public Repository**. Furthermore, we attempt to use an empty Authorization header to upload to the public repository R^A_{pub}, to determine whether an attacker can upload objects without creating an account.
4) **Deploy Key Upload Attempt**. We use K_{ro} to attempt to upload files to R^A_{priv}. This action should be rejected according to SP4, thereby confirming that the deploy key is restricted to read-only access.

The first three checks are applicable as long as the platform supports the **Public Repository** feature, which is commonly available. The final check is only applicable if the platform supports the **Deploy Key** feature.

5.2.3. File Integrity Check.

To evaluate SP5, we conduct a test by uploading an LFS object with deliberately manipulated content, such that its Object ID (SHA-256 hash) does not match its actual content. Upon successful upload of the manipulated file, we then attempt to download it through the web interface to confirm the violation.

If the platform supports the **Cloud Storage Direct Upload** feature, an optimization can be applied. In this case, the Batch API returns a pre-signed URL, allowing the client to upload the file directly to the cloud storage service. By analyzing this upload link, we can infer whether the cloud service enforces file integrity checks, such as validating the `X-Amz-Content-Sha256` parameter in AWS S3. If such checks are absent, a violation of SP5 can be immediately reported. A related case study will be provided in Section 6.4.2.

5.2.4. Quota Increase Check.

For quota related checks, **LFS Usage Query** feature must be supported, otherwise quota usage value cannot be queried. In this section, we assess the potential violation of SP6 and SP10, which would indicate that an attacker can increase the quota usage of repositories owned by other users. Specifically, we attempt to increase the LFS usage of the public repository R^A_{pub} without utilizing victim A's identity. We first record the repository's LFS usage data and then proceed with the following steps to determine if the quota usage increases:

1) **Cross-Repository Download**. If we have confirmed that cross-repository downloads of LFS objects are possible, we analyze the LFS usage data to check whether this download action consumes the quota of the public repository, thus checking whether SP6 is violated.
2) **Fork Upload**. We upload a new LFS object to the forked repository R^B_{fork}, and subsequently examine the quota usage of both R^B_{fork} and its parent repository R^A_{pub}. If the

usage of R^A_{pub} increases while R^B_{fork} remains unaffected, we can confirm the violation of SP10 and the feasibility of a quota-based DoS attack by uploading objects to the attacker's forked repository. This check is applicable only if the **Fork Public Repository** feature is supported.

Additionally, we assess whether the LFS server adheres to SP9 by investigating whether the attacker can manipulate the reported quota usage value. Specifically, we attempt to alter the size parameter during each step of the protocol interactions to detect any potential incorrect quota increments. For instance, if we upload a 1MB file but falsely claim its size as 1GB, and the quota usage of the target repository increases by 1GB, we can confirm a violation of SP9. Such a violation would significantly amplify the impact of a DoS attack by allowing the attacker to artificially inflate the repository's quota usage.

5.2.5. Quota Escape Check. Using user B's identity, we upload a new file to R^B_{priv}, with the objective of bypassing the recording of the upload or registering a smaller file size than was actually uploaded. This is verified by manually checking the quota usage of R^B_{priv} after the upload.

1) **Skipping the Verify Call**. We attempt to skip the verify call during the upload process and check whether the quota is unchanged but the LFS server still permits the download of the object, effectively excluding it from quota management. This violation of SP7 provides attackers with the potential to bypass quota management and abuse the platform.
2) **Delaying the Verify Call**. To evaluate whether SP8 is violated, we upload multiple LFS objects without executing the final verify API call, then after varying time intervals (e.g., 24 hours, 3 days, and 7 days), we complete the pending verify calls and assess if the uploaded files remain accessible for download.
3) **Size Parameter Manipulation**. Building on our previous discussion of SP9, instead of inflating the size parameter, we test whether using a smaller value during various stages of the protocol could result in the LFS server incorrectly registering a reduced quota usage. Additionally, we experiment with negative size parameters to determine whether the recorded quota usage can be decreased.
4) **Deleting the Repository**. To evaluate SP11, we delete and recreate the repository R^B_{priv} with the same name and attempt to download the previously uploaded objects. Additionally, we test whether the LFS objects can still be downloaded from other repositories.
5) **Archiving the Repository**. To explore the possibility that read-only restrictions may not apply to LFS uploads and that increased quota usage may not be accurately recorded in such cases, we attempt to upload LFS objects after archiving the repository R^B_{priv}, followed by an examination of any changes in quota usage. This check is only enabled for platforms supporting **Archive Repository** feature.

5.3. Vulnerability Validation

5.3.1. LFS File Leakage. Violations of SP1 or SP2 often result in LFS File Leakage attacks. Upon detecting such violations during property checking in Section 5.2.1, we confirm the vulnerability by verifying the content of the downloaded file F. Specifically, confirmation is achieved if: (1) User B obtains an auth token for R^A_{priv} and successfully downloads F using the batch API; (2) File F is accessible from other repositories through either the Batch API or a predictable LFS file endpoint; (3) A download request for F with an auth token from another repository, R^B_{priv}, is granted; or (4) a cross-repository empty upload is executed successfully, and the subsequent download yields the correct content of F.

5.3.2. LFS File Replacement. When a violation of SP5 is confirmed, it indicates that manipulated file content has been accepted by the LFS storage server, and is now available for download. Following this, we evaluate the potential threats and explore the extent of the damage this vulnerability could cause.

