UNIVERSITÄT BASEL

SSH-based Access Control for Git Using Git

Bachelor's thesis

Natural Science Faculty of the University of Basel
Department of Computer Science
Computer Networks Group
cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Pius Lukas Walser
pius.walser@stud.unibas.ch

17.06.2025

Acknowledgments

I would like to thank Prof. Dr. Christian Tschudin and Dr. Erick Lavoie for allowing me work on this thesis and their support.

I would especially like to express my gratitude to Dr. Erick Lavoie for his feedback during the process of writing this thesis. His expertise in this field and mentorship guided me while writing this thesis.

Finally I would like to thank Jörg Ammann for his feedback on an earlier version of this thesis.

ChatGPT was used to look up syntax and usage patterns for libraries. It was not used to write any of the text.

Abstract

Administrating a Git repository can present a certain workload. We present a way to manage a repository using commits to control access. A user can express his trust to an other user. This creates a trust graph which is the base for calculating who can access the repository on which way. The owner of a repository can grant read and write access while users with this access can grant read access to other users. This is done in a secure way by using self-certification. Incoming commits are processed by a Git hook and saved in a trust graph. This trust graph is then translated into access privileges. These privileges are enforced by restricting the SSH access of users to certain actions. We provide a TLA+ specification that clearly defines the behavior of the system and show that the system implements the specification.

Table of Contents

1 Introduction

Using a Git repository[1] is a common occurrence when developing software. It allows developers to concurrently modify the same source files and is a powerful tool for version control. It may also be used to backup files to a different device which is best practice to ensure those files can be recovered from a different source in case of failures.

Platforms hosting Git repositories, such as GitHub and GitLab, are very commonly used, because they streamline the exchange of updates between repositories by keeping the canonical version of a repository on a highly available server. These two tools provide many features that are not part of Git itself [1, 2] such as continuous integration and deployment tools or issue trackers as well as free hosting within certain limitations [1]. Issues might counter-balance the convenience that such platforms offer. Platforms owners [3] might at any time change their data policies and start using the data produced by users for new purposes, such as training Large Language Models [4], without explicit consent.[2] In reaction, some users may simply not want their code in the hands of a major tech company.

If one does not want one's code to be on servers owned by a big corporation or used to train large language models there is the possibility of hosting an instance of GitLab. For an individual maintaining a GitLab instance, this requires significant maintenance work [5] because the platform provides numerous features related to roles, permissions, in-browser editing, etc. [2] that are unnecessary, at the level of an individual or a small team, i.e. when collaborating on a single repository with a limited number of collaborators and local tools.

A third alternative, is to have a repository on a server and controlling access through SSH. When doing so, the public key of every participant has to be added to the `authorized_keys` file on the server that hosts the repository. Unless careful, this can give full SSH access to the server to all authorized. The administrator of the server also has to add and remove keys manually.

Unrestricted SSH access is rarely needed when collaborating on a Git project and poses a security risk if one does not trust the users fully that have this type of access. Reducing

[1] The words "repository" and "replica" are used interchangeably in this thesis and refer to the same concept.

[2] Hosting platforms provide access control, allowing one to make their repository private and selectively offer access. Nonetheless, if a repository is public on GitHub it will be used to train large language models [4].

the workload of the administrator of such a repository is also important since the time they spend on administrating the repository can be used for more useful things.

As a group of collaborators grows, the management burden can quickly grow as well as well as the vulnerability surface, if not careful. For example, suppose Alice is managing a repository and Bob is collaborating with her. This means she now has to add his SSH key manually and lets suppose she is careful and restricts his access to Git. Now imagine if Bob wanted to show the code to his friend Carol for some help with debugging. He would need to tell Alice to add Carol and Carol, who Alice might not know, can edit all the files in their project or delete them. In general, this means that all permission modifications have to be managed by the server administrator, which may represent a significant workload.

This is where the software developed in this Thesis comes in. The aim is to simplify the hosting and managing of a Git repository using authenticated commits to manage access. This is done by using Git hooks, a few scripts and self certifying commits. If a developer hosts a repository they designate a public key and the user with the corresponding private key will be treated as the owner of the server.

Using a script any user can create self certifying commits that express that they trust another public key. After pushing these commits to the server the user holding the private key that corresponds to this public key will now have push and pull privileges on the server if the commit was created by the owner. If the commit was created by a user trusted by the owner, these users will only gain pull capabilities but will not be able to push commits to the server. A user can also create commits that express that they do not trust a user anymore, which will remove any privileges granted to this user previously.

