End-to-End Encrypted Git Services

Ya-Nan Li
The University of Sydney
Sydney, Australia
yanan.li@sydney.edu.au

Yaqing Song
UESTC
Chengdu, China
yaqings@163.com

Qiang Tang
The University of Sydney
Sydney, Australia
qiang.tang@sydney.edu.au

Moti Yung
Google & Columbia University
New York, United States
moti@gmail.com

ABSTRACT

Git services such as GitHub, have been widely used to manage projects and enable collaborations among multiple entities. Just as in messaging and cloud storage, where end-to-end security has been gaining increased attention, such a level of security is also demanded for Git services. Content in the repositories (and the data/code supply-chain facilitated by Git services) could be highly valuable, whereas the threat of system breaches has become routine nowadays. However, existing studies of Git security to date (mostly open source projects) suffer in two ways: they provide only very weak security, and they have a large overhead.

In this paper, we initiate the needed study of efficient end-to-end encrypted Git services. Specifically, we formally define the syntax and critical security properties, and then propose two constructions that provably meet those properties. Moreover, our constructions have the important property of platform-compatibility: They are compatible with current Git servers and reserve all basic Git operations, thus can be directly tested and deployed on top of existing platforms. Furthermore, the overhead we achieve is only proportional to the actual difference caused by each edit, instead of the whole file (or even the whole repository) as is the case with existing works. We implemented both constructions and tested them directly on several public GitHub repositories. Our evaluations show (1) the effectiveness of platform-compatibility, and (2) the significant efficiency improvement we got (while provably providing much stronger security than prior ad-hoc treatments).

KEYWORDS

Secure cloud storage; End-to-End security; Version control systems

1 INTRODUCTION

Git services have become indispensable in the IT industry, facilitating project management and collaboration among multiple (potentially a large number of) entities via hosting platforms like GitHub, Bitbucket, GitLab, Azure Repos services, and many others. In these platforms, the entirety of a project's data, including files (such as code and documentation) and directory structures, constitutes a repository. Moreover, in a Git repository, the file data includes each version of all tracked files and their corresponding directory structure. Authorized users can access and edit the shared repository data in a local Git client and then synchronize to the Git server via pull/push operations. Repositories can be public or private, while private ones allow project owners to manage visibility and keep data hidden from the public.

The rising demand for *end-to-end* security. Privacy is undoubtedly a great concern for both individual users and enterprises that collaborate over hosting Git platforms for projects that may contain sensitive information and/ or trade secrets. The situation becomes particularly more alarming in the AI era when repositories become very powerful, containing AI models that are trained on code and data stored in Git repositories or even that directly provide coding assistance (e.g., GitHub Copilot).

Unfortunately, in existing Git hosting platforms, the data, even in private repositories, is visible to the server itself. Even if Git servers may not actively disclose users' data (e.g., for compliance and reputation) and have taken actions to protect data against external attackers (e.g., encrypting the data using the server's own key), the usual risk of data breach (due to external attacks or internal misbehavior of staff) is paramount nowadays.

Moreover, the collaborations on Git services essentially form a supply chain of software development (open source or not, and more broadly online collaboration): any unauthorized modification could have detrimental impacts on the final "product". The potential issues are partially solved by a few Git platforms, such as GitHub [20], GitLab [27], Gitea [21], and Bitbucket [35] that started to support an optional verified commit signature to authenticate the author of each edit but most versions are not verified. It follows that ensuring integrity, proper write access control, and even authenticity of each edit in Git services is also of utmost importance. Unfortunately, current practice mainly relies on the honesty of the Git servers/ platforms, which might even have conflicts of interest on certain projects, to ensure that each repository version is integrated and written by the shown author.

The above situation highlights the need for *end-to-end* (E2E) security [1] in Git services that guarantees critical security properties, even against possibly *corrupted* servers. Indeed, a few industrial projects have been introduced with the aim of moving toward an E2E secure system supporting Git service and online collaborations such as [4, 8, 9, 32, 34].

We note that similar security concerns were widely recognized in *secure messaging*, where there is a long line of research work [1, 12, 17, 18, 29–31, 36]. These works attempt to rigorously realize E2E security in different settings and analyze potential vulnerabilities in widely deployed tools. Recently, a sequence of work has also emerged [5, 15], initiating the study of E2E security for *cloud storage*. In the latter setting, further complications arise due to some features that cloud storage possesses, such as sharing among multiple users, portability via password-based authentication, and subtle vulnerabilities that led to attacks on real-world products, such as [2, 6, 28].

E2E secure Git service is not yet available. Despite the recent progress in relevant applications and strong demand, E2E secure Git service is currently out of reach of current techniques and methods.

[1]We will use the standard terminology "end-to-end encrypted" Git services; however, it goes beyond confidentiality and includes integrity and more security properties.

Insufficiency of using E2E encrypted cloud storage directly. First, one may wonder whether deploying Git servers on E2E secure cloud storage immediately solves the problem. Unfortunately, the situation is more complex than we thought. These recent E2E encrypted storage solutions [5, 15] are at a very early stage of their own development and are insufficient in both functionality and security.

In terms of functionality, the most common operations in Git services are "push" and "pull," used to upload/retrieve missing versions to the server/client. Unlike E2E encrypted cloud storage systems, which return only the content specified in the request without computation, the Git "pull" operation requires the server to compute and return the minimal missing parts, as the client cannot determine which parts are missing or minimal.

In terms of security, E2E encrypted cloud storage only considered basic security properties for *static* data. While for Git services, storage exhibits a feature that data is constantly *updated*. This not only further complicates the already complicated security properties, such as confidentiality (and basic integrity), but also raises new (yet natural) security requirements on "access control." As Git service is a distributed collaborative environment, some basic access control actions for managing authorized users (without relying on an honest Git server), called unforgeability – which we will discuss below soon, are not only required for identifying the author of edits but also has direct impacts on confidentiality. We note that this type of operation might also be needed for cloud storage but has not yet been studied in the literature.

Furthermore, efficiency is a very important consideration. Unlike traditional cloud storage systems, which often store only a limited number of file versions (for example, Google Drive keeps only 100 versions), Git servers keep the whole chain of edits for users to track the history. This creates a strong incentive to deduplicate data across versions to minimize storage costs. Naively adopting encryption as in E2E encrypted cloud storage to Git service, i.e., treating each file within each repository version as a new file to encrypt, transfer, and store, may incur very high costs for users on computation in file encryption, on communication in data transfer, and on storage in local client and remote servers (note that usually free cloud storage is only provided with limited space).

Security risks in existing ad hoc secure Git service designs. Numerous industrial products and some research papers exist, attempting to address the data protection needs of Git services, including Keybase [9], Git-secret [34], Git-crypt [4], and others. Despite the fact that some of these tools have received thousands of stars (being saved with stars) in GitHub, none of them have been rigorously analyzed. Jumping ahead, we can easily see (in Table 1) important security properties that actually fail, *especially when we do not place blind trust in the storage server.* We elaborate on this below.

First, even very basic confidentiality requires care when efficiency improvements are considered. For example, Git-crypt [4] and Gringotts [37] tried to save storage costs by utilizing deterministic encryption schemes to enable data compression on the ciphertext. This is at the (obvious) cost of privacy. It is well known that deterministic encryption offers weak protection, as it trivially leaks pattern information that the same data has the same ciphertext. Indeed, recent research has demonstrated how to abuse such leakage via "injection attacks" on E2E encrypted applications, such as backup, to, in fact, expose the content [24].

Furthermore, the conventional integrity of the *repository* (jumping ahead, actually a weaker form of unforgeability that is only against a corrupted server and users without legitimate access) may easily fail, too. Several systems simply employ the hybrid encryption paradigm as seen in Git services like Git-Secret [34] and Keybase [9]. To reduce storage costs, Git-secret [34] even allows the users to choose which files to encrypt (e.g., sensitive data only), while leaving others still in the clear. It is easy to see that in either case, repository integrity cannot be achieved, as a corrupted server could attack via injection and deletion. For example, a malicious server injects a new data encryption key using the receiver's public key and adds the corresponding encryption of new data. Attackers can also simply delete certain files of a repository version without being detected. What is worse, once the maliciously inserted data encryption key is used by the receiver to encrypt the next version, it further breaks the data confidentiality.

Additionally, existing Git services' features for tracking version history and authorship, as well as read-write access separation, are vulnerable to attacks, highlighting the lack of truly secure Git services. Currently, author tracking and read-write access separation mainly rely on a trusted Git server, allowing malicious users and a corrupted server to falsify authorship, write on behalf of others, frame honest users, or break the read-only limit to write. To prevent such attacks, E2E encrypted Git services require an "unforgeability" property related to edit-source authenticity and read-write access separation (defined later). This ensures that attackers, including corrupted servers or users with write access, cannot misattribute edits without detection and that read-only attackers cannot perform writes. [2]

We stress that these security vulnerabilities in ad hoc designs are not just of theoretical interest. Similar situations exist in cloud storage and secure messaging, where multiple practical attacks were identified [2, 24]. These highlight the need to design a formally analyzed E2E encrypted Git service.

Dealing with overhead and compatibility. If we are to ignore performance, designing an E2E secure Git service is easy via “trivial-enc-sign”. That is, for each edit, a user simply performs the following commands: fetch the latest ciphertext version, verify and decrypt to get the plaintext version, edit the files of the plaintext version to get the new plaintext version, then completely re-encrypt the new plaintext version, sign, and upload. Instead of this trivial operation (which can be followed on plaintext Git services without the cryptography), the repetitive nature of Git updates (with significant overlaps across versions) has been carefully leveraged in systems and led to the current implementation of plain Git adopting various measures for reducing complexity and enhancing performance.

We note that despite several existing systems (still with various security issues shown in Table 1) that have tried to provide security for Git services with reduced complexity (compared to the trivial secure solution), overheads for each edit are still significant. This is because they still operate on files, thus making the overhead

[2]Unforgeability is also essential in secure cloud storage systems. Unfortunately formal treatments [5, 15] have not captured this security, assuming an all-or-nothing type of access (either no access or full access), with no guarantees on the source of edits.

relevant to edited files or even the whole repository, even if only one character was changed. A more careful and fine-grained treatment may give us the opportunity to minimize the overhead (while maintaining E2E security), e.g., it makes sense to require operations to only be proportional to each actual edit.

Another important practical property (also recognized in [15]) is *platform compatibility* with existing infrastructure; in our setting, existing Git services include GitHub and Bitbucket servers. This is an important property often ignored in many theoretical works. In fact, with this compatibility, current users can employ E2E encrypted Git services by simply installing a new secure Git client and directly using Git servers that current Git services provide to do Git operations (in most cases, services are not accessible except based on the existing defined queries).

We summarize detailed comparisons in Table 1 and Section 1.3.

The discussion above showed that secure Git services of the E2E nature are really beyond the current state of the art. Hence, in this paper, we tackle the following challenge: *identify and formalize critical security properties of an E2E encrypted Git service, and give provably secure constructions that are both with minimal overhead and platform-compatible with existing Git servers.* [3]

1.1 Our results

- We present formal syntax and security models for E2E encrypted [4] Git service. Particularly, we propose two main security properties of *data confidentiality* and *repository unforgeability*, each with a weaker variant. All properties are against a malicious Git server and unauthorized users, while *unforgeability* is even further against malicious insiders.
- We give two constructions that provably meet data confidentiality and repository unforgeability (with the caveat that the first construction satisfies only a weaker confidentiality), and both are fully compatible with existing Git servers (including all Git hosting platforms like GitHub server) as shown in Figure 1 and standard cryptographic libraries. Moreover, these two constructions achieve security with minimal overhead that is relevant only to the edits (instead of the whole repository or files), and have different efficiency performances for different edit patterns on the managed projects.
- We implement our two constructions and carry out extensive experiments on popular GitHub repositories to evaluate computation, communication, and storage costs. Our experiment results show that our constructions perform better than the "naive" solution and those using deterministic encryption.

1.2 Technical overview

Git service architecture and defining security properly. We illustrate the architecture of plain and our E2E secure Git services in Figure 1, highlighting the syntax definition and our general principle of *platform compatibility* with Git services.

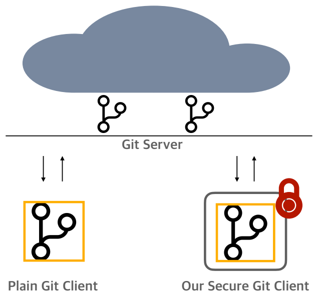


Figure 1: The architecture of plain/secure Git service

Abstracting out authentication. Unlike recent works on E2E secure cloud storage [5, 15] that blend password authentication into the overall security model, we abstract out authentication as a standalone black-box module. This approach simplifies the analysis by focusing on the core security properties of E2E secure Git services, which are already complex. Furthermore, authentication methods vary widely, including password login, two-factor authentication, device-based authentication, token-based authentication, and more. By treating authentication as a parameter, our framework can naturally inherit its security, enabling straightforward generalization.

Formalizing security properties. The basic intuition of E2E secure Git services is to "emulate" an ideal access control as if it is enforced by a trusted server (which we do not have now) to capture real-world attacks from malicious servers and unauthorized users.

Data confidentiality mimics the conventional IND-CPA security. However, additional subtleties arise when considering multi-user data sharing and frequent data updates. Both require us to provide the adversary with extra capabilities to flexibly interact with honest users via sharing, updating, and more. Moreover, certain "metadata" regarding updates should also remain concealed from the malicious server. Specifically, the update/edit operations, e.g., insertion or deletion, as well as the precise edit positions, including which lines or words are altered, should remain hidden. Essentially, while the server is aware that users are performing operations on specific files and sizes, it remains oblivious to the update details. We remark that only our SGitChar satisfies this strong confidentiality, while the SGitLine construction only satisfies a weaker version that only hides the data (not the "metadata") of the edit.

Repository integrity is a second natural property that ensures users can verify whether each version of a repository is intact and has not been modified by outsiders, including malicious servers and unauthorized users. This is clearly stronger than file-wise integrity, as a malicious server may delete a set of files to save storage without violating the latter, weaker form of integrity. We may also easily upgrade the notion to make sure the whole repository history remains intact. To capture the natural need for edit-source authenticity and read-write access separation, we further strengthen integrity to be against *malicious insiders* who may indeed have write permission. We call the strong integrity *repository unforgeability*. We remark that while repository integrity is weaker, it captures the existing integrity guarantees commonly provided in secure cloud storage.

[3]We note that there might be potential risks of misuse with E2E security. Addressing these risks while maintaining security for the majority of the innocent users requires additional countermeasures, which seem to require what we advocate herein, but which are beyond the scope of this paper.
[4]The term "encrypted" is used in a broader sense to refer to protection through cryptographic tools, not just encryption alone.

Table 1: Comparison with the state-of-the-art "encrypted" Git services.