Using file F in the repository R^A_{priv} as the target, we first upload the manipulated content to an attacker-controlled repository R^B_{priv}, while specifying the same SHA-256 hash as F. We then download F from R^A_{priv} to inspect its content, determining whether this upload action has resulted in the overwriting of the file in the victim's repository. Furthermore, if prior access control checks revealed violations of SP3 or SP4, we explore the possibility of directly overwriting an existing file in the target repository by uploading the modified content. Notably, a violation of SP5 alone is sufficient to enable a supply chain attack, provided the attacker has gained write access to a critical repository. This step aims to evaluate the potential escalation of such an attack and its broader impact.

In scenarios where the Git platform supports malware scanning, we also examine whether this vulnerability can be exploited to bypass this security mechanism. A case study illustrating this type of exploitation will be presented in Section 6.4.2.

5.3.3. Quota-based DoS Attack. A Quota-based DoS attack aims to exhaust the available LFS space in public repositories owned by other users, thereby preventing legitimate users from accessing LFS files in the affected repository. If we identify violations of SP3 or SP4 that allow us to upload files to R^A_{pub}, we monitor the repository's usage increment to confirm the feasibility of a Quota-based DoS attack.

Additionally, violations of SP6 and SP10 provide further avenues to increase the usage of the target public repositories, through methods such as cross-repository downloads and fork uploads, both of which directly contribute to a DoS attack.

Moreover, a violation of SP9 can significantly escalate the damage, which means recorded quota usage can be different than the size of the uploaded file. To investigate this, we employ size parameter manipulation techniques in

TABLE 1: Supported features of Git platforms.

Category	Platform	Public Repository	Public LFS File Downloadable via Webpage/Anon. Git	LFS Quota Policy	LFS Usage Query	LFS Object Deletion	LFS SSH	Deploy Key	Archive Repo	Fork Public Repo
Git-centric Platforms	GitHub	✓	✓ / ✓	1GB/user 1GB bw/month	✓	✗	✓	✓	✓	✓
	Gitee	✓	✓ / ✓	No free quota	✓	✓	✓	✓	✓	✓
	BitBucket	✓	✓ / ✓	1GB/user	✓	✓	✓	✓	✗	✓
	GitCode	✓	✗ / ✓	2GB/user	✓	✗	✓	✗	✓	✓
Self-hosted Git Providers	GitLab	✓	✓ / ✓	10GB/user	✓	✗	✓	✓	✓	✓
	Gitea	✓	✓ / ✓	Not supported	✓	✗	✓	✓	✓	✓
	RhodeCode	✓	✓ / ✓	Not supported	✗	✗	✗	✗	✓	✓
	Gogs	✓	✗ / ✗	Not supported	✗	✗	✗	✗	✗	✓
Cloud Service	Aliyun Codeup	✗	- / -	5GB/repo	✓	✓	✓	✓	✓	✗
	TencentCloud Coding	✓	✗ / ✓	20GB/user	✓	✗	✓	✓	✗	✗
	HuaweiCloud	✓	✓ / ✗	1GB/repo,10GB/user	✓	✗	✓	✓	✓	✓
	Azure Repos	✓	✓ / ✓	No policy docs	✗	✗	✗	✗	✗	✗
AI Platforms	Huggingface	✓	✓ / ✓	No policy docs	✗	✗	✓	✗	✗	✗
	ModelScope	✓	✓ / ✓	No policy docs	✗	✗	✗	✗	✗	✗

each scenario to determine whether such manipulation could enable a low-cost DoS attack against any public repository.

5.3.4. Quota Escape. Quota escape attacks exploit weaknesses in access control and quota management, allowing attackers to upload and download files arbitrarily without proper constraints. These attacks can be carried out through several vectors, such as abusing other users' quota or employing various quota bypass techniques. Specifically, abusing other users' quota can be done via direct uploads (SP3) or fork uploads (SP10). Additionally, SP7 through SP11 offer multiple avenues for quota escape. For example, if the LFS server queries a file's existence from the cloud service without verifying the corresponding database records, skipping the verify API call (SP7) becomes a viable vector for a quota escape attack. We consider the Quota Escape vulnerability validated if the uploaded file remains retrievable and the recorded quota usage is less than the actual file size.

In summary, assessing the security of Git LFS implementations presents unique challenges due to the diversity of platform features and infrastructure configurations. Our framework adopts a modular design that maps security properties to specific, targeted checks based on platform features. This divide-and-conquer approach enables systematic detection of vulnerabilities while maintaining extensibility for new platforms and features.

6. Real-world Evaluation

In this section, we explore the following research questions: ① What features are supported by Git platforms? (Section 6.1) ② Which security properties are violated? (Section 6.2) ③ Do these violations lead to vulnerabilities that facilitate novel attacks? (Section 6.3)

6.1. Git Platforms and Supported Features

To assess the effectiveness of our design and uncover real-world vulnerabilities, we selected 14 Git platforms spanning four distinct categories to conduct detection workflows in real-world environments, shown in Table 1. These platforms were chosen based on their popularity, widespread usage, and availability. Additionally, we considered their diversity in architecture and features to ensure a comprehensive evaluation across different providers. AWS and Google Cloud are excluded due to availability issues, as they are phasing out their Git services and restricting access for new users. Table 1 lists the tested platforms and their supported features, focusing on LFS-related features, including accessibility, quota mechanisms, SSH support, and archiving (Section 2.2). Most platforms support public repositories, with the exception of Aliyun Codeup, which is enterprise-focused. LFS objects are generally downloadable via both the web interface and Git, though some platforms either require user authentication or do not support web-based downloads.

GitHub is the only platform that enforces a bandwidth limit (1GB per month for free-tier users). Among self-hosted solutions, GitLab is the only one that supports LFS quota enforcement, with a 10GB limit on GitLab.com. Furthermore, none of the AI platforms we evaluated document quota policies, likely due to the nature of large datasets and models. Platforms with quota limits allow users to check usage, and most update usage data immediately after uploads, though TencentCloud Coding updates usage data only after a Git commit, while HuaweiCloud may delay updates until the next upload. Only three platforms allow users to view and delete LFS objects, a valuable feature that enables users to manage storage efficiently. When the quota is depleted, this feature offers an alternative to purchasing additional space or recreating repositories.

Most platforms support SSH-based LFS authentication, while others are limited to password or personal access token authentication. More than half support deploy keys, while the others may lack this feature due to early-stage development or alternative workflows (e.g., AI platforms with custom APIs). Archiving and forking are also unsupported by AI platforms. Although TencentCloud Coding and Azure Repos offer archiving, it renders repositories unreadable,

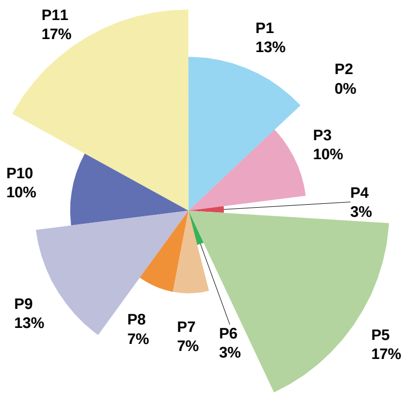


Figure 4: Distribution of violations of LFS Security Properties in 14 platforms. (N=30)

TABLE 2: Distribution of found vulnerabilities across four categories.

ID	File Leakage	File Replacement	Quota-based DoS Attack	Quota Escape	Total
1	0	0	1	1	2
2	0	0	2	2	4
3	0	0	0	0	0
4	0	1	0	1	2
5	0	0	0	1	1
6	1	1	0	1	3
7	0	0	0	0	0
8	1	0	0	1	2
9	0	1	0	2	3
10	0	0	1	1	2
11	1	2	1	3	7
12	1	0	1	3	5
13	0	1	0	0	1
14	0	0	1	3	4
Total	**4**	**6**	**7**	**19**	**36**

which does not meet the requirements for a Quota Escape attack. Therefore, we have marked them as not supporting the Archiving Repository feature in Table 1.

6.2. Observed Violations of Security Properties

Following the detection workflow, our tool identified 30 Security Property violations across 14 Git platforms. Including manual preparation and analysis, evaluating each platform took less than one hour, with the automated component executing in under one minute. To maintain ethical standards and address vulnerability disclosure concerns from stakeholders, we have anonymized platform names by assigning random number IDs in the table. As shown in Table 3 (in Appendix A), the majority of platforms (12 out of 14) violated at least one security property. Notably, one platform (ID 11) violates six security properties, highlighting significant oversights in its LFS server implementation. This creates significant opportunities for attackers to compromise the platform's confidentiality, integrity, and availability. The highest number of violations (five) occurred with SP5 and SP11, indicating widespread neglect in implementing proper file content validation and effective quota mechanisms for repository deletion.

Figure 4 illustrates the distribution of violations for each property. This distribution highlights that nearly all security properties have corresponding violations, underscoring both the effectiveness of our methodology and the critical importance of these properties. While we did not identify any current real-world instances violating SP2, the previously documented CVE-2019-6786 in Section 2.3 represents a historical example of this violation category.

6.3. Uncovered Novel Vulnerabilities

After the vulnerability validation step and manual confirmation, we identified 36 vulnerabilities across 14 Git platforms. The vulnerability counts are listed in Table 2. The aggregate count of vulnerability totals 36, which is higher than the violation count since individual violations may manifest in multiple vulnerabilities. Notably, uploading to public repositories owned by other users often constitutes both a Quota-based DoS attack and a Quota Escape vulnerability, as it drains the owner's quota and enables storage abuse through retrievable files. Despite sharing a common attack vector, these two vulnerabilities operate under distinct threat models and assumptions, and are thus regarded as separate vulnerabilities within the context of our analysis. Furthermore, all of the affected platforms have a broad user base, meaning that security flaws in these systems could affect a large number of users. For instance, the most vulnerable platform, P11, has over 3 million users, making it particularly susceptible to large-scale exploitation.

As shown in Table 2, Quota Escape vulnerabilities represent the predominant category, accounting for 52.7% (19 out of 36) of total identified vulnerabilities and affecting 78.6% (11 out of 14) of the Git platforms analyzed. This category also exhibits the most diverse attack vectors. Among these, six exploit other users' quota, five delete the repository, four manipulate size parameters, two delay the verify call, one archives the repository, and one skips the verify call. This distribution reveals significant weaknesses in quota enforcement mechanisms across Git platforms, indicating systemic vulnerabilities in storage resource protection. The second most common vulnerability category is Quota-based DoS Attacks, with seven vulnerabilities identified across six Git platforms. Upload-based DoS attacks constitute the majority, comprising three Direct Upload and three Fork Upload vulnerabilities, highlighting widespread permission control issues and inherent design flaws in repository forking mechanisms. Furthermore, our analysis revealed a Cross-Repository Download vulnerability in one of the platforms, where the platform automatically deducts quota from public repositories when processing download requests for files that exist in other repositories but are absent from the current repository. Moreover, this platform's implementation implicitly trusts the size parameters specified in download requests, allowing attackers to arbitrarily exhaust storage quota through a single request.