Schemes	Confidentiality	Integrity#	Unforgeability	Storage increase per version†	Client enc cost per update‡	Comm cost per update§	Compatibility**
Git-crypt* [4]	✗	✗	✗	$n_f L$	$n_f L$	$n_f L$	✓
Gringotts* [37]	✗	✗	✗	$n_f \ell_1$	$n_f L$	$n_f \ell_1$	✗
Git-secret [34]	✓⋆	✗	✗	$n_f L$	$n_f L$	$n_f L$	✓
Git-re-gcrypt [8]	✓	?	?	$n_f L$	$n_f L$	$n_f L$	✓✗
Disac [38]	✓	?	?	$n_f L$	$n_f L$	$n_f L$	✓
Keybase-Git [9]	✓	?	?	$n_f L$	$n_f L$	$n_f L$	✗
Trivial-enc-sign	✓	✓	✓	R	R	R	✓
Our SGitLine	✓⋆⋆	✓	✓	$n_f \ell_1$	$n_f \ell_1$	$n_f \ell_1$	✓
Our SGitChar	✓	✓	✓	$n_f \ell_2$	$n_f \ell_2$	$n_f \ell_2$	✓

n_f denotes the number of changed files per repository version, and L, R denotes the (average) size of files and repository, respectively. ℓ_1, ℓ_2 denote the average size of line change and character change per file update. Usually, each update works on a small portion of some files; thus, usually, $\ell_2 \ll \ell_1 \ll L$ and $n_f L \ll R$.
# Integrity denotes the conventional integrity of repository data, which is also a weaker version of unforgeability.
† Repo storage increment per version measures the average storage increases for updating to a new version.
‡ Client enc cost for update measures the rough computation cost of a client in each version update.
§ Comm cost per update measures the user-dominating communication cost of each version update.
* means the storage cost of the scheme may be slightly smaller due to compression on deterministic encryption.
** Compatibility asks if the Git service is compatible with existing Git hosting platforms, e.g., GitHub, Bitbucket, etc., and supports all basic Git operations, including commit, push, pull, fetch versions and objects, merge, etc. ✗: not compatible with Git server; ✓: compatible with Git server with full Git operations; ✓✗: compatible with Git server with limited Git operations.
⋆ means the corresponding security is conditional, as it is left to users to decide which part to encrypt.
⋆⋆ means a weaker version of our confidentiality definition.
? means that the security is unclear at the moment since there is no formal security analysis or obvious attack.

This weaker property is reasonable for single-user repositories or settings with all-or-nothing access control. The stronger property is needed in a multi-user collaborative setting like Git service, while currently it is not discussed in E2E encrypted storage literature where read-write access comes as an all-or-nothing flavor (but it exists in some secure group messaging work [1, 29, 30]).

Secure and efficient constructions with compatibility. As mentioned above, there are naive solutions that always download (`pull`) the whole repository before editing, then apply all needed cryptographic operations on the *whole repository* and upload (`push`). It is clear that this comes with a high cost. Namely, editing even a single character in one file results in the increment of a full-size repository (for both server storage and client communication/computation). Meanwhile, in conventional (plain) Git services, the cost could be minimized, as simple compression tricks can be applied. For example, during the `push` execution, the differences between two versions would be computed by a `Diff` operation on the client, and only the resulting "delta" (difference) induced by edits is uploaded.

File-wise treatment for secure Git services is possible (indeed adopted in most of the existing systems [8, 34]). However, they still mostly incur large overhead (proportional to file size instead of the difference size); further, optimizations to reduce overhead already cause various security vulnerabilities, as we briefly discussed above (and also in Table 1). The reason lies in the dilemma that if naively encrypting (say using standard authenticated encryption) the updated file as a whole, then during `push`, the two encrypted files would look completely irrelevant; the actual difference cannot be computed. On the other hand, if encryption is applied in a finer granularity, e.g., line-wise or character-wise, security risks increase due to greater leakage on operation type, position, and edit length. Moreover, the requirement of repository integrity and the enforcement of read-write access separation and edit-source authenticity add complexity.

To address the security vulnerabilities in existing schemes caused by overhead optimizations and summarized with three security properties in Table 1, our first construction, SGitLine, applies standard encryption at a finer granularity within files, specifically at the line level. Encrypting by line, a natural structural unit of files, helps preserve data confidentiality while reducing the update overhead, simplifying version tracking, and enabling more compact storage.

To achieve unforgeability (thus also integrity), enforcing read-write access separation, and tracking the source of each edit, we simply need some publicly verifiable mechanism for each repository version. We leverage digital signatures to sign on a whole version of the encrypted repository following the hash-and-sign paradigm. Since the hash of the whole version is generated by Git commit anyway, the signing cost is constant and small. Then, the encryption we apply could be a standard IND-CPA secure cipher instead of authenticated encryption. This fairly simple encrypt-then-sign paradigm was recently proven for secure "symmetric signcryption" [30]. And SGitLine may offer better efficiency in certain scenarios because each single version is history-free and not relevant to previous updates. However, line-wise encryption suffers from a drawback in confidentiality: it exposes information about update operations and positions during line insertions and deletions (thus only achieves weaker confidentiality). In addition, the communication costs of updating one version depend on the number and length of the modified lines, even if only one character is modified in each line.

To deal with the efficiency-confidentiality dilemma, we dig deeper into Git. When users push a new version after edits, current systems basically send a whole new encrypted file because the built-in compression in both Git client and Git server cannot properly work on ciphertext. We make a simple observation that we may create the ciphertext in a form that helps the deduplication (on ciphertext instead of plaintext, thus not influencing confidentiality).

In our main construction SGitChar, we propose a "Diff-then-Enc-then-Sign" paradigm that, after editing the pulled file, we let the client run a version of `Diff` algorithm that identifies the differences Δ at the character level (e.g., which position, which operation, on which character) with the previous version. Then the client encrypts Δ to obtain C^*, while he pushes $C||C^*$, where C is the ciphertext just pulled before modification. Of course, the whole version is always signed before push. In this way, the built-in deduplication mechanism will remove C and only upload C^* when executing push. Since all details, including update operations and content, are encrypted, SGitChar satisfies our data confidentiality, while unforgeability holds as before. Moreover, since the update only sends C^*, the overhead is still only relevant to the difference Δ (independent of the file).

Obviously, combining the various cryptographic techniques requires us to prove the security properties as defined in our model.

1.3 Other related works

Several open-source projects [8, 9, 34] focus on improving confidentiality by using standard CPA secure encryption to do file-wise encryption. So the storage cost is linear to the product of version numbers and the size of changed files in a repository. For each update operation, a user needs to re-run encryption on the changed files of the repository. So the encryption time and communication size are related to the size of all changed files. Moreover, Keybase Git [9] is designed to work with Keybase server and is not compatible with existing Git Servers such as GitHub. To save the storage cost, Git-crypt [4] sacrifices the CPA confidentiality by using the hash value of the file as the initialization vector of AES encryption. For those unchanged files, the hash values keep unchanged, so as to the ciphertexts of the files. But any tiny change of the file can produce a totally different ciphertext from the previous version. So the storage saving method does not apply to minor changes spreading most files of the repository. Git-remote-gcrypt [8] applies delta compression on the entire new plaintext version of the repository to generate a packfile before encryption. Thus, in the Git server, each update will add a new encrypted packfile, compressing all files (including objects for the new version) together. Since the packfile is encrypted, the Git server cannot interpret it to parse a new version, and all versions are treated as a single version. As a result, some Git operations, such as fetching a specified version or object from the Git server, and merging two versions on the Git Server are not supported in Git-remote-gcrypt.

Gingotts [37] uses another deterministic encryption to save storage. They fix the IV of AES and do line-wise encryption, so that the data compression can be done cross files and a tiny change within a line only brings a new line of ciphertext, which saves the storage cost more than doing file-wise encryption. [37, 38] further enhances the access control of VCS via attribute-based encryption. Gringotts [37] considered a weaker model in terms of unforgeability, where the remote server is assumed to be honest. Disc [38] applies attribute-based signature to force write access control without formal model analysis. [13] studied the auditable integrity of VCS to ensure that each version of the repository is retrievable in the malicious server setting.

2 SYNTAX

We abstract seven core operations for E2E encrypted Git services below, and each operation is formalized as an interactive protocol between the user and server (e.g., Git server).
– *Registration.* Users register to the Git server.
– *Authentication.* Users authenticate to the Git server to open an active session, within which users can interact with the Git server to do the following repository operations.
– *Initialization.* Users set up the Git repository structure locally and remotely, which is mapped to `git init` command of Git and initialize the first version of the repository with tracking files.
– *Update.* Users update the repository with new files or new versions of existing files, which are mapped to a series of Git commands `git add, git commit, git push`.
– *Pull.* Users fetch the local repository's missing part from the server to sync with the server's repository, which is mapped to Git commands `git pull, git fetch`.
– *Share.* Users share the repository with others. There are two sub-protocols, denoted as $share_I$ and $share_{II}$, where the sender interacts with the Git server to request, and the receiver interacts with the Git server to accept, respectively.

Syntax. Formally, an end-to-end encrypted Git service is composed of a tuple of interactive protocols SGit:= $(\Pi_{reg}, \Pi_{auth}, \Pi_{init}, \Pi_{update}, \Pi_{pull}, \Pi_{share_I}, \Pi_{share_{II}})$ outlined above, where each is run by a user $\mathcal{U}$ and a server $\mathcal{S}$ via a subroutine, e.g., $\Pi_{reg} = \langle \mathcal{U}_{reg}, \mathcal{S}_{reg} \rangle$. Users and the server maintain their states st_U, st_S, respectively.

In the following, we will describe the protocols with compulsory inputs and outputs and omit other optional ones. In each protocol, each party has a bit of implicit output indicating the execution state, where one indicates it succeeds, otherwise fails.

– $\Pi_{reg}\langle uid; st_S \rangle \rightarrow \langle (cred, km); (st'_S) \rangle$: the registration protocol creates a new user, where the user takes the unique user ID uid as input and gets the authentication credential $cred$ and key materials km as output. Server updates state st_S with the new user record.

A user record in st_S includes all data related to the user uid. After registration, it has at least two attributes: uid and necessary material for verifying user authentication, e.g., $cred$ or the corresponding public key if $cred$ is a private key. Π_{reg} must be run once on behalf of that user before any other protocols can be run. It does not involve any persistent state of the user yet (i.e., the user state is empty $st_U.s = \epsilon$).

– $\Pi_{auth}\langle (uid, cred); st_S \rangle \rightarrow \langle (st_U); (st'_S) \rangle$: the authentication protocol authenticates a user to the server and initiates a new active user session. The user takes $uid, cred$ as input. After passing the authentication, the two parties update their states with the new session state.

This user session state $st_U.s$ is shared among all following protocols run within this session. A user can initiate multiple user sessions in parallel (each holding their own state $st_U.s$), which can concurrently access the user's repositories in the Git server. [5]

– $\Pi_{init}\langle(st_U, km, rid, \mathbf{f}^{pt}); st_S\rangle \rightarrow \langle(repo); (st'_S)\rangle$: the initialization protocol runs within an active session and initiates a new repository locally and remotely. A user takes as input st_U, km, a globally unique identifier rid, and plain tracking files $\mathbf{f}^{pt}$ including file path, name, and contents. The user outputs Git repository $repo$, including a ciphertext repository $repo^{ct}$. The server updates st_S by adding $(rid, repo^{ct})$ to the user's record.

– $\Pi_{update}\langle(st_U, km, rid, repo_{old}, \mathbf{f}^{pt}_{new}); st_S\rangle \rightarrow \langle(repo_{new}); (st'_S)\rangle$: the update protocol runs within an active session, and updates the contents locally and remotely. $repo_{old}$ denotes the latest committed repositories locally and $\mathbf{f}^{pt}_{new}$ is the new plain files to be updated. The user's output is the updated repositories $repo_{new}$, including the updated ciphertext repository $repo^{ct}_{new}$. The server updates st_S with an updated user record $(uid, rid, repo^{ct}_{new})$.

– $\Pi_{pull}\langle(st_U, km, rid, repo_{old}); st_S\rangle \rightarrow \langle(repo_{new}); (st_S)\rangle$: the pull protocol runs within an active session, fetch the missing part from the remote repository, and get the plain contents. The user outputs the new repository $repo_{new} = (repo^{pt}_{new}, repo^{ct}_{new})$, where $repo^{ct}_{new}$ is the latest ciphertext repository in st_S's user record with (uid, rid).

– $\Pi_{share_I}\langle(st_U, km, rid, acs, repo_{old}, uid_{re}); st_S\rangle \rightarrow \langle(repo_{new}, oob); (st'_S)\rangle$: the repository sharing protocol runs within an active session and enables users to share the access defined in acs of the repository rid with the receiver uid_{re}. The protocol updates the repository to a new version with a new access list. We consider two types of access: read-only and write access, and only the repository owner can share it with others. The user outputs the new repository $repo_{new}$ and the out-of-band message oob, which can be communicated via the out-of-band secure channel. The server states that st'_S has one more pending record for managing the receiver's access acs and gets updated with new ciphertext repository $repo^{ct}_{new}$.

– $\Pi_{share_{II}}\langle(st_U, rid, oob); st_S\rangle \rightarrow \langle(st'_U); (st'_S)\rangle$: the repository accepting protocol runs within an active session and enables the receiver to accept the repository sharing. The server removes the pending access of uid and adds it to the access list of the repository rid. As such, the user uid can access it when logged in. *Remark on revocation.* In this paper, we do not consider access revocation. Once a user gets access to a repository, it lasts until the repository gets deleted. To enable access revocation, one possible method is to revoke the user's access to future versions of the repository. It can be done by changing the repository encryption key for future versions, not sharing the encryption key with revoked users, and removing the revoked user's signing key from the repository. However, revoking access to previous versions of the repository is more challenging. A trivial method is to delete the whole repository and re-initialize it from scratch, which is inefficient and results in the loss of all history. Better methods for revocation is an interesting open problem.[6]

[5] All remaining protocols operating on repositories can only be called with non-empty user session state $st_U.s$ (i.e., we implicitly require them to fail otherwise). Overall, this enforces that user registration must be run before authentication, and authentication must be run before any repository operation protocol.

[6] And in general, there are many interesting questions to be studied, including systematic treatment on post-compromise security, e.g., via updatable encryption, e.g., [14, 23], better cryptographic group management [7], full metadata protection [16], accountability, enabling AI assistance while maintaining E2E security, and many more.

Notations for modification. We follow [11] to define document modifications. A document D is denoted as a sequence of blocks $m_1, \ldots, m_n$, where the block size may depend on the security parameter k, and n denotes an integer since a document can always be padded using standard padding methods if the original size is not a multiple of the block size. We use $O = (op, idx, m)$ to denote a generic modification operation, where op is the operation type, idx is the operation position, m is the new message, and $|O.m|$ is the message length. $O(D)$ denotes the effect of O on document D. So, a sequential modification operations $\{O\}_n$ on document D can be denoted as $O_n(\ldots(O_3(O_2(O_1(D)))))$. In this paper, we consider the basic two operation types $O.op \in \{delete, insert\}$ that essentially can capture all modifications on the document, including replace, copy-and-paste, cut-and-paste, etc.

$O = (insert, i, m_i)$ insert m_i as the i-th block of the document.
$O = (delete, i)$ deletes the i-th data block.

Data update. In the update protocol SGit.Π_{update}, the modification operations between the two versions f, f' of each tracked file are calculated. We use ComDiff algorithm that takes f, f' as input and outputs a sequential set of modification operations $\{O\}_n$ in the form we defined before. We do not specify the specific construction for ComDiff algorithm. The correctness of ComDiff requires that $f' = O_n(O_{n-1}(\ldots(O_1(f))))$ where $\{O\}_n \rightarrow \mathsf{ComDiff}(f, f')$.

Correctness. When the Git server and all users who have access to the repository act honestly, users can always pull the repository with the same contents as the last push.