We identified six File Replacement vulnerabilities affecting five Git providers, posing significant risks for supply chain attacks and virus scan circumvention. Our analysis indicates that the globally shared LFS storage architecture exacerbates these vulnerabilities, enabling attackers to overwrite files in popular repositories by uploading malicious content with the same SHA-256 object identifier as the original. Section 6.4.1 and 6.4.2 provide detailed case studies on cross-repository file overwrite and virus scan bypass exploitation, respectively.

Moreover, our investigation revealed four Git platforms susceptible to LFS File Leakage vulnerabilities. Notably, while certain platforms implemented proper permission checks in their SSH server and Batch API infrastructure, the predictable nature of LFS file download URLs, combined with globally shared storage design and inadequate access controls, facilitates unauthorized access. An attacker with knowledge of a file's SHA-256 hash can exploit this vulnerability to obtain any LFS file, circumventing intended access restrictions.

6.4. Case Studies

To clarify the security risks and attack vectors, in this section, we present several case studies on various platforms to explain the vulnerabilities and how attackers can exploit them. Due to space limitations, additional case studies are included in the GitHub repository, which include a Quota Escape attack that involves resetting the repository and a permission check bypass that exploits the Predictable LFS Endpoint feature.

6.4.1. Cross-Repository File Overwrite. RhodeCode [24] is an open-source enterprise code management platform that offers unified security and collaboration for Git, Mercurial, and Subversion repositories. It has gained traction within the software development community, particularly among enterprises necessitating comprehensive source code management capabilities across multiple version control systems. However, during testing, we identified a vulnerability related to Cross-Repository LFS File Overwrite attacks. An attacker could upload manipulated content to his own repository by specifying a SHA-256 hash matching that of a target file in another repository. RhodeCode would accept such uploads, returning the message `{"upload": "ok"}`, effectively overwriting the original file's contents. Further analysis indicates that the root cause lies in the use of shared backend storage paths without repository-level isolation, combined with the absence of content verification against claimed SHA-256 values, violating SP5. This vulnerability poses a significant risk for supply chain attacks, as it does not require any specific permissions from the attacker, and any public repository hosted on the platform is a potential target.

```
+import hashlib

 # Function to upload LFS object with OID validation
 def lfs_objects_oid_upload(request):
+    # Step 1: Check if OID is already in store
+    if store.has_oid():
+        return {'upload': 'ok', 'state': 'in-store'}

+    # Step 2: Initialize SHA256 digest
+    digest = hashlib.sha256()

+    # Step 3: Update digest for each written chunk
     for chunk in request_body:
+        digest.update(chunk)
         engine.write(chunk)

+    # Step 4: Compare computed digest with OID
+    hex_digest = digest.hexdigest()
+    if hex_digest != oid:
+        engine.cleanup()  # Trigger cleanup on mismatch
+        return error_response('oid_mismatch')

-    return {'upload': 'ok'}
+    return {'upload': 'ok', 'state': 'written'}
```

Figure 5: Patch snippet from RhodeCode.

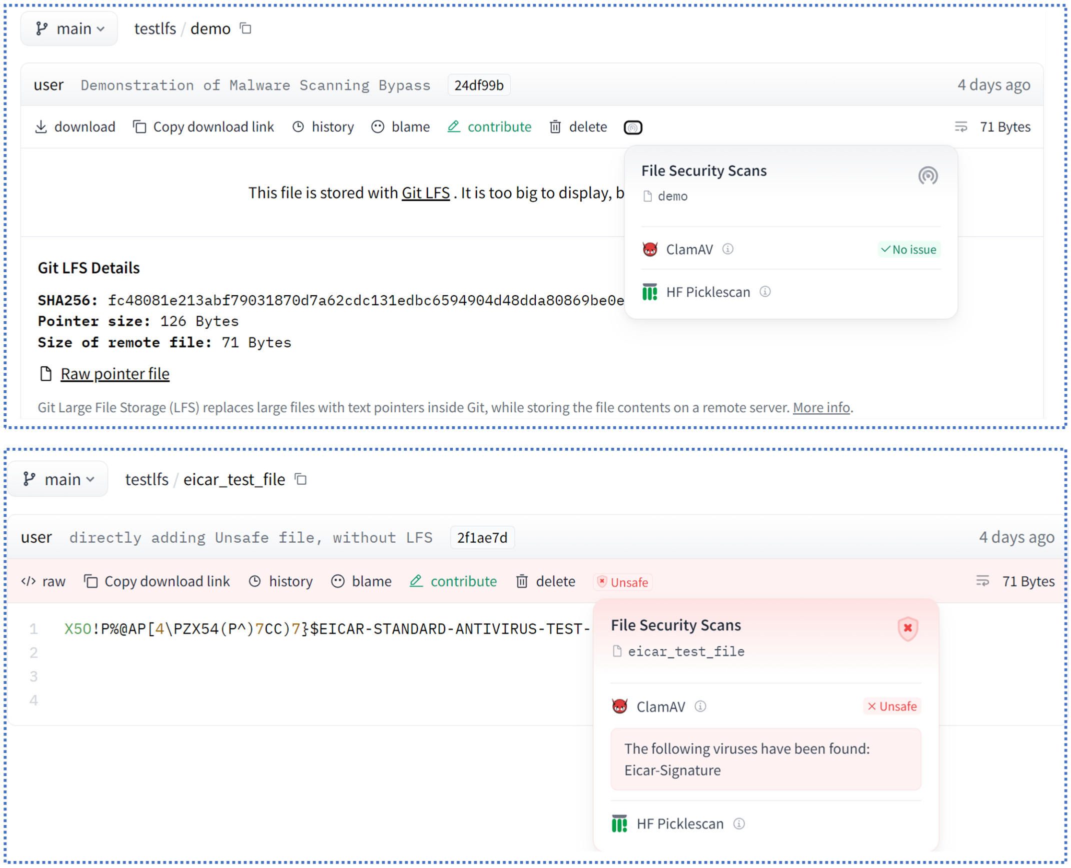
After we reported this vulnerability to the developers, they promptly addressed the issue and released a new version of the software. Figure 5 provides a simplified representation of the corresponding patch [25]. The patch first verifies whether the file being uploaded is already present in the storage, thereby preventing the overwriting of existing files. Additionally, within the file chunk processing loop, the code `digest.update(chunk)` is introduced to compute the SHA-256 hash of the file. Finally, if the calculated hash does not match the claimed object ID (oid), the upload is aborted, an error message is returned, and the uploaded file is cleaned up to prevent it from being saved to storage, thereby ensuring compliance with SP5.

6.4.2. Malware Scanning Bypass. An additional critical implication of the file overwrite vulnerability lies in its potential to bypass security mechanisms, particularly malware scanning protocols. Platform Y (anonymized due to ongoing disclosure and pending resolution), a widely recognized artificial intelligence platform, employs Git LFS for the storage of model weights and datasets. A key feature of this platform is its automated malware scanning, which is triggered upon each new Git commits. In our testing, we discovered that Platform Y uses AWS S3 as its LFS storage server. The Batch API returns pre-signed upload URLs that follow the pattern:

```
https://<redacted_s3_domain>/repos/<repoid_pref
ix>/<repoid>/<sha256>?X-Amz-Content-Sha256=UNSI
GNED-PAYLOAD&X-Amz-Expires=900&X-Amz-Signature=
<signature>&...
```

From this URL, we observed two notable security implications: the upload link remains valid for 900 seconds, and the parameter `X-Amz-Content-Sha256` is set to `UNSIGNED-PAYLOAD`, meaning AWS S3 does not verify the integrity of the uploaded content, violating SP5. This design flaw allows attackers sufficient time to first upload a benign file, complete the Git commit to pass the malware scan, and then overwrite the file with malicious content using the same upload URL.

To validate this vulnerability, we performed an experiment using the `eicar_test_file`, a standard virus scan engine test file [26]. Initial tests confirmed that both direct uploads and LFS-mediated submissions of this file were appropriately flagged as unsafe. However, by employing the file replacement technique described above, we successfully



user `Demonstration of Malware Scanning Bypass` 24df99b 4 days ago

This file is stored with Git LFS . It is too big to display, b

Git LFS Details

```
SHA256: fc48081e213abf79031870d7a62cdc131edbc6594904d48dda80869be0e
Pointer size: 126 Bytes
Size of remote file: 71 Bytes
```

Raw pointer file

Git Large File Storage (LFS) replaces large files with text pointers inside Git, while storing the file contents on a remote server. More info.

user `directly adding Unsafe file, without LFS` 2f1ae7d 4 days ago

```
X50!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-
```

The following viruses have been found:
Eicar-Signature

Figure 6: Platform Y’s web interface shows two files with identical malicious payload. The top file bypassed the malware detection via the LFS file replacement attack, while the directly uploaded file was flagged as unsafe (anonymized display).

evaded the malware scanning process. Figure 6 illustrates comparative screenshots of two files with identical content, demonstrating empirical evidence of the successful malware scanning bypass.

7. Discussion

Ethics Considerations. In conducting security tests on live Git LFS servers, we followed strict ethical guidelines to prevent harm to users and platforms. We restricted our testing to two isolated accounts we created and used the EICAR test file [26], a standard, non-malicious tool, to detect malware scanning bypass vulnerability. With respect to potential DoS vulnerabilities, we relied on reported quota values to determine attack feasibility, avoiding excessive resource consumption or service disruption. All discovered vulnerabilities were promptly reported through responsible disclosure channels, and we are actively working with affected platforms to address these security issues.

Root Cause and Mitigations. The attacks identified in this paper expose significant vulnerabilities in the design and implementation of Git LFS servers. The LFS protocol specification offers only minimal security guidance, delegating critical aspects such as access control models and quota management to Git platform providers. As a result, these vulnerabilities often stem from development oversights and a limited understanding of the LFS infrastructure. In particular, the multi-user, multi-repository nature of modern Git workflows—combined with the multi-step handling of LFS objects—leads many platforms to inadequately enforce access control and integrity validation at each stage. Furthermore, the integration of Git LFS with third-party cloud storage complicates the trust boundary, necessitating strict access control and content validation from all involved parties. Finally, Git LFS may exhibit insecure state transitions, creating opportunities for incomplete mediation and loosely coupled workflow steps, which in turn allow certain APIs to be skipped, delayed, or manipulated.