Correctness captures that: (1) an honest user registered to the service can authenticate with the same user ID and credentials used during registration; (2) a repository initialized, updated, or shared by an honest user can be retrieved with its original contents.

$$\begin{aligned}
&\Pr[\Pi_{auth}(uid, cred;) = (1;1) | \Pi_{reg}(uid;) = (1, cred;1)] = 1\\
\wedge &\Pr[\Pi_{pull}(st_U, rid;) = (repo;) | \Pi_{init}(st_U, rid, repo;) = (1;1)] = 1\\
\wedge &\Pr[\Pi_{pull}(st_U, rid;) = (repo';) | \Pi_{update}(st_U, rid, repo';) = (1;1)] = 1\\
\wedge &\Pr[\Pi_{pull}(st_{U_{uidre}}, rid;) = (repo;) |\\
&\Pi_{share_I}(st_U, rid, uid_{re};) = (1;1) \wedge \Pi_{pull}(st_U, rid;) = (repo;)] = 1
\end{aligned}$$

3 SECURITY MODELS

We will formally define security properties for an E2E encrypted Git service. Intuitively, in a plain Git service, if one fully trusts the Git server, the server can enforce access control policies. End-to-end secure Git services try to "emulate" this ideal setting via algorithm/protocol design. Naturally, it has to satisfy the security requirements of confidentiality (w.r.t the *read* access) and integrity (we consider a stronger version, called unforgeability, that is w.r.t the *write* access). However, modeling these properties becomes significantly more complex in practice due to the functionality of data *updates* and the multi-user *sharing* setting; these allow adversaries to interact with honest users in a dynamic and complex manner.

Setup assumption: First, we will assume a plain PKI model; that is, users can know others' public keys via an out-of-band channel. This can be achieved via the PGP mechanism in practice.

Data confidentiality, captures not only the content in the repository but also the update details, even against a corrupted Git server, which can interact with honest users. The subtle point is that Git services allow version updates. The ciphertexts of multiple versions can give more power to adversaries than purely exposing the ciphertext of a single version (which is the case in the standard confidentiality model). More specifically, the ciphertexts of multiple versions may not all be generated directly from their plaintexts, and the ciphertext of a later version might be generated based on the ciphertext of its former version. This complicates the modeling and analysis: In the conventional confidentiality model (with CPA flavor), adversaries can obtain a ciphertext by querying a chosen plaintext. But now, adversaries are allowed to additionally obtain ciphertexts by "updating" with a previous ciphertext and a new chosen plaintext. Our confidentiality is already stronger than CCA security.

We also give a slightly weaker version of confidentiality by allowing adversaries to learn the update locations.

Repository unforgeability, tries to capture verifiable write access, which is necessary for Git services of version control among multiple users. Unforgeability guarantees that even if a Git server gets corrupted, each user can only edit on their own behalf and cannot forge other users' edits or frame other honest users. We thus consider that attackers have strong capabilities and could corrupt the server and legitimate users who have read and/or write permission to the target repository (malicious insiders), pretend to be other honest users, and try to forge a new version of the repository on behalf of honest users. For example, a user who has read-only access to the repository may get compromised. In this case, attackers can pull the contents of the repository but should not be able to break the access restriction (e.g., push) or pretend to any honest user to write even if the attacker corrupts some other users with write access.

Interestingly, by simply restricting adversaries to corrupt only the server and users who do not have read permission to the target repository (can be viewed as external attackers and slightly adapt security games), we can easily get a weaker notion of repository unforgeability called *repository integrity*. This notion may also be useful to ensure repository integrity so that an honest repository will remain complete (no file deletion/insertion without being noticed). [7]

3.1 Modeling preparations: oracles & states

To prepare for the security modeling, we first introduce eight oracles $O = \{O_{reg}, O_{auth}, O_{init}, O_{pull}, O_{upd}, O_{share_I}, O_{share_{II}}, O_{corrupt}\}$ to capture the adversary's capability. Since each protocol in SGit is run between the user and the Git server, which is corrupted in all security models. Oracles mainly run user-side algorithms and provide interfaces for adversaries to interact with honest users, except that $O_{corrupt}$ is provided for adversaries to corrupt honest users.

Our models consider the single server setting and selective user corruption. So in each security game, adversary $\mathcal{A}$ first specifies the list of corrupted users $U_{corrupt}$, which means later $\mathcal{A}$ can only query $O_{corrupt}$ with user id $uid \in U_{corrupt}$.

In our security games, $\mathcal{A}$ has the same access to all oracles in O except different restrictions on the user corruption oracle $O_{corrupt}$. In the data confidentiality and weak repository unforgeability model, adversaries are not allowed to corrupt users who have legitimate access to the challenge/target repository since the two models only capture security against outsiders of the challenge/target repository. In the repository unforgeability model, adversaries are allowed to query $O_{corrupt}$ to corrupt insiders with read or write access to the target repository as long as the target user is not corrupted.

We define several global states maintained by the challenger and oracles for security games.

U: a set of uid recording registered users.
C: a credential dictionary mapping user id uid to authentication credential $cred$.
K: a key material dictionary mapping user id uid to a tuple of key materials $km = (mk, sk_e, pk_e, sk_s, pk_s)$.
R: a set of rid recording existing repositories.
S: a user session state dictionary mapping a tuple of user id uid and session id sid to the session state st.
RP: a dictionary mapping repository id rid to its latest local repository $repo$.
O: a dictionary mapping repository id rid to its owner id uid.
$A[rid]$: the set of accessible users uid for the repository id rid.
$W[rid]$: the set of users uid with write access to the repository rid.
rid^*: the challenge repository identifier.
fid^*: the challenge file id.
f_b^*: the two challenge related plain files for $b \in \{0,1\}$.
$repo^*$: the challenge repository.

The formal oracle description is shown in Figure 2. For clarity, each oracle has an implicit output indicating the procedure succeeds or fails and is specified for other output. The details of oracle description are described as follows:
– O_{reg} allows $\mathcal{A}$ to initiate a user registration with user id uid. The generated credential and key materials are hidden from $\mathcal{A}$ and can be corrupted via $O_{corrupt}$. Each uid is globally unique and can only be registered once with a successful record.
– O_{auth} allows $\mathcal{A}$ to initiate the user authentication with given uid. A successful authentication starts a new session with id sid and persistent state $S[uid, sid]$.
– O_{init} allows $\mathcal{A}$ to initialize the repository with repository id rid, the plain files $\mathbf{f}^{pt}$ including each file path $fid \in \mathbf{f}^{pt}.Fid$ and corresponding contents $\mathbf{f}^{pt}[fid]$.
– O_{pull} retrieves the latest repository rid in the specified active session sid on behalf of user uid. It returns the specified repository $repo_{new}$. To avoid $\mathcal{A}$'s trivial win of the confidentiality game, the retrieval of the challenge repository only returns plaintext versions of non-challenge files.
– O_{upd} updates the repository rid with new files $\mathbf{f}^{pt}$ on behalf of user uid in session sid. For update queries on the challenge repository, further checks are needed to avoid $\mathcal{A}$'s trivial wins via differences of update operation and content length or update position to get an advantage in the confidentiality game.

[7] There is an intermediate security notion between repository integrity and unforgeability, which captures integrity against malicious read-only insiders, ensuring that read-only users cannot modify the repository. We leave the systematic exploration of this case, which supports more fine-grained access control, for future work.

```
$O_{reg}(uid)$
if $uid \in U$   return "Registered!"
else $\langle \mathcal{U}_{reg}(uid), \mathcal{A}\rangle \to (cred, km)$   / only show user-side output
    add $uid \to U, km \to K[uid], cred \to C[uid]$
    return $(uid, pk_e, pk_s)$.   / $km$ includes $(mk, sk_e, pk_e, sk_s, pk_s)$

$O_{auth}(uid)$
if $uid \notin U$, return $\perp$   / only for registered users via $O_{reg}$
else $cred \leftarrow C[uid]$, $sid \leftarrow\!\!\$$, $\langle \mathcal{U}_{auth}(uid, cred), \mathcal{A}\rangle \to st$
    set $st \to S[uid, sid]$, return $sid$

$O_{init}(uid, sid, rid, \mathbf{f}^{pt})$
if $S[uid, sid] = \epsilon \vee rid \in R$   return $\perp$   / only for active sessions of $O_{auth}$
else $\langle \mathcal{U}_{init}(S[uid, sid], K[uid], rid, \mathbf{f}^{pt}), \mathcal{A}\rangle \to repo_{new}$
    add $rid \to R$, $uid \to A[rid]$, $uid \to W[rid]$
    set $repo_{new} \to RP[rid]$, $uid \to O[rid]$

$O_{upd}(uid, sid, rid, \mathbf{f}^{pt}_{new})$
if $S[uid, sid] = \epsilon \vee uid \notin W[rid]$   return $\perp$
else parse $RP[rid] = repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$
    if $rid = rid^* \wedge fid^* \in \mathbf{f}^{pt}_{new}.Fid$ then set $f \leftarrow \mathbf{f}^{pt}_{new}[fid^*]$
      run $O_0 \leftarrow \mathsf{ComDiff}(f^*_0, f), O_1 \leftarrow \mathsf{ComDiff}(f^*_1, f)$
      require $O_0.op = O_1.op \wedge |O_0.m| = |O_1.m|$
      / require same type and length of modification for challenges $f^*_0, f^*_1$
      $\boxed{\text{require } O_0.idx = O_1.idx}$   / require modification on same position
    run $\langle \mathcal{U}_{upd}(S[uid, sid], rid, repo_{old}, \mathbf{f}^{pt}_{new}), \mathcal{A}\rangle \to repo_{new}$,
    set $repo_{new} \to RP[rid]$, if $rid = rid^*$, set $RP[rid] \to repo^*$

$O_{pull}(uid, sid, rid, v)$
if $S[uid, sid] \to st = \epsilon$ return $\perp$
else $\langle \mathcal{U}_{pull}(st, rid, RP[rid]), \mathcal{A}\rangle \to repo_{new} = (repo^{pt}_{new}, repo^{ct}_{new})$
    set $repo_{new} \to RP[rid]$
    if $rid \notin R$ then add $rid \to R$, $repo^{ct}_{new}.owner \to O[rid]$
    for each $f_{acs} \in repo^{ct}_{new}$
      set $A[rid] \cup f_{acs}.R \to A[rid], W[rid] \cup f_{acs}.W \to W[rid]$
    if $rid \neq rid^*$ return $repo_{new}$
    else set $repo_{new} \to repo^*$
      return $\mathbf{f} \in repo^{pt}_{new}$, where $fid^* \notin \mathbf{f}.Fid$   / no challenge file return

$O_{share_I}(uid, sid, rid, uid_{re}, acs)$
if $S[uid, sid] \to st = \epsilon \vee uid \notin O[rid]$ return $\perp$
else $\langle \mathcal{U}_{shr}(st, rid, uid_{re}, K[uid], RP[rid]), \mathcal{A}\rangle \to (repo_{new}, oob)$
    set $repo_{new} \to RP[rid]$, update $A[rid], W[rid]$per $acs$
    return $oob$

$O_{share_{II}}(uid, sid, rid, oob)$
if $S[uid, sid] = \epsilon$ return $\perp$
else $\langle \mathcal{U}_{acp}(S[uid, sid], rid, oob), \mathcal{A}\rangle$

$O_{corrupt}(uid)$
if $uid \notin U \vee uid \notin U_{corrupt}$ return $\perp$ else return $(C[uid], K[uid])$
```

Figure 2: Oracles O. $\boxed{\textit{Boxed}}$ is for weaker confidentiality.

– O_{share_I} initiates the repository sharing. $\mathcal{A}$ specifies uid, sid, rid, and the receiver id uid_{re}. It returns the out-of-band message oob.
– $O_{share_{II}}$ initiates the receiver acceptance process of sharing within an active session. $\mathcal{A}$ specifies uid, sid, rid.
– $O_{corrupt}$ allows $\mathcal{A}$ to corrupt honest users and returns their secrets.

3.2 Data confidentiality

Data confidentiality captures both the file content confidentiality and *update* confidentiality against outsiders (who do not have legitimate access to the target repository), including the malicious server, except the length of the initial file and update metadata.

The confidentiality game is defined in Figure 3. In the game, $\mathcal{A}$ has access to all eight oracles in O. In the challenge submission phase, $\mathcal{A}$ submits a targeted registered honest user uid^* and his repository identified by rid^*, and specifies the file id fid^* and two challenge files f^*_0, f^*_1. The challenger C randomly selects one file f^*_b to initiate or update the repository via interacting with $\mathcal{A}$. $\mathcal{A}$'s goal is to distinguish which file (the bit b) was chosen. To avoid trivial wins, $\mathcal{A}$ is not allowed to corrupt any user who has legitimate access to the challenged repository.

```
Data Confidentiality Game $G^{\mathsf{CONF}}_{\mathsf{SGit},\mathcal{A},q}$
$b \leftarrow\!\!\$ \{0, 1\}$
Global $U, C, K, R, S, RP, O, \mathbf{A}, \mathbf{W}, rid^*, fid^*, f^*_0, f^*_1, repo^*$
$U_{corrupt} \leftarrow \mathcal{A}$   / specify user corruption
$(uid^*, sid^*, rid^*, fid^*, f^*_0, f^*_1) \leftarrow \mathcal{A}^{O}$   / submit challenge after queries
update global state $(rid^*, fid^*, f^*_0, f^*_1)$
if $uid^* \notin U \vee S[uid^*, sid^*] \to st = \epsilon \vee (rid^* \in R \wedge uid^* \notin W[rid^*])$
    return $\perp$   / exclude invalid challenge
if $uid^* \in U_{corrupt} \vee U_{corrupt} \cap A[rid^*] \neq \emptyset$
    return $\perp$   / exclude insider corruption
if $rid^* \notin R$ then set $f_{ori} = \emptyset$
else parse $repo_{old} \leftarrow RP[rid^*]$to get $\mathbf{f}^{pt}_l$   / $\mathbf{f}^{pt}_l$ is the latest version of plain files
      set $f_{ori} \leftarrow \mathbf{f}^{pt}_l[fid^*]$
$O_0 \leftarrow \mathsf{ComDiff}(f_{ori}, f^*_0), O_1 \leftarrow \mathsf{ComDiff}(f_{ori}, f^*_1)$
if $|O_0| \neq |O_1| \vee O_0.op \neq O_1.op$ return $\perp$
  / exclude trivial win with different update operation or length
$\boxed{\text{if } O_0.idx \neq O_1.idx \text{ return } \perp}$   / exclude trivial win via different update locations
if $rid^* \notin R$ : $\langle \mathcal{U}_{init}(st, rid^*, K[uid^*], fid^*, f^*_b), \mathcal{A}\rangle \to repo_{new}$
else   $\langle \mathcal{U}_{upd}(st, K[uid^*], rid^*, repo_{old}, fid^*, f^*_b), \mathcal{A}\rangle \to repo_{new}$
  / challenge enc or update
set $repo_{new} \to repo^*$, $repo_{new} \to RP[rid^*]$
$b' \leftarrow \mathcal{A}^{O}$   / $\mathcal{A}$ guess after challenge and queries
if $U_{corrupt} \cap A[rid^*] \neq \emptyset$, return $(b = b')$;   else return $\perp$
```

Figure 3: Data confidentiality game. $\boxed{\textit{boxed}}$ is for weaker confidentiality $G^{\mathsf{CONF_w}}_{\mathsf{SGit},\mathcal{A},q}$, in which the update position is required to be the same.

Definition 1 (Data Confidentiality). *Let SGit be a Git service, and $G^{\mathsf{CONF}}_{\mathsf{SGit},\mathcal{A},q}$ be the data confidentiality game defined in Figure 3* *with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of $\mathcal{A}$ playing this game as*

$$\mathrm{Adv}^{\mathsf{CONF}}_{SGit,\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{CONF}}_{SGit,\mathcal{A},q} = 1] - 1/2.$$

Remark. We define a weaker version of data confidentiality, called weak data confidentiality, as follows. It is formalized via the game $G^{\mathsf{CONF}_w}_{\mathsf{SGit},\mathcal{A},q}$, in which the attacker is allowed to learn the update positions.