To enhance LFS security, Git platforms should implement comprehensive permission checks at all endpoints, and enforce content validation using mechanisms like `X-Amz-Content-Sha256` headers or content hash verification (as shown in Figure 5), ensuring LFS objects are protected from tampering. These measures will help prevent Private File Leakage, File Replacement, and Quota-based DoS attacks. To mitigate Quota Escape attacks and prevent resource abuse, platforms should implement precise and reliable quota tracking mechanisms. Additionally, the enforcement of rate-limiting policies and the deployment of robust monitoring systems are recommended to ensure that quota checks and updates are performed synchronously, while also enabling the detection of anomalous storage behavior. When combined with adherence to the 11 Security Properties outlined in this paper, these defensive strategies can substantially reduce the risk of security breaches in LFS implementations, thereby fostering a more secure environment for managing LFS files.

Limitations and Future Work. While this paper presents the first comprehensive security analysis of Git LFS, certain limitations remain. First, the detailed attack techniques we identified may represent a subset of potential vulnerabilities. Other attack vectors may exist that have not been uncovered in our current analysis. However, the framework we developed for testing LFS server security is extensible and can accommodate future discoveries of additional attack vectors. As the security landscape evolves, researchers and practitioners can build on our work to identify new threats and implement defenses accordingly. Second, although we tested 14 popular Git platforms, this coverage may not fully encompass the entire ecosystem of Git platforms that support Git LFS. Nevertheless, we focused on analyzing the most widely used platforms to ensure that our findings have broad relevance. In future work, we plan to extend our testing to include more platforms, especially those that may have smaller user bases but still rely on Git LFS for file management.

8. Related Work

File Storage Security. File storage security is a critical concern, fraught with numerous vulnerabilities, including unauthorized access [27], data leakage [28], [29], and data corruption [30]. These challenges have been under investigation for years. For instance, early research [31] in 2009 delved into data leakage through side-channel attacks, and recent research [32], [33] proposed approaches to understand the security implications of exposed cloud services and cloud storage. Various strategies have been developed to mitigate such security risks, including the management mechanism designed to bolster security protocols [34], innovations in cloud storage that enhance data integrity and confidentiality [35], and mitigations of access-control risks [36]. Moreover, novel protocols and schemes for secure file storage continue to be developed, including CDStore [37], EPCBIR [38] and an efficient public auditing protocol [39]. These works collectively push the boundaries of file storage security, but none of them address the file storage security of Git LFS.

GitHub Security Studies. As the most prominent code hosting platform, GitHub's security is of paramount importance. Existing work on GitHub security spans several key areas, like GitHub Copilot [40], GitHub Actions [41], and malware presence in fork repositories [42]. Besides, Meli *et al.* [43] and Sinha *et al.* [44] both addressed the leakage of secrets in public GitHub repositories, and Saha *et al.* [45] further developed machine learning techniques for better detection of secrets within source code. These studies collectively highlight a multi-faceted approach to addressing security vulnerabilities. However, despite the critical role of LFS in managing large assets and the broader support for LFS across various Git platforms, the security of LFS remains under-explored. There are few documented server-side Common Vulnerabilities and Exposures (CVEs) [7], [15], [16], [17] related to this feature, but they pertain solely to GitLab. This reveals a significant gap in the current research landscape that this paper seeks to address.

9. Conclusion

In this paper, we presented the first comprehensive security analysis of Git LFS implementations, identifying critical vulnerabilities across major platforms. By establishing 11 essential security properties and developing a modular testing framework, we uncovered four novel attack vectors that enable unauthorized access, malware injection, DoS attack and resource abuse. Our analysis revealed 36 previously unknown vulnerabilities across 14 popular Git platforms, highlighting the prevalence of these security issues and demonstrating the effectiveness of our framework as a practical security assessment tool. We recommend that all Git platforms implement comprehensive permission checks at all endpoints, enforce rigorous content validation, and adopt accurate quota tracking systems. Through responsible disclosure and our open-source testing framework, we have not only improved the security of existing platforms, but also laid the groundwork for future security research in Git LFS, which is increasingly critical for managing large files in modern software development workflows.

Acknowledgment

We sincerely appreciate our shepherd and all the anonymous reviewers for their insightful and valuable feedback. This work was partly supported by the NSFC under No. U244120033, U24A20336, 62172243, 62402425 and 62402418, the China Postdoctoral Science Foundation under No. 2024M762829, the Zhejiang Provincial Natural Science Foundation under No. LD24F020002, and the Zhejiang Provincial Priority-Funded Postdoctoral Research Project under No. ZJ2024001.

References

[1] Git LFS, "Git Large File Storage (LFS)," https://git-lfs.com, 2024, accessed Oct. 2024.

[2] StackOverflow, "Is git good with binary files?" https://stackoverflow.com/questions/4697216/is-git-good-with-binary-files/4697279, 2011, accessed Oct. 2024.

[3] S. Just, K. Herzig, J. Czerwonka, and B. Murphy, "Switching to Git: The Good, the Bad, and the Ugly," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 400–411.

[4] GitHub, Inc., "Configuring git large file storage," https://docs.github.com/en/repositories/working-with-files/managing-large-files/configuring-git-large-file-storage, accessed Oct. 2024.

[5] Gogs contributors, "Gogs: Git Large File Storage (LFS)," https://github.com/gogs/gogs/blob/main/docs/user/lfs.md, accessed Oct. 2024.

[6] Hugging Face, "Getting started with repositories," https://huggingface.co/docs/hub/repositories-getting-started#set-up, accessed Oct. 2024.

[7] MITRE, "CVE-2019-6786: GitLab LFS Incorrect Access Control Vulnerability," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6786, 2019, accessed Oct. 2024.

[8] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[9] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *25th Annual Network And Distributed System Security Symposium (NDSS 2018)*. Internet Soc, 2018.

[10] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein, "Poison frogs! targeted clean-label poisoning attacks on neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[11] CODING, "Product Pricing," https://coding.net/pricing, accessed Oct. 2024.

[12] MITRE, "CVE-2022-24826: Windows-specific path traversal vulnerability in Git LFS 2.12.1-3.1.2 enables arbitrary code execution through malicious executables when target programs are missing from PATH." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-24826, 2022, accessed Oct. 2024.

[13] ——, "CVE-2021-21300: Git versions 2.14.2+ vulnerable to arbitrary code execution via symlink and Git LFS manipulation on case-insensitive filesystems." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21300, 2021, accessed Oct. 2024.

[14] ——, "CVE-2021-21237: Git LFS before v2.13.2 on Windows is vulnerable to arbitrary code execution when processing malicious repositories containing git.bat/git.exe files, due to Go's command execution behavior." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21237, 2021, accessed Oct. 2024.

[15] ——, "CVE-2024-3035: A permission check vulnerability in GitLab allowed unauthorized access to repositories via LFS tokens," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-3035, 2024, accessed Oct. 2024.

[16] ——, "CVE-2020-13355: An issue has been discovered in GitLab CE/EE affecting all versions starting from 8.14. A path traversal is found in LFS Upload that allows attacker to overwrite certain specific paths on the server." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13355, 2020, accessed Oct. 2024.

[17] ——, "CVE-2020-10081: GitLab LFS import process allows unauthorized access to LFS objects," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10081, 2020, accessed Oct. 2024.

[18] Amazon Web Services, Inc., "Amazon S3," https://aws.amazon.com/s3/, accessed Oct. 2024.

[19] GitHub, "About storage and bandwidth usage," https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-storage-and-bandwidth-usage, 2024, accessed Oct. 2024.

[20] A. Larionov, "Git LFS Limitations," https://github.com/git-lfs/git-lfs/wiki/Limitations, 2023, GitHub Wiki page, last updated Sep. 2023.

[21] Maxim Ivanov, Nick Thomas et al., "GitLab workhorse issue #197: Users can access the content of any LFS object when only OID and size are known," https://gitlab.com/gitlab-org/gitlab-workhorse/-/issues/197, 2018, accessed Oct. 2024.

[22] Hugging Face, "huggingface-hub 0.26.2," https://pypi.org/project/huggingface-hub/0.26.2/, 2024, accessed Nov. 2024.

[23] ——, "Hugging Face Docs: Malware Scanning," https://huggingface.co/docs/hub/en/security-malware, 2022, accessed Oct. 2024.

[24] RhodeCode, "RhodeCode: Enterprise Code Management for Hg, Git, SVN," https://rhodecode.com/, 2024, accessed Oct. 2024.

[25] ——, "Commit a680a605 - RhodeCode VCS Server," https://code.rhodecode.com/rhodecode-vcsserver/changeset/a680a60521bf02c29413d718ebca36c4f692ea4a?diffmode=unified, 2024, accessed Oct. 2024.

[26] European Institute for Computer Antivirus Research (EICAR), "EICAR Anti-Malware Test File," https://www.eicar.org/download-anti-malware-testfile/, 1998, accessed Oct. 2024.

[27] A. Markandey, P. Dhamdhere, and Y. Gajmal, "Data access security in cloud computing: A review," in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*. IEEE, 2018, pp. 633–636.

[28] A. Syed, K. Purushotham, and G. Shidaganti, "Cloud storage security risks, practices and measures: A review," in *2020 IEEE International Conference for Innovation in Technology (INOCON)*. IEEE, 2020, pp. 1–4.

[29] T. Bhatia and A. Verma, "Data security in mobile cloud computing paradigm: a survey, taxonomy and open research issues," *The Journal of Supercomputing*, vol. 73, pp. 2558–2631, 2017.

[30] D. Zhe, W. Qinghong, S. Naizheng, and Z. Yuhan, "Study on data security policy based on cloud storage," in *2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity), High Performance and Smart Computing (HPSC) and Intelligent Data and Security (IDS)*. IEEE, 2017, pp. 145–149.

[31] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.

[32] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Credit karma: Understanding security implications of exposed cloud services through automated capability inference," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6007–6024.

[33] Y. Chen, Y. Li, Y. Lu, Z. Pan, Y. Chen, S. Ji, Y. Chen, Y. Li, and Y. Shen, "Understanding the security risks of websites using cloud storage for direct user file uploads," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 2677–2692, 2025.

[34] M. T. Khan, C. Tran, S. Singh, D. Vasilkov, C. Kanich, B. Ur, and E. Zheleva, "Helping users automatically find and manage sensitive, expendable files in cloud storage," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1145–1162.

[35] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich, "Proving confidentiality in a file system using DiskSec," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 323–338.

[36] K. Borgolte, T. Fiebig, S. Hao, C. Kruegel, and G. Vigna, "Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates," in *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*, P. Traynor and A. Oprea, Eds. Internet Society (ISOC), 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23327

[37] M. Li, C. Qin, and P. P. Lee, “CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal,” in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 111–124.

[38] Z. Xia, N. N. Xiong, A. V. Vasilakos, and X. Sun, “EPCBIR: An efficient and privacy-preserving content-based image retrieval scheme in cloud computing,” *Information Sciences*, vol. 387, pp. 195–204, 2017.