Definition 2 (Weak Data Confidentiality). *Let SGit be a Git service, and $G^{\mathsf{CONF_w}}_{SGit,\mathcal{A},q}$ be the weak data confidentiality game defined in Figure 3 including additional boxed restriction, with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of the adversary playing this game as*

$$\mathrm{Adv}^{\mathsf{CONF_w}}_{SGit,\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{CONF_w}}_{SGit,\mathcal{A},q} = 1] - 1/2.$$

3.3 Repository unforgeability

Repository unforgeability captures that an adversary cannot forge a new version of a valid ciphertext repository on behalf of honest users, even if the adversary has the capability to corrupt users with write access to the repository (weak repository unforgeability restricts the adversary's capability to access the repository). Moreover, this unforgeability inherently inherently enforces both edit-source authenticity and read-write access separation. Edit-source authenticity ensures that users cannot impersonate others when editing the repository, preventing any user from writing a new version on behalf of another without detection. Read-write access separation guarantees that read-only users are cryptographically prevented from performing write operations. A weaker form of unforgeability still ensures write access control against attackers with no access privileges.

The unforgeability game is defined in Figure 4, where $\mathcal{A}$ has access to all eight oracles in O. To capture the security against malicious insiders, $\mathcal{A}$ is allowed to corrupt users who have legitimate access to the challenge repository except for the challenged honest user, which may cause a trivial win. In this game, $\mathcal{A}$'s goal is to impersonate an honest user by forging a new version of the repository on behalf of the target honest user. The weaker unforgeability game imposes an additional restriction, boxed in Figure 4, which prohibits the adversary $\mathcal{A}$ from corrupting any user who has access (read or write) to the target repository.

Definition 3 (Repository Unforgeability). *Let SGit be a Git service, and $G^{\mathsf{UNF}}_{SGit,\mathcal{A},q}$ be the repository unforgeability game defined in Figure 4 with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of an adversary playing this game as*

$$\mathrm{Adv}^{\mathsf{UNF}}_{SGit,\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{UNF}}_{SGit,\mathcal{A},q} = 1].$$

Repository integrity. We define repository integrity (also called weak repository unforgeability) against malicious repository outsiders, including the malicious server, except users who have legitimate access to the repository. It captures that attackers who have no access to the target repository cannot forge a new version of the target repository, even given many versions of the repository. This guarantees that even a malicious server cannot cheat users with an incomplete version of the target repository where partial files are deleted or lost.

```
Repository Unforgeability Game $G^{\mathsf{UNF}}_{\mathsf{SGit},\mathcal{A},q}$
Global $U, C, K, R, S, RP, O, \mathbf{A}, \mathbf{W}$
$U_{corrupt} \leftarrow \mathcal{A}$    / specify user corruption
$(uid^*, sid^*, rid^*, repo^*_{ct}) \leftarrow \mathcal{A}^{O}$    / submit challenge after queries
if $repo^*_{ct} \subseteq RP[rid^*] \lor uid^* \in U_{corrupt} \lor uid^* \notin U \lor$
$S[uid^*, sid^*] \to st = \epsilon$ return $\perp$
    / exclude trivial win and invalid challenge
$\langle \mathcal{U}_{pull}(st, rid^*, K[uid^*], RP[rid^*]), \mathcal{A} \rangle \to (repo^*;)$
require $repo^* = (repo^*_{pt}, repo^*_{ct}) \neq \perp$    / check valid repo
$\boxed{\textbf{require } A[rid^*] \cap U_{corrupt} = \emptyset}$    / exclude trivial win via user corruption
if $\exists\, repo^*_v, s.t., repo^*_v \in repo^*_{ct} \land repo^*_v \notin RP[rid^*] \land f_{tag}.au = uid^*$
    / $repo^*_v = (\mathbf{f}^{ct}, f_{acs}, f_{tag})$ is a version of repository, where $f_{tag}.au$ is the author
    return 1    / $\mathcal{A}$ win if exist one untracking version edited by honest user
else return 0
```

Figure 4: The repository unforgeability game. $\boxed{\text{boxed}}$ is for weaker unforgeability $G^{\mathsf{UNF_w}}_{\mathsf{SGit},\mathcal{A},q}$ (called repository integrity), in which corrupted users do not have any legitimate access to the target repository.

The Integrity game is shown in Figure 4 with additional restrictions in the box to the user corruption. During the game, $\mathcal{A}$ has access to the eight oracles, and the goal is to provide a new version of the ciphertext repository, which is valid but is different from all existing versions. The trivial win is that $\mathcal{A}$ corrupts a user who has legitimate access to the target repository.

Definition 4 (Repository Integrity). *Let SGit be a Git service, and $G^{\mathsf{INT}}_{SGit,\mathcal{A},q}$ be the repository integrity game, which is also weak repository unforgeability game $G^{\mathsf{UNF_w}}_{SGit,\mathcal{A},q}$ shown in Figure4 including the boxed restriction with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of the adversary playing this game as*

$$\mathrm{Adv}^{\mathsf{INT}}_{SGit,\mathcal{A},q}(\mathcal{A}) = \mathrm{Adv}^{\mathsf{UNF_w}}_{SGit,\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{UNF_w}}_{SGit,\mathcal{A},q} = 1].$$

Repository unforgeability and integrity. We defined repository unforgeability and integrity (the weaker version of unforgeability) to capture different attackers. In both security modelings, adversaries share the same goal of forging a valid ciphertext. But they have different capabilities. The integrity adversary can be seen as an outside attacker who has no access to the repository. However, the unforgeability adversary acts as an inside attacker who has legitimate access to the repository, including read and write access. It is easy to conclude that the unforgeability against insiders is stronger than the integrity against outsiders. For this reason, we will only prove the unforgeability against insiders for our constructions.

Further modeling discussion: *strengthening integrity and unforgeability.* In this paper, we consider integrity and unforgeability where malicious attackers can not forge a different one from existing versions of the repository. The malicious server may delete or lose an entire version of a repository. A stronger security notion captures that it can be caught if the malicious server cannot provide an old version. A promising solution could be to use a hash chain to link the previous versions with the next version and sign it so that, as long as the hash chain is signed, malicious attackers cannot forge one from an internal point. *Remarks.* Regarding defending denial-of-service (DoS) attacks, we always assume the server is semi-honest, which blocks illegitimate users and provides available service to legitimate users. Also, besides the data confidentiality, there could also be metadata privacy protecting the file name, file directory, etc, which we leave for further study.

4 PROVABLY SECURE CONSTRUCTIONS

We propose two constructions of E2E encrypted Git services, SGitChar and SGitLine, which are fully compatible with existing Git servers, including GitHub, and formally analyze their security. In this section, we first take SGitChar as an example to show the main workflow. Then we introduce our two constructions: SGitLine which we briefly describe and achives weak data confidentiality, and SGitChar which we describe in detail and which satisfies both data confidentiality and repository unforgeability.

Overview of E2E encrypted Git workflow. In E2E encrypted Git services, both a client (a user's device) and the Git server maintain a repository but in the form of ciphertext. To enable the user to efficiently read and edit the repository, the user's device maintains one more repository with the corresponding plaintext. As in conventional Git services, users register at the very beginning and authenticate to the Git server before doing repository-related operations. After authentication, the initialization protocol enables the user to create a new repository and synchronize the first version with the Git server. Later, users can update the repository and synchronize with the Git server, share the repository with other users, and pull new versions from the Git server to synchronize the local repository.

Concretely, (1) for secure initialization, a user initiates two repositories for plain data and ciphertext first, configures the remote ciphertext repository, and connects it with the local ciphertext one. Then, the user takes the first version of the plaintext repository as input, generates a ciphertext version by applying encryption on each file, commits it to the local ciphertext repository with a signature, and pushes it to the remote ciphertext repository in the Git server to finish the initialization. (2) Regarding the secure update, the user has the previous and new plaintext repository and the previous ciphertext repository, forms a new version of ciphertext repository locally, and pushes it to the Git server. The methods to form a new version of the ciphertext repository are different for the two constructions. (3) Regarding secure pull, the user pulls local absent versions from the Git server. The workflow is that the user first pulls the remote ciphertext version to the local ciphertext repository and then verifies and recovers the plaintext version of the repository. The ways of recovering the plaintext version correspond to the methods to form the ciphertext version, which are different in the two schemes. (4) To share a repository with others, the sender needs to send a request to the Git server to give access permission to the receiver. The sender also needs to update the ciphertext repository with a new access control file, which includes key material encrypted under the receiver's public key and the sender's authorization via a signature. The receiver needs to accept the access via interacting with the Git server.

We assume each user has two key pairs, for digital signature and public key encryption, which are bound to the user's identity. The distribution of public keys is via an out-of-band channel so that each user knows other users' public keys. Each user has a small, constant size of secure storage, e.g., hundreds of bits, for keeping secrets locally. To be compatible with the most widely deployed user authentication, we support users in authenticationg to the server usingwith general credentials such as a passwords and a tokens. So, each users may need to remember a passwords and keep all private secrets locally.

A diagram describing the main workflow of the pullupdate and updatepull procedure with SGitChar as the specific construction is shown in Figure 5.

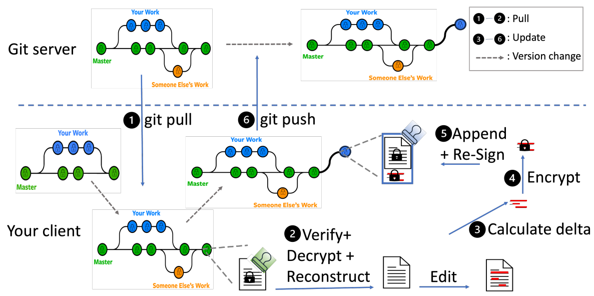


Figure 5: The main workflow of SGitChar

Preparation of construction. An important component of Git we will leverage for efficient construction is the Diff computation functionality. $\mathsf{ComDiff}(repo, repo') \to \delta$: The ComDiff algorithm takes two versions of a repository as input, and generates the difference δ. The difference δ makes sure that $repo'$ can be reconstructed from $repo$ and δ. The reverse reconstruction is not compulsory but could be useful for optimizing reconstruction efficiency. With different implementations of ComDiff, the size of difference δ is also different. One direction of optimizing the storage cost is to make δ as small as possible. We have two schemes $\mathsf{ComDiff}_{char}$ and $\mathsf{ComDiff}_{line}$ with different granularities to compute the difference. $\mathsf{ComDiff}_{char}$ compares difference between two characters and $\mathsf{ComDiff}_{line}$ for two lines. For the two repositories, the size of $\mathsf{ComDiff}_{char}$ should be no larger than $\mathsf{ComDiff}_{line}$.

4.1 Construction: SGitLine

A secure construction should get rid of deterministic encryption that leaks data patterns and only provides a weak security guarantee. To reduce cost, the encryption is not trivially applied on all files so that the computation and storage cost is not linear to the product of the version number and file size. One of our goals is to balance confidentiality and efficiency. The other is to achieve desired integrity and unforgeability while keeping confidentiality and efficiency.

After a careful investigation of version control systems, we observe that they have a more fine-grained data partition and location method, so that they can reduce storage and communication for

```
$< \mathcal{U}_{Init}(st_U, km, rid, \mathbf{f}^{pt}), \mathcal{S}_{Init}(st_S) >$
$\mathcal{U}$: $req : st_U.\{uid, sid\}$.  $k \leftarrow \mathsf{KDF}(mk, rid)$
$\mathcal{U}$: for $f_i \in \mathbf{f}^{pt}, i \in [1, n]$        / $n$ is # of content files in $\mathbf{f}^{pt}$
$\mathcal{U}$:     $\boxed{\text{for } j \in [1, f_i.l]}$        / $f_i.l$ is # of lines in $f_i$
$\mathcal{U}$:         $\boxed{lct_j \leftarrow Enc(k, l_j)}$        / encrypt by line
$\mathcal{U}$:     $\boxed{ct_i \leftarrow (lct_1, \ldots, lct_{f_i.l})}$     $\boxed{ct_i \leftarrow Enc(k, f_i)}$
$\mathcal{U}$: $\mathbf{f}^{ct} \leftarrow (ct_1, \ldots, ct_n)$
$\mathcal{U}$: $rh \leftarrow \mathtt{MerkleDAG}(\mathbf{f}^{ct})$ / each $ct_i$ is a leaf node, $i \in [1, n]$
$\mathcal{U}$: $h \leftarrow Hash(rid \| uid \| rh), \sigma \leftarrow \mathsf{Sign}(sk_s, h)$
$\mathcal{U}$: add commit message $f_{tag} \leftarrow (uid, \sigma)$
$\mathcal{U}$: $repo^{ct}_{new} \leftarrow (\mathbf{f}^{ct}, f_{acs} = \emptyset, f_{tag})$
$\mathcal{U}$: Send $uid, sid, rid, repo^{ct}_{new}$ to $\mathcal{S}$
$\mathcal{U}$: add $repo^{pt}_{v1} = (\mathbf{f}^{pt}, f^{pt}_{acs} = \emptyset) \rightarrow repo^{pt}_{new}$
$\mathcal{S}$: $req : st_S.\{usr, ses, Rid\}$
$\mathcal{S}$: if $ses[sid] \neq uid$ or $rid \in Rid, Fail$
$\mathcal{S}$: Add $repo^{ct} \leftarrow repo^{ct}_{new}$
```

```
$< \mathcal{U}_{update}(st_U, km, rid, repo_{old}, \mathbf{f}^{pt}_{new}), \mathcal{S}_{update}(st_S) >$
$\mathcal{U}$: $req : st_U.\{uid, sid\}$
$\mathcal{U}$: $repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$ / last committed local repositories
$\mathcal{U}$: parse $repo^{pt}_{vl} = (\mathbf{f}^{pt}_{vl}, f^{pt}_{acs}) \in repo^{pt}_{old}$
$\mathcal{U}$: parse $repo^{ct}_{vl} = (\mathbf{f}^{ct}_{vl}, f_{acs}, f_{tag}) \in repo^{ct}_{old}$ and $rid = uid_o \| nonce$
$\mathcal{U}$: if $uid = uid_o$    $k \leftarrow KDF(mk, rid)$
$\mathcal{U}$: else $c_k \leftarrow f_{acs}.R[uid]$, $k \leftarrow \mathsf{PKE.Dec}(sk^{uid}_e, c_k)$
$\mathcal{U}$: for $fid \in \mathbf{f}^{pt}_{vl}.Fid \cap \mathbf{f}^{pt}_{new}.Fid$
$\mathcal{U}$:     $f \leftarrow \mathbf{f}^{pt}_{vl}[fid]$, $f' \leftarrow \mathbf{f}^{pt}_{new}[fid]$, $ct_f \leftarrow \mathbf{f}^{ct}_{vl}[fid]$
$\mathcal{U}$:     $\boxed{\{O\}_z \leftarrow \mathsf{ComDiff}_{line}(f, f')}$
$\mathcal{U}$:     $\boxed{\text{for } i \in [1, z]}$
$\mathcal{U}$:         $\boxed{ct_l \leftarrow Enc(k, O_i.m),\ O'_i = (O_i.op.O_i.idx, ct_l)}$
$\mathcal{U}$:     $\boxed{ct'_f \leftarrow O'_z(\ldots(O'_1(ct_f)))}$, $\boxed{\{O\}_z \leftarrow \mathsf{ComDiff}_{char}(f, f')}$
$\mathcal{U}$:     $\boxed{ct_o \leftarrow Enc(k, \{O\}_z),\ ct'_f \leftarrow (ct_f, ct_o)}$
$\mathcal{U}$:     add $ct'_f \rightarrow \mathbf{f}^{ct}_{new}$, $\mathbf{f}^{pt}_{new}[fid] \rightarrow \mathbf{f}^{pt}_{new}$
$\mathcal{U}$: for $fid \in \mathbf{f}^{pt}_{new}.Fid \backslash \mathbf{f}^{pt}_{vl}.Fid$
$\mathcal{U}$:     $f' \leftarrow \mathbf{f}^{pt}_{new}[fid]$
$\mathcal{U}$:     $\boxed{\text{for } j \in [1, f'.l]}$     / $f'.l$ is # of lines in $f'$
$\mathcal{U}$:         $\boxed{lct_j \leftarrow Enc(k, l_j)}$        / encrypt by line
$\mathcal{U}$:     $\boxed{ct'_f \leftarrow (lct_1, \ldots, lct_{f'.l})}$    $\boxed{ct'_f \leftarrow Enc(k, f')}$
$\mathcal{U}$:     add $ct'_f \rightarrow \mathbf{f}^{ct}_{new}$, $\mathbf{f}^{pt}_{new}[fid] \rightarrow \mathbf{f}^{pt}_{new}$
$\mathcal{U}$: for $fid \in \mathbf{f}^{ct}_{vl}.Fid \backslash \mathbf{f}^{pt}_{new}.Fid$
$\mathcal{U}$:     add $\mathbf{f}^{ct}_{vl}[fid] \rightarrow \mathbf{f}^{ct}_{new}$, $\mathbf{f}^{pt}_{vl}[fid] \rightarrow \mathbf{f}^{pt}_{new}$
$\mathcal{U}$: $h' \leftarrow \mathtt{MerkleDAG}(\mathbf{f}^{ct}_{new}, f_{acs})$ / each $ct'_f$ and $f_{acs}$ are leaf nodes
$\mathcal{U}$: $\sigma' \leftarrow \mathsf{Sign}(sk_s, rid \| uid \| h')$, update file $f'_{tag} \leftarrow (uid, \sigma')$
$\mathcal{U}$: $repo^{ct}_{new} \leftarrow (\mathbf{f}^{ct}_{new}, f_{acs}, f'_{tag})$, $repo^{pt}_{new} \leftarrow (\mathbf{f}^{pt}_{new}, f^{pt}_{acs})$
$\mathcal{U}$: Send $uid, sid, rid, repo^{ct}_{new}$ to $\mathcal{S}$
$\mathcal{S}$: $req : st_S.\{usr, ses, Rid, repo^{ct}\}$
$\mathcal{S}$: Update $repo^{ct} \leftarrow repo^{ct}_{new}$
```

```
$< \mathcal{U}_{pull}(st_U, km, rid, repo_{old}), \mathcal{S}_{pull}(st_S) >$
$\mathcal{U}$: $req : st_U.\{uid, sid\}$, send $uid, sid, rid, \mathbf{v}_{old}$ to $\mathcal{S}$
$\mathcal{S}$: $req : st_S.\{usr, ses, Rid, repo^{ct}_{new}\}$.   if $ses[sid] \neq uid$, $Fail$
$\mathcal{S}$: for $vi \in repo^{ct}_{new}.\mathbf{v} \backslash \mathbf{v}_{old}$
$\mathcal{S}$:     Send $repo^{ct}_{vi} = (\mathbf{f}^{ct}, f_{acs}, f_{tag})$ to $\mathcal{U}$
$\mathcal{U}$: parse $rid = uid_o \| nonce$, $repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$
$\mathcal{U}$: if $uid = uid_o$    $k \leftarrow KDF(mk, rid)$
$\mathcal{U}$: else    $c_k \leftarrow f_{acs}.R[uid]$, $k \leftarrow \mathsf{PKE.Dec}(sk^{uid}_e, c_k)$
$\mathcal{U}$: set $repo^{pt}_{new} \leftarrow repo^{pt}_{old}$, $repo^{ct}_{new} \leftarrow repo^{ct}_{old}$
$\mathcal{U}$: for each $repo^{ct}_{vi} = (\mathbf{f}^{ct}, f_{acs}, f_{tag})$ sent from $\mathcal{S}$
$\mathcal{U}$:     parse $f_{acs} = (R, W, \sigma_{acs})$ and $f_{tag} = (uid_w, \sigma)$
$\mathcal{U}$:     $rh \leftarrow \mathtt{MerkleDAG}(\mathbf{f}^{ct}, f_{acs})$, $r \leftarrow Hash(rid \| uid_w \| rh)$
$\mathcal{U}$:     if $\neg\mathsf{Vrfy}(pk^{uid_o}_s, f_{acs}) \vee uid_w \notin f_{acs}.W \vee$
$\mathcal{U}$:         $\neg\mathsf{Vrfy}(pk^{uid_w}_s, r, \sigma)$     then $Fail$
$\mathcal{U}$:     for each $ct_i \in \mathbf{f}^{ct}$
$\mathcal{U}$:         $\boxed{\text{for } j \in [1, ct_i.l] :\quad l_j \leftarrow Dec(k, l_j)}$
$\mathcal{U}$:         $\boxed{f_i \leftarrow (l_1, \ldots, l_{ct_i.l})}$
$\mathcal{U}$:         $\boxed{\text{parse } ct_i = (ct_{i_0}, ct_{o_1} \ldots ct_{o_z})}$    / $z$ is # of update in $ct_i$
$\mathcal{U}$:         $\boxed{f_{i_0} \leftarrow Dec(k, ct_{i_0})}$
$\mathcal{U}$:         $\boxed{\text{for } j \in [1, z]\quad O_j \leftarrow Dec(k, ct_{o_j})}$
$\mathcal{U}$:         $\boxed{f_i \leftarrow O_z(\ldots(O_1(f_{i_0})))}$
$\mathcal{U}$:         add $f_i \rightarrow \mathbf{f}^{pt}$
$\mathcal{U}$:     add $repo^{pt}_{new} \leftarrow repo^{pt}_{vi} = (\mathbf{f}^{pt}, f_{acs})$, $repo^{ct}_{new} \leftarrow repo^{ct}_{vi}$
$\mathcal{U}$: get $repo_{new} \leftarrow (repo^{ct}_{new}, repo^{ct}_{new})$
```

```
$< \mathcal{U}_{share_I}(st_U, km, rid, uid_{re}, acs, repo_{old}), \mathcal{S}_{share_I}(st_S) >$
$\mathcal{U}$: $req : st_U.\{uid, sid\}$, parse $repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$
$\mathcal{U}$: parse $repo^{ct}_{vl} = (\mathbf{f}^{ct}, f_{acs}, f_{tag}) \in repo^{ct}_{old}$, $rid = (uid_o, nonce)$
$\mathcal{U}$: if $uid \neq uid_o, Fail$
$\mathcal{U}$: $k \leftarrow \mathsf{KDF}(mk, rid)$, $ct_{shr} \leftarrow \mathsf{PKE.Enc}(pk^{uid_{re}}_e, uid \| k)$
$\mathcal{U}$: $f'_{acs}.R \leftarrow f_{acs}.R \cup \{(uid_{re}, ct_{shr})\}$
$\mathcal{U}$: if $acs = \mathsf{write}$ then $f'_{acs}.W \leftarrow f_{acs}.W \cup \{uid_{re}\}$
$\mathcal{U}$: $f'_{acs}.\sigma \leftarrow \mathsf{Sign}(sk_s, f_{acs}.W \| f_{acs}.R)$
$\mathcal{U}$: $f'_{acs} \leftarrow (f'_{acs}.R, f'_{acs}.W, f'_{acs}.\sigma)$
$\mathcal{U}$: $rh' \leftarrow \mathtt{MerkleDAG}(\mathbf{f}^{ct}, f'_{acs})$, $h' \leftarrow Hash(rid \| uid \| rh')$
$\mathcal{U}$: $\sigma' \leftarrow \mathsf{Sign}(sk^{uid}_s, h')$, $f'_{tag} = (uid, \sigma')$
$\mathcal{U}$: $repo^{ct}_{new} = repo^{ct}_{old} \cup \{(\mathbf{f}^{ct}, f'_{acs}, f'_{tag})\}$
$\mathcal{U}$: Send $uid, sid, rid, uid_{re}, acs, f'_{acs}, f'_{tag}$ to $\mathcal{S}$
$\mathcal{S}$: $req : st_S.\{usr, ses, repo^{ct}, shr\}$
$\mathcal{S}$: $oob \leftarrow\!\!\$$, add $shr[rid] \leftarrow (uid_{re}, oob, acs)$
$\mathcal{S}$: add new version $(\mathbf{f}^{ct}, f'_{acs}, f'_{tag})$ to $repo^{ct}$, send $oob$ to $\mathcal{U}$
```

```
$< \mathcal{U}_{share_{II}}(st_U, rid, oob), \mathcal{S}_{share_{II}}(st_S) >$
$\mathcal{U}$: $req : st_U.\{uid, sid\}$, send $uid, sid, rid, oob$ to $\mathcal{S}$
$\mathcal{S}$: $req : st_S.\{usr, ses, shr, A\}$
$\mathcal{S}$: if $ses[sid] \neq uid$ or $(uid, oob, acs) \notin shr[rid], Fail$
$\mathcal{S}$: add $(uid, acs) \rightarrow A[rid]$.
```

Figure 6: The constructions of SGit**, where boxed purple part with solid line belongs to** SGitLine**, and boxed teal part with dashed line belongs to** SGitChar**.**

repetitive data. We can apply standard encryption on a smaller unit of data, aligning with the version control system's common data processing unit, so that any changes can be located in a more fine-grained way, not as a whole file or repository.

Git already treats lines as essential units, organizing data based on line, and utilizing line indexes to display differences. So we apply symmetric encryption to the repository in a line-wise way. For each data update, only those changed lines are re-encrypted, and unchanged lines remain unchanged in terms of ciphertexts. We propose using lines as the treatment unit, as the name SGitLine indicates.

Leveraging this existing organization eliminates the need for partitioning and reconstruction. Additionally, the flexibility of line length allows users to customize it, potentially mitigating the significance of IV storage, especially in average cases with longer lines.

At a high level, SGitLine involves encrypting data line by line, maintaining the ciphertext unchanged if the plaintext remains the same. In the repository initialization procedure, the user first encrypts every line for each file and then takes the entire ciphertext version as a whole together with the repository id and the user's id to sign using the user's private key. Digital signature helps for the integrity and unforgeability. The repository id and user id bind the repository version to the specified repository and the user who writes this version. In subsequent update operations, the user compares the line-wise differences between two versions of the plaintext repository before and after the update. These differences indicate insert and delete modifications, such as which lines are deleted, and where to insert a line of content. With the modifications, the user can encrypt the contents line by line in each modification, and operate each modification on the ciphertext repository with the corresponding ciphertext. Then, the user follows the same way to sign the entire version of the ciphertext repository. It is evident that in the ciphertext repository, unchanged lines remain unchanged. Consequently, the computation, communication, and storage cost of one more version is only linear to the size of changed lines, not the entire version of the repository. In the pull procedure, the user first interacts with the Git server to retrieve the entire version of the ciphertext. Then, the user checks the signatures to make sure the pulled version is written by a valid user with write access to the repository. After passing all checks, the user decrypts ciphertexts line-by-line for each file to form the plaintext repository. The share procedure includes requesting access permission to the server (which relies on the Git server api is), appending the encrypted key material to the access control file, and authorizing the sharing via signing the access file on the repository. The receiver accepts the sharing access via the server share api. Later, when reading or writing the repository, the receiver first decrypts the encrypted key file to get the data encryption key.

Storage drawback. While SGitLine offers substantial storage savings compared to simply applying encryption to the entire repository, it may incur higher storage costs for code repositories. This is particularly true for repositories where each line tends to be very short. Additionally, in the case of minor patch updates involving only a few word changes on certain lines (e.g., correcting typos), the storage cost could be significantly larger compared to the plaintext repository.

Security drawback: We will prove that SGitLine satisfies the weak data confidentiality in Section 4.4.3. Regarding weak confidentiality, it does not protect the updated position. While SGitLine ensures IND-CPA security for the repository data, it does not conceal the update operation itself. Specifically, the position information about which lines get deleted and which lines get newly inserted remains unprotected. Also, the length of each line of the file is unprotected.

4.2 Construction: SGitChar

A secure construction should avoid deterministic encryption, which leaks data patterns. To reduce cost, encryption cannot be trivially applied to all files, as this would make the total computation and storage cost scale linearly with the number of repository versions n and the size of each version f. Our goal is to achieve both security (at least standard semantic security) and high efficiency where the update cost is independent of the whole repository size, ideally only depending on the size of the modified content.

The high-level idea of SGitChar is that by shifting our perspective on repository updates, we can consolidate various update operations within a file into a single operation encapsulating a set of *differences*. With this approach, a single insert operation containing a line of content encompassing all differences can record all the modifications with minimal storage cost. More importantly, by aggregating all differences into one operation and storing them in one position (at the end of the file), which is independent of any modification position, we effectively conceal the position of each internal update operation. The encryption cost is only related to the size of the difference, not the entire version of the repository, which reduces the computation cost. For similar reasons, the communication cost of push/pull is also minimal and depends only on the size of the difference between the two consecutive versions.

Furthermore, the straightforward method to achieve unforgeability is signing all files of one repository version, which costs linear time to the size of the version. Inspired by the Git tree graph to organize the repository directory and compute the root hash of all files, we do not directly sign on all files but sign on the root hash of all files and their hierarchical organization structure, which is known as a tree structure. In this way, a single file change only needs hash computation on the changed file as a leaf node and on the intermediate nodes to the root of the tree that depends only on the height of the leaf node (not related to the size of the entire version of files).

Detailed construction. Let $\Pi_{\mathsf{authenticate}} = (\Pi_{\mathsf{AuthReg}}, \Pi_{\mathsf{Auth}})$ be any authentication mechanism. Let $\Pi_{SE} = \{\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}\}$ be a symmetric key encryption scheme, KDF be a secure key derivation function, $Hash$ be a collision resistant hash function, and `MerkleDAG` be a directed acyclic graph structured collision resistant hash function [22]. Let $\Pi_{PKE} = \{\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}\}$ be public key encryption, and $\Pi_{DS} = \{\mathsf{KG}, \mathsf{Sign}, \mathsf{Vrfy}\}$ be digital signature. The detailed constructions of SGitChar and SGitLine are described as follows and shown in Figure 6.

– $\Pi_{reg}(uid; st_S) \rightarrow (cred, km; st'_S)$: a user registers a new account with user id uid, formally shown in Figure 7. In the registration, the user first runs the key generation algorithm KeyGen includes

$\mathcal{U}_{reg}(uid)$		$\mathcal{S}_{reg}(st_S)$
$mk \leftarrow \Pi_{SE}.\mathsf{KG}$		
$(sk_e, pk_e) \leftarrow \Pi_{PKE}.\mathsf{KG}$		
$(sk_s, pk_s) \leftarrow \Pi_{DS}.\mathsf{KG}$		
$\mathcal{U}_{\mathsf{AuthReg}}(uid)$	$\longleftrightarrow$	$\mathcal{S}_{\mathsf{AuthReg}}(st_S)$
$cred_u$		st'_S
Publish (uid, pk_e, pk_s)		

Figure 7: Registration protocol

$\mathcal{U}_{auth}(st_U, uid, pwd)$		$\mathcal{S}_{auth}(st_S)$
		$req : st_S.\{usr, ses\}$
$\mathcal{U}_{\mathsf{Auth}}(uid, cred)$	$\longleftrightarrow$	$\mathcal{S}_{\mathsf{Auth}}(st_S)$
b		b
		if $b \neq 0$,
		$sid \leftarrow\$$, s.t., $sid \notin ses$
		$ses[sid] \leftarrow (uid)$
$st_U \leftarrow (uid, sid)$	$\xleftarrow{sid}$	**else** $sid \leftarrow \perp$

Figure 8: Authentication protocol

running symmetric key encryption's key generation algorithm $\Pi_{SE}.\mathsf{KG}$, the key generation of public key encryption $\Pi_{PKE}.\mathsf{KG}$, and digital signature's key generation algorithm $\Pi_{DS}.\mathsf{KG}$ to generate the key material $km = (mk, pk_e, sk_e, pk_s, sk_s)$ including a master secret key mk, a key pair (pk_e, sk_e) for public key encryption, and a key pair (pk_s, sk_s) for digital signature. Then the user and the server run Π_{AuthReg} of an authentication mechanism to let the user get a credential $cred$ and the server update st_S for later authentication.