[39] J. Shen, J. Shen, X. Chen, X. Huang, and W. Susilo, “An efficient public auditing protocol with novel dynamic structure for cloud data,” *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2402–2415, 2017.

[40] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the keyboard? assessing the security of github copilot’s code contributions,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[41] H. O. Delicheh and T. Mens, “Mitigating security issues in github actions,” in *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, 2024, pp. 6–11.

[42] A. Cao and B. Dolan-Gavitt, “What the fork? finding and analyzing malware in github forks,” in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, San Diego, CA, USA, April 2022.

[43] M. Meli, M. R. McNiece, and B. Reaves, “How bad can it git? characterizing secret leakage in public github repositories.” in *NDSS*, 2019.

[44] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, “Detecting and mitigating secret-key leaks in source code repositories,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 396–400.

[45] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, “Secrets in source code: Reducing false positives using machine learning,” in *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 2020, pp. 168–175.

TABLE 3: Security Property violations across 14 Git platforms.

ID	SP1	SP2	SP3	SP4	SP5	SP6	SP7	SP8	SP9	SP10	SP11	Total
1			✗									1
2			✗							✗		2
3												0
4					✗						✗	2
5								✗				1
6	✗				✗						✗	3
7												0
8	✗										✗	2
9					✗		✗		✗			3
10						✗			✗			2
11	✗		✗	✗	✗				✗		✗	6
12	✗						✗			✗	✗	4
13					✗							1
14								✗	✗	✗		3
Total	4	0	3	1	5	1	2	2	4	3	5	30

Appendix A. Security Property Violations Table

As discussed in Section 6.2, Table 3 provides an overview of the security property violations observed across 14 different platforms. The table lists 11 security properties (SP1 through SP11) and marks violations with an "✗" for each platform ID where a specific security property was violated. In total, we identified 30 security property violations, with SP9 and SP11 having the highest number of violations (five each). This table highlights the widespread security issues in LFS implementations across various platforms.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper performs a thorough analysis of Git LFS platforms defining eleven security properties which should be maintained to ensure security. Using these security properties, the paper analyzes 14 widely-used Git LFS platforms identifying 36 vulnerabilities across them.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability

B.3. Reasons for Acceptance

1) The paper identifies an impactful vulnerability across Git LFS implementations. To do so it presents a thorough and systematic analysis of Git LFS implementations, defining eleven security properties which are critical for maintaining security across the service.
2) The paper creates a new tool which can be leveraged to thoroughly analyze the security properties across different Git platforms. The authors have evaluated 14 widely-used Git LFS platforms using their tool and shown multiple vulnerabilities across these platforms.

B.4. Noteworthy Concerns

1) The security properties mostly focus on denial of service which does not seem as important as leaking sensitive data. Also, most of the identified vulnerabilities are related to the DoS category which is less security-critical than maintaining data integrity and confidentiality.
2) Some reviewers were concerned that the paper focuses on implementation issues and not design choices in the platforms, therefore limiting the contribution of the paper.

Appendix C. Response to the Meta-Review

The meta-review notes that the security properties discussed in the paper primarily focus on denial-of-service (DoS) attacks, which are considered less critical. To clarify, our analysis encompasses a broader range of security properties, including private file leakage, unauthorized access, and integrity violations—issues that are fundamental to preserving data confidentiality and trust. Notably, we have discovered multiple real-world vulnerabilities involving file leakage and unauthorized file replacement, demonstrating that the impact of our findings extends well beyond DoS-related concerns.

The meta-review also suggests that the paper emphasizes implementation flaws rather than platform design choices. In response, we emphasize that our analysis identifies critical design-level vulnerabilities in existing Git platforms, particularly in the areas of access control enforcement and quota management.

Specifically, the Git LFS specification provides only minimal guidance for request authentication, leaving access control models and quota mechanisms to be defined by individual platforms. Our analysis reveals three core design flaws:

1) **Insufficient Fine-Grained Authorization and Integrity Enforcement.** Many Git platforms fail to implement robust, stage-by-stage access control and integrity validation in the LFS workflow. Given the multi-user, multi-repository nature of modern development and the multi-step handling of LFS objects, security checks must be applied consistently throughout the process. However, we find that several platforms rely on coarse-grained entry-point checks, neglecting to validate user permissions and object integrity across intermediate operations.
2) **Overreliance on External Cloud Storage and Implicit Trust.** The integration of Git LFS with third-party cloud storage providers complicates the trust boundary. In many cases, platforms delegate file uploads via pre-signed cloud storage URLs, implicitly trusting both the client and cloud provider to enforce correctness and integrity. This architectural decision weakens the platform's control over security-critical operations, especially when SHA-256 content validation is not supported by the cloud service.
3) **Insecure State Transitions and Workflow Decoupling.** Git LFS platforms frequently exhibit loosely coupled workflow steps and incomplete mediation. For instance, the verification of uploaded objects often occurs through a separate API, which can be skipped, delayed, or manipulated by an attacker. If platforms allow access to uploaded content prior to verification, or if quota accounting is deferred and poorly synchronized, this enables quota evasion and persistent unauthorized access. These vulnerabilities stem from flawed state management logic, not merely implementation oversights.

In summary, our work highlights systemic weaknesses in both the Git LFS specification and the architectural design of platform implementations. We are actively working to inform improvements to the LFS specification and to guide more secure infrastructure designs across Git platforms.