– $\Pi_{auth}(uid, cred; st_S) \rightarrow (st'_U; st'_S)$: a user authenticates to the server by running Π_{Auth}, which is formally depicted in Figure 8. Concretely, the user interacts with the server to run the authentication procedure Π_{Auth} and sets the user session state $st_U \leftarrow (uid, sid)$. In the current programmable access to Git and our experiment, we use a token-based authentication method, i.e., in the registration phase, the token is randomly generated by the server and kept secret by the user for authentication. During the authentication procedure, the user shows the token to authenticate to the server.

– $\Pi_{init}(st_U, km, rid, \mathbf{f}^{pt}; st_S) \rightarrow (repo_{new}, st'_U; st'_S)$: In an active session, a user first encrypts each file in $\mathbf{f}^{pt}$ with the key k derived from the master key mk and the repository id rid to get a ciphertext version $\mathbf{f}^{ct}$. For SGitChar, the user takes each file as a whole to run encryption and get the ciphertext as content. For SGitLine, the file is parsed by line, and users take each line as input to run encryption to get the ciphertext as the content of the corresponding line of the ciphertext file. Then, the user runs $rh \leftarrow \mathtt{MerkleDAG}(\mathbf{f}^{ct})$ to hash on ciphertexts and the file structure, and gets the signature tag $\sigma \rightarrow \mathsf{Sign}(sk, Hash(rid\|uid\|rh))$, then creates a new repository with the repository id $rid = (uid, nonce)$, required to be globally unique and include the creator's uid and a random nonce. The new ciphertext repository of the client and Git server includes tracking ciphertext files $\mathbf{f}^{ct}$, an empty access file $f_{acs} = \emptyset$, and a tag message $f_{tag} = (uid, \sigma)$.

– $\Pi_{update}(st_U, km, rid, repo_{old}, \mathbf{f}^{pt}_{new}; st_S) \rightarrow (st'_U, repo_{new}; st'_S)$: a user first takes latest committed plaintext and ciphertext repository $repo_{old} = ({repo_{old}}^{pt}, {repo_{old}}^{ct})$, and the new plaintext files $\mathbf{f}^{pt}_{new}$ as input to get ciphertext files $\mathbf{f}^{ct}_{new}$.

Concretely, for newly added files, the user encrypts files as in initialization to get the ciphertext version. For updated files, the user computes the differences with the last committed plain file and encrypts the differences to get the corresponding updated ciphertext file. SGitChar runs $\mathsf{ComDiff}_{char}$ on the new and last prior version of the plain file to get a set of modifications $\{O\}_z$, encrypts $\{O\}_z$, and appends the ciphertext at the end of last prior version of ciphertext file, to get the new version of corresponding ciphertext file. While SGitLine runs $\mathsf{ComDiff}_{line}$ to get $\{O\}_z$ and only encrypts the contents of the insert operation to replace the plaintext content and leave the delete operation unmodified. Then, those operations are applied to the last prior ciphertext files to get new versions that correspond to them.

With new ciphertext files $\mathbf{f}^{ct}_{new}$ and unchanged last prior ciphertext files as the new version of the repository, the user then hashes and signs the new version to get a tag, then commits it with a message including the tag and push to the Git server.

– $\Pi_{pull}(st_U, km, rid, repo_{old}; st_S) \rightarrow (st'_U, repo_{new}; st'_S)$: In an active session, a user interacts with the server to fetch the missing versions from the server. Then, the user runs the signature verification algorithm Vrfy to check the integrity of each missing version and access control file. If all checks pass, the user derives or decrypts to get the data encryption key. Then, do the following to get a new plaintext repository.

For each missing version from old to new, SGitChar runs the decryption algorithm on the differences compared with the last prior version to get a set of modification operations and applies modifications on the last prior plaintext version to get the next version of the plaintext repository. SGitLine can directly decrypt each file line-by-line to get the next new version or, based on the difference, decrypt the contents of insert operations as new content and leave the content of delete operation empty, then apply those operations on the last prior version of plaintext files to get the next plaintext version.

– $\Pi_{share_I}(st_U, km, rid, uid_{re}, acs, repo_{old}; st_S) \rightarrow (st'_U, repo_{new}; st'_S)$: a user interacts with the server to add a collaborator with the id uid_{re} for the repository identified by rid so that the user uid_{re} also has access acs to the repository. The user generates ct_{shr}, which is an encryption of the repository encryption key and sender uid under a collaborator's public key $\mathsf{PKE.Enc}(pk_e^{uid_{re}}, uid\|k)$. Then the user updates the access control file accordingly by adding one entry (uid_{re}, ct_{shr}) to the read list $f_{acs}.R$ repository by inserting ct_{share} to the shared file. If acs is write access, then add uid_{re} to write list $f_{acs}.W$. Then, the user hashes and signs the repository according to the new ciphertext version, commits locally, and pushes it to the Git server.

– $\Pi_{share_{II}}(st_U, rid, oob; st_S) \rightarrow (st'_U; st'_S)$ After receiving the out-of-band message from the Git server, the user can take this message as input to interact with the server to accept the shared repository access right via APIs of the Git server.

4.3 Construction extensions

We give three further extensions to support more functionalities, including the delete and merge function, to enable portability and to optimize retrieval efficiency.

More functionalities. Based on the syntax and Git supported operations, we can easily extend our E2E encrypted Git services to support more functions, such as file deletion and branch merge. Both are special cases of the update operation Π_{update}, which can be captured by our general construction, and do not affect the security.

To delete a file, the update protocol can achieve it with the input $\mathbf{f}_{new}^{pt}$, which includes a file with the same file name as the file to be deleted and empty content.

To merge two branches, user can run update protocol with specified inputs: $repo_{old}$ and $\mathbf{f}_{new}^{pt}$, where $repo_{old}$ is one branch of the repository (identified branch 1), and $\mathbf{f}_{new}^{pt}$ is the files in the other branch (identified as branch 2) which are different from branch 1. Git provides basic functions to find the needed $\mathbf{f}_{new}^{pt}$, such as `git diff`. The method works as it applies one branch's update to the other branch so that the updated version includes all updates of the two branches. The result is the correct merge version as it is independent of the order of branches. Please note that the Git merge function only merges two branches without conflict updates, i.e., different updates on the same file, so does our method.

Achieving semantic security and portability simultaneously. The above secure Git services require users to keep their secret keys by themselves. The trivial way is to store the secrets locally, which hinders the portability and brings extra inconvenience when users want to change devices or just access remote repositories via multiple devices. Local storage is vulnerable to all kinds of fishing attacks, viruses, ransomwares, etc. To get rid of the reliance on secure local storage and improve the portability, we propose a solution to integrate password-based key management into E2EE Git.

Currently, users have three secrets: password, private key, and secret key. Users use a password to authenticate to the Git server, use the private key to authenticate to other users, and use the secret key to derive a data encryption key for data encryption. Note that among the three secrets, only the password is easy for users to memorize and bring everywhere, and thus we consider using password-based key management to improve the portability. However, we know that a password has low entropy and is vulnerable to dictionary attacks when the Git server gets compromised or the server storage gets breached.

We utilize End-to-Same-End encryption (E2SE for short) design idea, where another server is introduced to increase the user's entropy for each server. We integrate E2SE into E2EE Git by introducing a new server acting as a key server and letting the GitHub server act as a storage server, so that users use passwords to authenticate to two servers and derive a master key for encrypting/decrypting the private key and secret key.

Further optimizations. For SGitChar, a single version of the ciphertext file includes the initial version of the ciphertext and a sequence of updates. It is efficient in terms of repository storage cost, update communication size, and encryption time, only related to the size of the difference. But in one case that client does not have the repository locally and only wants to fetch a single latest version of the repository, the communication cost and decryption time are linear to the versions of the repository, as the latest version includes all the previous updates. SGitLine does not have such an issue due to each ciphertext version is history independent. To mitigate the special case cost of SGitChar due to history dependence, we can have further optimization by setting a length of history dependence, e.g., 6 versions. Concretely, for every six updates of a file, users treat the file as the first version to directly encrypt an entire version from scratch, which is independent of the history. Even if users do not have any local repository and only fetch a specified version, the communication cost at most includes five updates, and the decryption overhead is at most linear to five update differences.

4.4 Security analysis

4.4.1 *Data confidentiality of* SGitChar.

We give a formal proof of Thm.1 that SGitChar satisfies the data confidentiality defined in Def. 1 in the following.

Theorem 1 (Data confidentiality). *Let* $\Pi_{SE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be an IND-CPA secure symmetric encryption,* $\Pi_{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be an IND-CPA public-key encryption,* $\Pi_{Sig} = (\mathsf{KG}, \mathsf{sign}, \mathsf{Vrfy})$ *be a strongly existentially unforgeable digital signature,* KDF *be a random oracle. Let* `MerkleDAG` *be a directed acyclic graph structured collision-resistant hash function.* SGitChar *has data confidentiality, i.e.,*

$$\mathrm{Adv}^{\mathsf{CONF}}_{\mathsf{SGitChar},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{CONF}}_{\mathsf{SGitChar},\mathcal{A},q} = 1] - 1/2 = negl(\lambda).$$

Proof. In a high level, we use game hop to reduce the data confidentiality to the security of underlying schemes. When playing a game with SGitLine adversary $\mathcal{A}$, challenger $\mathcal{B}$ could act as an adversary of underlying schemes and interact with their respective challenger C to answer $\mathcal{A}$'s queries. If $\mathcal{A}$ can distinguish, $\mathcal{B}$ can leverage it to break underlying schemes. We use four game hops to deal with the decryption queries of O_{pull}, the sharing queries of O_{share_I}, the encryption queries of O_{init}, and the update queries of O_{upd}, gradually reducing the data confidentiality to the repository unforgeability of SGitChar proved in Theorem 2, the IND-CPA security of Π_{PKE}, the pseudorandomness of KDF, and the IND-CPA security of Π_{SE}.

Specifically, Game 1 ignores all decryption queries of O_{pull} that involve malformed ciphertexts. The adversary $\mathcal{A}$ cannot distinguish this change due to the unforgeability of SGitChar, which is further reduced to the security of Π_{Sig} and `MerkleDAG`. Game 2 replaces the key materials shared with honest users with random values, and encrypts these values when responding to O_{share_I} queries. This modification is indistinguishable from using real key materials due to the IND-CPA security of Π_{PKE}. In Game 3, the derivation of the repository encryption key is replaced with a random value in O_{init} queries. This change is hidden from $\mathcal{A}$ due to the pseudorandomness of the KDF output when given a random input. For challenge related queries, Game 4 converts into challenges to the challenger in the IND-CPA security game of Π_{SE}, and uses the challenge responses as the corresponding ciphertexts to proceed, which is indistinguishable due to the IND-CPA security of Π_{SE}.

The details are as follows:

- Game 0: $\mathcal{B}$ acts the same as challenger in SGitLine confidentiality game.
- Game 1: we deal with the decryption oracle. For those pull queries to O_{pull} on those repositories which only honest users have access to, $\mathcal{B}$ refuses to respond if the queried ciphertext repository is not the previous queried one. $\mathcal{A}$ can distinguish Game 1 from Game 0 with negligible probability due to the repository unforgeability of SGitLine.
- Game 2: $\mathcal{B}$ first replaces all key materials shared with honest users with encryption on a random key, which is indistinguishable from Game 1 due to the IND-CPA security of PKE. When queried to the O_{pull} oracle with shared repository on shared honest user, $\mathcal{B}$ just looks up the table to find the real data encryption key instead of decryption to get the key. This ensures that the shared key material with honest users does not leak any information about the real data encryption key.
- Game 3: for each initialization query to O_{init} on honest users, $\mathcal{B}$ replaces repository encryption key with a random value. Even if the malicious user is shared by the honest user, the random value is indistinguishable from the correct output of key deviation function on honest user's master key mk and repository id rid as input. So that the honest user's mk is never leaked to corrupted users. Due to Game 2, the key material of the repository with only honest users is never leaked to $\mathcal{A}$.
- Game 4: given the challenge two files, if it is an initialization query, $\mathcal{B}$ directly forwards the challenge to the challenger of symmetric encryption $C_{\Pi_{SE}}$. Later for the update queries on the challenge repository, $\mathcal{B}$ comparing the differences between update file f and two challenges by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{char}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{char}(f_1, f)$. $\mathcal{B}$ checks the validity of $\{O_0.m\}$ and $\{O_1.m\}$ based on different conditions for SGitChar and SGitLine. If the challenge is valid without trivial win, $\mathcal{B}$ forwards $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $C_{\Pi_{SE}}$. If the challenge query is an update query, $\mathcal{B}$ first calculate two sets of challenge modifications in terms of their prior file f by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{char}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{char}(f_1, f)$. Then $\mathcal{B}$ submits the different modification messages $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $C_{\Pi_{SE}}$. $\mathcal{B}$ follows the same way to deal with later update queries on the challenge repository. Finally, $\mathcal{B}$ forwards $\mathcal{A}$'s guess to $C_{\Pi_{SE}}$. If $\mathcal{A}$ has a non-negligible probability to distinguish the two challenges, then $\mathcal{B}$ can break the IND-CPA security of Π_{SE}.

As a result, SGitChar has data confidentiality. □

4.4.2 *Repository unforgeability of* SGit.

We formally prove the Thm. 2 that two SGit constructions SGitLine and SGitChar satisfy the repository unforgeability defined in Def. 3.

Theorem 2 (Repository unforgeability). *Let* Π_{Sig} = (KG, Sign, Vrfy) *be a strongly unforgeable digital signature scheme and* `MerkleDAG` *be a directed acyclic graph structured collision-resistant hash function.* SGitLine *and* SGitChar *have repository unforgeability, i.e.,* $\mathrm{Adv}^{\mathsf{UNF}}_{\mathsf{SGitLine},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{UNF}}_{\mathsf{SGitLine},\mathcal{A},q} = 1] = negl(\lambda)$ *and* $\mathrm{Adv}^{\mathsf{UNF}}_{\mathsf{SGitChar},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{UNF}}_{\mathsf{SGitChar},\mathcal{A},q} = 1] = negl(\lambda)$.

Proof. Intuitively, we begin by assuming that $\mathcal{A}$ produces a successful forgery, which must fall into one of three possible cases: a new signature, a new hash root, or new repository contents. However, each case contradicts the original assumptions on the underlying building blocks—specifically, the strong unforgeability of Π_{Sig} and the collision resistance of `MerkleDAG`. Therefore, $\mathcal{A}$ cannot produce a successful forgery, and SGitChar satisfies repository unforgeability.

Since SGitLine and SGitChar use the same components to ensure unforgeability, namely, applying a structured hash function to an entire repository version and signing the resulting hash root, the proof of repository unforgeability applies equally to both constructions.

Concretely, a valid forgery $repo*_{ct} = repo^h_{ct}, f_{acs}, f_{tag}$ on target user uid^* and repository rid^* contains three parts where σ is signature on message $rid^*\|uid^*\|h$, $f_{tag} = (uid^*, \sigma)$, and $h = \mathtt{MerkleDAG}(repo^h_{ct}, f_{acs})$. The forgery is a new message signature pair. So there are two cases.

- Case 1: The signature σ is new.
- Case 2: The signature is the previous one, but the message $rid^*\|uid^*\|h$ is new.
- Case 3: Both signature and message are previous ones, but $(repo^h_{ct}, f_{acs})$ is new.

Case 1 and Case 2 mean that $\mathcal{A}$ forges a new message signature pair, which is contradictory with the strong unforgeability of digital signatures. For case 3, since the message is old, the h is also the previous one. But $(repo^h_{ct}, f_{acs})$ is new. Previously, there exists one $(repo'^h_{ct}, f'_{acs})$ such that $h = \mathtt{MerkleDAG}(repo'^h_{ct}, f'_{acs})$. We know that $h = \mathtt{MerkleDAG}(repo^h_{ct}, f_{acs})$. So there is a collision which contradicts with the collision resistent property of `MerkleDAG`. As a result, SGitLine has unforgeability. □

4.4.3 *Weak data confidentiality of* SGitLine.

We prove that the SGitLine construction satisfies weak data confidentiality defined in Def. 2.

Theorem 3 (Weak data confidentiality). *Let* Π_{SE} = (KG, Enc, Dec) *be an IND-CPA secure symmetric encryption,* Π_{PKE} = (KG, Enc, Dec) *be an IND-CPA public-key encryption,* Π_{Sig} = (KG, sign, Vrfy) *be an strong existing unforgeable digital signature,* KDF *be a random oracle. Let* `MerkleDAG` *be a directed acyclic graph structured collision resistant hash function.* SGitLine *has weak data confidentiality, i.e.,*

$$\mathrm{Adv}^{\mathsf{CONF_w}}_{\mathsf{SGitLine},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathsf{CONF_w}}_{\mathsf{SGitLine},\mathcal{A},q} = 1] - 1/2 = negl(\lambda)$$

Proof. Intuitively, we use game hop to reduce the data confidentiality to the security of the underlying schemes. When playing a game with SGitLine adversary $\mathcal{A}$, challenger $\mathcal{B}$ could act as adversary of underlying schemes and interact with their respective challenger C to answer $\mathcal{A}$'s queries. If $\mathcal{A}$ can distinguish, $\mathcal{B}$ can leverage it to break underlying schemes. We use four game hops dealing with decryption query of O_{pull}, sharing query of O_{share_l}, encryption queries of O_{init}, and update queries of O_{upd} to reduce the security. Compared with the proof of SGitChar, the only difference of the weak confidentiality proof of SGitLine is in Game 4 for dealing with challenges to the O_{init} or O_{upd} oracles and the following update queries to the O_{upd} oracle with one more restriction on the challenge update position. The details are as follows:

- Game 0, $\mathcal{B}$ acts the same as challenger in SGitLine confidentiality game.
- Game 1, we deal with the decryption oracle. For those pull queries to O_{pull} on those repositories which only honest users have access to, $\mathcal{B}$ refuses to respond if the queried ciphertext repository is not previously queried one. $\mathcal{A}$ can distinguish Game 1 from Game 0 with negligible probability due to the repository unforgeability of SGitLine.
- Game 2, $\mathcal{B}$ first replaces all key materials shared with honest users with encryption on a random key, which is indistinguishable from Game 1 due to the IND-CPA security of PKE. When queried to the O_{pull} oracle with shared repository on shared honest user, $\mathcal{B}$ just looks up the table to find the real data encryption key instead of decryption to get the key. This ensures that the shared key material with honest users does not leak any information about the real data encryption key.
- Game 3, for each initialization query to O_{init} on honest users, $\mathcal{B}$ replaces repository encryption key with a random value. Even if a malicious user is shared by the honest user, the random value is indistinguishable from the correct output of the key deviation function on the honest user's master key *mk* and repository id *rid* as input. So that the honest user's *mk* is never leaked to corrupted users. Due to Game 2, the key material of the repository with only honest users is never leaked to $\mathcal{A}$.
- Game 4, given the challenge two files, if it is an initialization query, $\mathcal{B}$ directly forwards the challenge to the challenger of symmetric encryption $C_{\Pi_{SE}}$. Later for the update queries on the challenge repository, $\mathcal{B}$ comparing the differences between update file f and two challenges by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{line}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{line}(f_1, f)$. $\mathcal{B}$ checks the validity of $\{O_0.m\}$ and $\{O_1.m\}$ based on the trivial condition for SGitLine including the restriction of the consistent update position. If the challenge is valid without trivial wins, $\mathcal{B}$ forwards $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $C_{\Pi_{SE}}$. If the challenge query is an update query, $\mathcal{B}$ first calculates two sets of challenge modifications in terms of their prior file f by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{line}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{line}(f_1, f)$. Then, $\mathcal{B}$ submits the different modification messages $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $C_{\Pi_{SE}}$. $\mathcal{B}$ follows the same way to deal with later update queries on the challenge repository. Finally, $\mathcal{B}$ forwards $\mathcal{A}$'s guess to $C_{\Pi_{SE}}$. If $\mathcal{A}$ has a non-negligible probability to distinguish the two challenges, then $\mathcal{B}$ can break the IND-CPA security of Π_{SE}.

As a result, SGitLine has weak data confidentiality. □

5 IMPLEMENTATION AND EVALUATION

Implementation. We implemented both SGitLine and SGitChar using Python and the pycryptodome library and used AES-CTR as the encryption algorithm, ECDSA as a signature scheme, SHA-256 as the hash function, and HKDF-SHA-256 as the key derivation function. We will open-source it soon. For a fair comparison, we re-implemented the deterministic encryption-based scheme adopted by Git-crypt [4] using Python, where AES-CTR encrypts each file with an initialization vector (IV) derived from the SHA-1 HMAC of the file. We also implemented Trivial-enc-sign, where the whole updated version of a repository would be re-encrypted before pushing.

For SGitLine, we utilize `git diff` to obtain the line-wise difference for each update. For SGitChar, we utilize the diff_match_patch package [19] to obtain the character-wise delta. To record the user's signature in the current commit, we take the following steps: 1) run the `git commit` command, 2) sign on the commit information including the parent commit hash value, the `MerkleDAG` hash value of the current commit, the author, timestamps, and the commit message; 3) use the `git --amend` command to update the commit message with the signature. In this way, a user can obtain the updated ciphertext and the signature from one commit and then verify them. Git uses SHA-1 by default, while due to the insecurity of SHA-1, we recommend utilizing SHA-256 as the default hash function.

Experiments. The local Git repositories were hosted on a Windows laptop with an Intel Core i7 processor (2.1 GHz) and 32 GB RAM. We also carried out the experiments on Amazon Web Service (AWS for short), where the repositories are hosted on an AWS virtual machine with Ubuntu (64-bit), 1 vCPU, and 30 GB of disk storage. We used Git tools and the GitHub API to interact with GitHub, deploying a remote repository on the GitHub server. To compare our schemes with the other two in typical scenarios, we selected five of the top ten rated code repositories on GitHub, considering the variety in their scale and number of files. The selected repositories are awesome [3], free-programming-books (FPB) [25], bootstrap [10], react [33], and freeCodeCamp (FCC) [26]. Additionally, we included a paper repository (denoted as DecRepo), which mainly contains LaTeX files of an academic manuscript and has a different structure and pattern compared to conventional code repositories. The specific information is provided in Table 5.

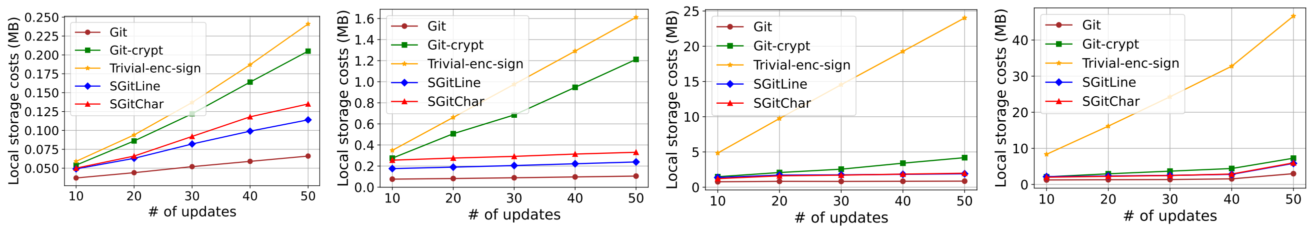
We evaluated the four schemes on these six repositories, comparing their performance in terms of communication, computation, end-to-end time, and local storage costs. In the initialization, the first ciphertext version of a repository is generated locally and pushed to the GitHub server. Regarding one version update, we randomly select ten commits from each repository and calculate the average computation costs for updating the ciphertext, as well as the average communication costs for pushing it. We also utilize the same commits to test the recovery costs, supposing that the client has the original version of a commit, another collaborator makes a new commit to GitHub, and the client needs to update the local repository. We also test the average end-to-end time of the randomly selected ten commits, including local computation delay and communication delay of pushing to the GitHub server. For storage costs, we utilize the first commit of each repository as

Table 2: The communication costs of each operation on six repositories using different schemes.

Repo.	Initialization (KB)					One Version Update/Recovery (KB)				
	Git	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign	Git	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign
awesome [3]	0.54	1.06	1.36	1.06	1.06	0.33	0.48	0.44	38.17	0.21 MB
FPB [25]	0.94	1.58	1.99	1.58	1.58	0.41	0.69	0.58	19.62	0.68 MB
bootstrap [10]	253.42	400.50	471.57	400.50	400.50	0.80	4.51	3.31	122.78	2.52 MB
react [33]	620.61	984.00	984.00	984.00	984.00	1.92	8.77	10.20	49.81	23.82 MB
FCC [26]	1.04	1.69	2.30	1.69	1.70	2.39	11.49	11.98	120.61	59.57 MB
DocRepo	445.93	729.38	847.20	719.89	729.45	2.23	10.41	11.01	74.50	0.83 MB

Table 3: The computation overhead of updating six repositories under different schemes.

Repository	SGitChar (s)			SGitLine (s)			Git-crypt (s)	Trivial-enc-sign(s)
	Compare	Encrypt	Total	Compare	Enc+update	Total		
awesome [3]	0.0003	0.0001	0.0004	0.0277	0.0001	0.0278	0.0002	0.0008
FPB [25]	0.0003	0.0001	0.0004	0.0275	0.0001	0.0276	0.0001	0.0045
bootstrap [10]	0.1004	0.0001	0.1005	0.0287	0.0010	0.0297	0.0006	0.0229
react [33]	0.0888	0.0001	0.0889	0.0376	0.0010	0.0386	0.0003	0.1235
FCC [26]	0.0683	0.0002	0.0685	0.0340	0.0009	0.0349	0.0008	0.6045
DocRepo	0.3337	0.0002	0.3339	0.0336	0.0008	0.0344	0.0005	0.0033



(a) Repo awesome [3] **(b) Repo FPB [25]** **(c) Repo bootstrap [10]** **(d) Paper Repository**

Figure 9: The costs of storing the repositories using different schemes.

the initial version and record the storage costs after 10, 20, 30, 40, and 50 commits. The detailed description is provided as follows.

In the initialization phase, we measure the computation costs of running the initialization algorithm on the first version of a repository and the communication costs of pushing the ciphertext. To evaluate the update costs, we randomly select a commit from the repository, with each commit corresponding to two versions: the original and the updated version. We first run the initialization algorithm on the original version, then apply the update algorithm to generate the ciphertext for the updated version, and finally push the ciphertext to the GitHub server. To ensure generality, we randomly select ten commits from each repository and calculate the average computation costs for updating the ciphertext, as well as the average communication costs for pushing it. We also evaluate the end-to-end delay of one update. For Git, the end-to-end delay includes the time of pushing the updated version to the GitHub server. For SGitChar and SGitLine, it contains the time of delta computation, encryption, signing, and pushing, and the end-to-end time of Trivial-enc-sign contains the same components except for delta computation. For Git-crypt, it only includes the delay of encrypting the modified files and pushing.

Regarding recovery, we use the same ten commits to test the communication costs of pulling data. Specifically, we assume that the client has the original version of a commit, and another collaborator makes a new commit to GitHub. We then measure the communication costs of pulling the updated version from the GitHub server and the computation costs of recovering the updated version. We measure the costs of storing each encrypted repository using the four schemes and the costs of storing the plaintext repository using plain Git. We utilize the first commit of each repository as the initial version and record the storage costs after 10, 20, 30, 40, and 50 commits. We run the `git gc` command to pack the objects that have been generated after we commit a new version. This command allows us only to store the initial version and the delta generated by a new commit.

Evaluation summary. The communication costs of the four schemes are shown in Table 2. Regarding updates, both SGitLine and SGitChar perform much better than the other two schemes, especially for a large repository with minor updates. SGitChar takes fewer communication costs than SGitLine, except for some special cases. In terms of initialization, SGitChar achieves comparable performance to Git-crypt and Trivial-enc-sign.

The computation costs of updates are provided in Table 3. Our schemes SGitChar and SGitLine generally perform better than Trivial-enc-sign, apart from a few special scenarios, such as updates throughout the entire repository. Regarding the end-to-end time cost shown in Figure 10, SGitLine consistently outperforms Trivial-enc-sign, as it may incur higher computation costs but achieves significantly lower communication overhead, resulting in overall greater efficiency.

Figure 9 shows that SGitChar and SGitLine take less storage costs compared with Git-crypt and Trivial-enc-sign as the number of updates increases, and SGitChar generally outperforms SGitLine, except for some special cases. The detailed analysis of special cases is provided later.

Communication costs. As the costs of initialization and recovery shown in Table 2, SGitChar, Git-crypt, and Trivial-enc-sign share similar costs, while SGitLine incurs a little bit more costs, since SGitLine needs to store a nonce for each line. We observe that the pull costs of recovery are almost as large as the push costs of updating a commit to the GitHub server, since the pull and push communication costs are all determined by the delta of the commit. For both push and pull operations, SGitLine and SGitChar have much fewer communication costs than Git-crypt and Trivial-enc-sign, especially for a large repository with minor updates. Our constructions are generally $2 \sim 3$ orders of magnitude more efficient than Trivial-enc-signin in terms of update/recovery communication cost. In general, SGitChar spends fewer communication costs than SGitLine in the update phase, since the word-wise difference is usually shorter than the line-wise difference, except for some special cases (analyzed below). Particularly, the update costs of SGitChar are at most 5.6 times that of Git, which is acceptable given the strong security guarantees our scheme provides.

There are some special cases that SGitChar has slightly more communication than SGitLine, e.g., [3], [25], and [10]. The reason is besides recording the updated content itself, the character-wise delta needs to keep extra information, including the exact update position and update type (insert or delete). This extra information may cost more space than line-wise delta when a new line is inserted, a line is deleted, or multiple major modifications occur within one line.

Computation costs. The computation costs of the initialization and recovery phases are presented in Table 4. SGitChar performs as well as Trivial-enc-sign and outperforms Git-crypt and SGitLine in the initialization phase, since Git-crypt needs to compute SHA-1 HMAC from files and SGitLine needs to encrypt the file line-by-line, which costs much for files with many lines. Regarding recovery, SGitChar is more efficient than Trivial-enc-sign but slightly less efficient than Git-crypt, since SGitChar does not need to decrypt each entire file as Trivial-enc-sign, but has to decrypt the patch(es) and then apply them to the original content. We observe that there are two decryption methods of SGitLine. One is to directly decrypt files line by line. The other is to first compute the delta on the ciphertext repositories and then only decrypt the modified lines, as the client has the original version. We adopt the former because it does not need line-wise computation and is more efficient. Even with the more efficient method, SGitLine underperforms, as it has to decrypt files line by line, which is time-consuming.

The costs of the update phase are shown in Table 3. The costs of SGitChar include computing word-wise difference and encrypting it. The costs of SGitLine include obtaining line-wise difference using `git diff` and encrypting it as well as updating the ciphertext. SGitChar performs better than SGitLine when fewer modifications are made, where the costs for computing the difference and encrypting it in SGitChar are smaller than those of SGitLine, e.g., for repositories [3, 25]. When more modifications are made, i.e., computing word-wise difference costs much more than computing line-wise one, SGitLine performs better in [10, 26, 33] and DocRepo. Git-crypt outperforms, since it does not need to compute the difference. For the same reason, Trivial-enc-sign performs better than our two schemes for small repositories. As the size of the repository and the number of files increase, its advantage disappears.

The costs of generating and verifying an ECDSA signature are 0.93 ms and 0.69 ms, respectively. We directly obtain the MerkelDAG hash value from the output of `git commit`, and thus the costs of signing an update and verifying it are constant. Therefore, we omit these costs in Table 3 and 4.

End-to-end delay. The experiments were conducted on an AWS virtual machine. We measured the round-trip time to the GitHub server using `ping github.com`, and the average latency was approximately 20 milliseconds. Figure 10 shows the end-to-end delay for each repository using different schemes. The end-to-end delay is primarily determined by communication delay, i.e., the time of running `git push`. The time spent on encryption (except for Trivial-enc-sign) and signing operations is negligible in comparison. The communication delay is primarily determined by the size of the transmitted data. According to Table 2, although Git-crypt incurs more communication costs compared to Git, SGitChar, and SGitLine, their communication costs remain at the KB-level. As a result, all schemes share similar communication delays in practice. Generally, SGitLine and SGitChar outperform Trivial-enc-sign, except for a special case, DocRepo.

For DocRepo, each commit is a large revision, and there are multiple changes to each `.tex` file, and computing the character-wise differences takes more time. Due to the small number of files and their small size, encrypting the whole version takes much less time than computing character-wise differences and then encrypting. Therefore, SGitChar needs more end-to-end time than Trivial-enc-sign, even though SGitChar takes less communication delay.

Storage costs. Figure 9 and Figure 11 show the storage costs of the six repositories using SGitLine and SGitChar, Git-crypt [4], and Trivial-enc-sign. Generally, SGitChar performs best among the four schemes. For example, as the storage costs of bootstrap [10] shown in Figure 9(c), the costs of SGitChar and SGitLine are 1.98 MB and 2.06 MB after 50 commits, respectively, which are much smaller than 24.97 MB for Trivial-enc-sign and 4.19 MB for Git-crypt.

In general, SGitChar outperforms SGitLine, while there are two special cases, awesome [3] and FPB [25]. This is because modified lines have multiple changes, which would cause the character-wise delta to be larger than the length of the lines.

SGitChar and SGitLine take fewer local storage costs compared with Git-crypt and Trivial-enc-sign. However, there are two special cases for storing the repository react [33] and FCC [26] in Figure

Table 4: The computation costs of initializing and recovering six repositories under different schemes.

Repository	Initialization (s)				One version Recovery (s)			
	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign
awesome [3]	0.0001	0.0004	0.0001	0.0001	0.0001	0.0048	0.0001	0.0008
FPB [25]	0.0001	0.0005	0.0002	0.0001	0.0002	0.0029	0.0001	0.0058
bootstrap [10]	0.0021	0.0474	0.0033	0.0021	0.0013	0.0322	0.0003	0.0237
react [33]	0.0120	0.4917	0.0235	0.0150	0.0005	0.0140	0.0002	0.0976
FCC [26]	0.0001	0.0007	0.0002	0.0001	0.0021	0.0368	0.0005	1.328
DocRepo	0.0033	0.0717	0.0051	0.0032	0.0007	0.0074	0.0002	0.0030

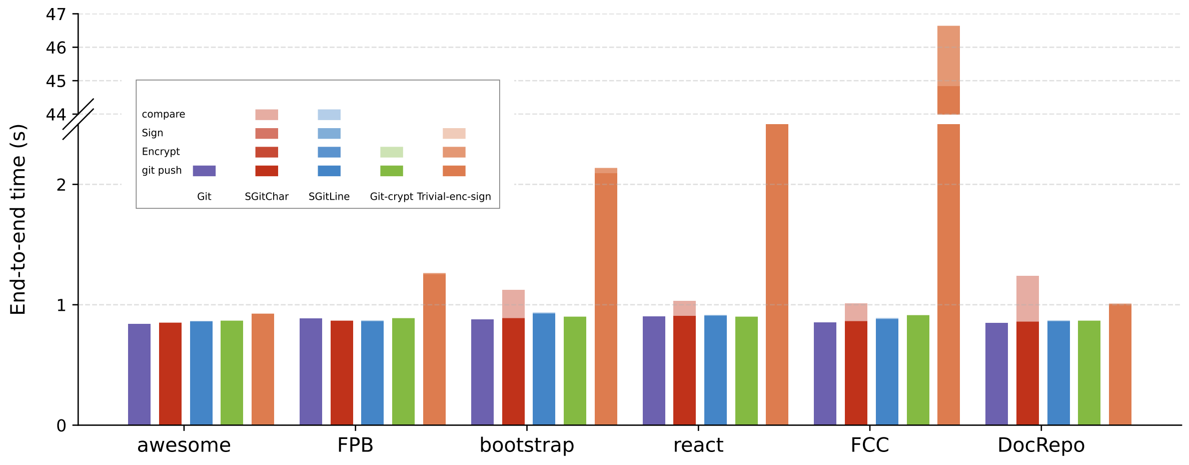
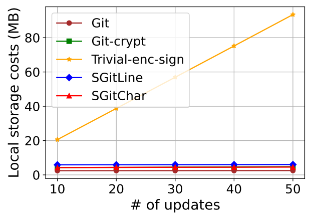
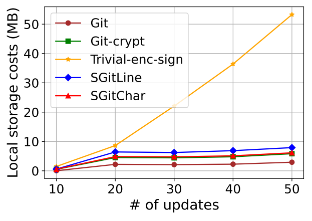


Figure 10: Average end-to-end client time cost of each version update and push to the server under different schemes



(a) Repo react [33]



(b) Repo FCC [26]

Figure 11: Storage costs of repositories using different schemes.

Table 5: The repository information (as of April 2024). The .git folder includes all history versions. The size including the .git folder is the size of all versions, excluding the .git folder is the size of the current version.

Repository	size (MB)		# of files	# of lines
	include .git	exclude .git		
awesome [3]	2.1	0.37	25	2560
FPB [25]	23.2	2.5	217	30690
bootstrap [10]	295.2	20.3	755	174764
react [33]	474.2	30.5	2598	655335
FCC [26]	934.2	451.3	75438	11033103
DocRepo	3.7	2	67	18301

11(a) and 11(b), where SGitLine, SGitChar, and Git-crypt have similar storage costs since these tested versions mainly involve file-wise modifications.

For example, in FCC [26], we can see that Git-crypt takes fewer storage costs than SGitChar. This is because some updates removed a lot of text, which caused the size of the character-wise delta to be larger than that of the file. This result shows that when using SGitChar, if there is a significant version update, such as rewriting a large portion of the files or deleting most of the original content, the user can re-encrypt the updated repository instead of adding incremental ciphertext of the delta. We can also see that Git-crypt takes fewer costs than SGitLine. The reason is that some updates add many new files with lots of lines. For these files, the ciphertext generated by SGitLine is much larger than that generated by Git-crypt. Thus, the storage costs incurred by SGitLine are relatively higher than those of Git-crypt.

Further discussion. We notice that some files cannot be encrypted by line, e.g., images, and the character-wise delta computation method does not apply to these files. Therefore, when implementing SGitLine and SGitChar, we directly encrypt such files for initialization and re-encrypt them if they are modified. GitHub servers would check the format of files uploaded by users and may block the user who tries to upload files with the wrong format. Actually, encryption may destroy the file format, especially for images. To upload the ciphertext to GitHub servers, we use Base64, a binary-to-text encoding, to encode the ciphertext bytes into ASCII characters, since text files have no special format, which may enable the ciphertext to pass the format check. The drawback of this approach is that it results in a 30% increase in ciphertext size. Thus, how to more efficiently upload encrypted files needs further research.

6 CONCLUSION AND OPEN PROBLEMS

This work is the first formal systematic investigation of end to end encrypted Git services. We formalize security properties including confidentiality and integrity to capture real-world vulnerabilities of Git. Moreover, our proposed secure designs are compatible with existing Git servers, making it easy to be augmented. There are many interesting questions to be explored by the community.

Security-wise, our security models capture both the privacy considerations and software supply-chain security. The latter remains underexplored and could be used as a lens to analyze actual security or real-world attacks of products that claim to offer end-to-end security in multi-user setting. Moreover, achieving stronger metadata security, such as hiding users' access patterns and edit behaviors (particularly in systems like Git), remains an interesting and important open problem.

Functionality-wise, our current design focuses on the most critical operations of Git. Many advanced Git features remain unexplored and could be valuable directions for future work. Furthermore, it is important to also investigate how to support more flexible cryptographic group management (which defines access control policies), as well as features such as key rotation, revocation, accountability, and secure integration with web-based Git interfaces.

ACKNOWLEDGMENTS

This work is supported in part by Google via the Digital Future Initiative.

REFERENCES

[1] Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. 2025. Formal Analysis of Multi-device Group Messaging in WhatsApp. In *EUROCRYPT (8) (Lecture Notes in Computer Science, Vol. 15608)*. Springer, 242–271.

[2] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. 2023. Caveat Implementor! Key Recovery Attacks on MEGA. In *EUROCRYPT (5) (Lecture Notes in Computer Science, Vol. 14008)*. Springer, 190–218.

[3] awesome. [n. d.]. https://github.com/sindresorhus/awesome.

[4] Andrew Ayer. 2024. git-crypt - transparent file encryption in git. https://github.com/AGWA/git-crypt.

[5] Matilda Backendal, Hannah Davis, Felix Günther, Miro Haller, and Kenneth G. Paterson. 2024. A Formal Treatment of End-to-End Encrypted Cloud Storage. In *CRYPTO (2) (Lecture Notes in Computer Science, Vol. 14921)*. Springer, 40–74.

[6] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. 2023. MEGA: Malleable Encryption Goes Awry. In *SP*. IEEE, 146–163.

[7] David Balbás, Daniel Collins, and Serge Vaudenay. 2023. Cryptographic Administration for Secure Group Messaging. In *USENIX Security Symposium*. USENIX Association, 1253–1270.

[8] bluss, Joey Hess, and Sean Whitton. 2024. git-remote-gcrypt: a gitremote helper to push and pull from repositories encrypted with GnuPG. https://spwhitton.name/tech/code/git-remote-gcrypt.

[9] Keybase Book. 2024. Security on Keybase. https://book.keybase.io/security.

[10] bootstrap. [n. d.]. https://github.com/twbs/bootstrap.

[11] Enrico Buonanno, Jonathan Katz, and Moti Yung. 2001. Incremental Unforgeable Encryption. In *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers (Lecture Notes in Computer Science, Vol. 2355)*, Mitsuru Matsui (Ed.). Springer, 109–124.

[12] Anrin Chakraborti, Darius Suciu, and Radu Sion. 2023. Wink: Deniable Secure Messaging. In *USENIX Security Symposium*. USENIX Association, 1271–1288.

[13] Bo Chen and Reza Curtmola. 2014. Auditable Version Control Systems. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society.

[14] Long Chen, Yanan Li, and Qiang Tang. 2020. CCA Updatable Encryption Against Malicious Re-encryption Attacks. In *ASIACRYPT (3) (Lecture Notes in Computer Science, Vol. 12493)*. Springer, 590–620.

[15] Long Chen, Ya-Nan Li, Qiang Tang, and Moti Yung. 2022. End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage. In *USENIX Security Symposium*. USENIX Association, 2353–2370.

[16] Weikeng Chen and Raluca Ada Popa. 2020. Metal: A Metadata-Hiding File-Sharing System. In *NDSS*. The Internet Society.

[17] Cas Cremers, Charlie Jacomme, and Aurora Naska. 2023. Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations. In *USENIX Security Symposium*. USENIX Association, 1235–1252.

[18] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. 2023. Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol. In *CRYPTO (4) (Lecture Notes in Computer Science, Vol. 14084)*. Springer, 330–361.

[19] diff-match patch. [n. d.]. https://github.com/google/diff-match-patch.

[20] Github Docs. [n. d.]. About commit signature verification. https://docs.github.com/en/enterprise-cloud@latest/authentication/managing-commit-signature-verification/about-commit-signature-verification. Accessed: 2025-01-09.

[21] Gitea Docs. [n.d.]. GPG Commit Signatures, Version: 1.22.6. https://docs.gitea.com/administration/signing. Accessed: 2025-01-09.

[22] IPFS Docs. [n.d.]. Merkle Directed Acyclic Graphs (DAGs). https://docs.ipfs.tech/concepts/merkle-dag/. Accessed: 2025-06-26.

[23] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. 2017. Key Rotation for Authenticated Encryption. In *CRYPTO (3) (Lecture Notes in Computer Science, Vol. 10403)*. Springer, 98–129.

[24] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. 2023. Injection Attacks Against End-to-End Encrypted Applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 82–82.

[25] free-programming books. [n. d.]. https://github.com/EbookFoundation/free-programming-books.

[26] freeCodeCamp. [n. d.]. https://github.com/freeCodeCamp/freeCodeCamp.

[27] GitLab.com. [n. d.]. Signed commits. https://docs.gitlab.com/ee/user/project/repository/signed_commits/. Accessed: 2025-01-09.

[28] Jonas Hofmann and Kien Tuong Truong. 2024. End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem. In *CCS*. ACM, 3988–4001.

[29] Joseph Jaeger and Akshaya Kumar. 2025. Analyzing Group Chat Encryption in MLS, Session, Signal, and Matrix. In *EUROCRYPT (8) (Lecture Notes in Computer Science, Vol. 15608)*. Springer, 272–301.

[30] Joseph Jaeger, Akshaya Kumar, and Igors Stepanovs. 2024. Symmetric Signcryption and E2EE Group Messaging in Keybase. In *EUROCRYPT (3) (Lecture Notes in Computer Science, Vol. 14653)*. Springer, 283–312.

[31] Kenneth G. Paterson, Matteo Scarlata, and Kien T. Truong. 2023. Three Lessons From Threema: Analysis of a Secure Messenger. In *USENIX Security Symposium*. USENIX Association, 1289–1306.

[32] Proton. [n. d.]. Protocol Drive. https://proton.me/drive.

[33] react. [n. d.]. https://github.com/facebook/react.

[34] Nikita Sobolev. 2024. git-secret: A bash-tool to store your private data inside a git repository. https://github.com/sobolevn/git-secret.

[35] Bitbucket support. [n. d.]. Verify commit signatures. https://confluence.atlassian.com/bitbucketserver/verify-commit-signatures-1279066267.html. Accessed: 2025-01-09.

[36] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. 2015. SoK: Secure Messaging. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 232–249.

[37] Wenhan Xu, Hui Ma, Zishuai Song, Jianhao Li, and Rui Zhang. 2024. Gringotts: An Encrypted Version Control System With Less Trust on Servers. *IEEE Trans. Dependable Secur. Comput.* 21, 2 (2024), 668–684.

[38] Xin Xu, Quanwei Cai, Jingqiang Lin, Shiran Pan, and Liangqin Ren. 2019. Enforcing Access Control in Distributed Version Control Systems. In *IEEE International Conference on Multimedia and Expo, ICME 2019, Shanghai, China, July 8-12, 2019*. IEEE, 772–777.