1.1 Contributions

We provide an easy and safe way of managing access to a shared Git repository. In addition, we wrote and verified by model-checking a clear TLA+ specification of the software which provides a very precise specification of the behavior of the software. Since it works only by using features provided by Git and SSH which are needed anyway to share a repository there is minimal overhead except for an installation of Python. Building on top of these widely used technologies in combination with self-certification provides strong security properties and ensures only trusted users can modify the access to the repository. It is also easy to use because: 1) a server installation has less dependencies and configuration options than platforms such as GitLab; and 2) it is not necessary to log into a user account on the server to manage access permissions.

1.2 Outline

In this Thesis, we first introduce important methods and concepts (chapter 2), then present a TLA model of the proposed solution (chapter 3) and talk about the implementation (chapter 4). This is followed by an evaluation (chapter 5), a review of related work (chapter 6) and a summary of the thesis as well as an overview of future work (chapter 7).

2 Background

2.1 Git

Git is a version control system based on commits. A commit encapsulates all changes made to certain files and has all the meta data listed in Table 2.1. The actual changes to the files are stored in a Git tree object which is then referenced by the commit [6, pp. 417–423]. Each commit has checksum or commit hash which is what we will call it [6, p. 16] and is linked to any amount of previous commits which are its parents [6, p. 63]. Because of the link to previous commits a causal ordering of the commits can be created.

Table 2.1: Fields of a Git commit [7, 8]

Field
Commit hash
Commit message
Tree hash
Parent hashes
Author name
Author email
Author date
Committer name
Committer email
Committer date
Signature

Git lets you create branches which are just a pointer to a certain commit [6, pp. 63–66]. These are stored in `.git/refs/heads/<branch_name>` [6, pp. 425–426]. It also has hooks where custom functionality can be added that is triggered if certain events occur [9].

2.2 Secure Shell

You can log into a server securely over an unsecure network using Secure Shell (SSH). One can authenticate oneself using host-based authentication, public key authentication, challenge-response authentication, or password authentication [10]. For public key authentication a user needs a key pair consisting of a public and private key. It allows the user to authenticate

itself to a server by uploading the public key to the server before using it for authentication. After the symmetrical encryption is established to secure communication between the server and the client, the client must be authenticated to be allowed access. The server uses the public key to encrypt a challenge message to the client. If the client can prove that it was able to decrypt this message, it has demonstrated that it owns the associated private key [11]. The user can then log in.

2.3 Self-Certification

Self certification works by creating a file that contains a public key and a signature made with the corresponding private key [12]. This enables anyone to certify that the file was signed by the user in possession of this private key and therefore proving its authenticity. This can be implemented with Git using a field of a Git commit to store the public key and signing the commit with the corresponding private key [13].

2.4 TLA+

TLA+ is a high-level state-based modelling language for modelling distributed concurrent systems. It uses mathematical methods to model these and provides a model checker to ensure the model is correct. This high-level view lets you write a model while abstracting away any details about how it should be implemented. Therefore it enables engineers to ensure their model is accurate before implementing it. This again makes implementation easier because one already has a model [14]. All TLA+ notation will be explained as needed in the following Chapter.

3 Model

The design is based on an authenticated and eventually-consistent replicated trust graph. Therefore, unless careful at the design stage, the resulting implementation may exhibit surprising and undesired behavior, during the temporary phases of inconsistencies. To ensure the correctness of the system, the replicated trust graph was modeled using TLA+. This allows us to precisely define the expected behavior and enables the verification of the system's properties using the TLC model checker. The full TLA+ specification can be found in Appendix A.

We first explain how the system should work to give an overview and then explain all parts of the TLA+ specification by starting with the initialization, going over the actions, the helper functions, the invariants and ending with the liveness properties.

3.1 Transitive Capabilities with a Replicated Trust Graph

There are three different levels of access that can be granted by this software. One is the owner of the repository and the person that has full SSH access to the server. If the owner now adds trust to Person A, Person A will be able to push new commits to the server and pull other commits from the server. If Person A now adds trust for Person B, who is not trusted by the owner directly, Person B will only be able to pull commits from the server but will not be able to push their own commits to the server.

At a high-level, we implement those capabilities by having each participant signal modification of their trust towards another participant in operations, which we call *commits*, sequentially. Commits are combined in a data structure that only the committer may modify, which we call an *access control branch*. Access control branches are replicated on all participants, which can then use them to compute the latest state of the trust graph and modify permissions accordingly.

3.2 Initialization

The model has a certain number of clients. It contains a trust graph, with one replica for each client. In their own replica, each client stores the state of the trust graph that they are aware of. This is from the perspective of the server owned by this client. They store who they trusts and who all other clients trust. Each client can always only modify who is trusted by them in their own replica. All this information is stored in the variable TG.

In TLA+, this is modelled as nested mappings from client to client, ending in a tuple of a boolean and a natural number representing respectively whether there is trust and how many times the trust relationship was modified. At the first level, the mapping uses a client r as a key and specifies the local replica for r, with the corresponding value representing the current state of the replicated trust graph for r. At the second level of nesting, the mapping uses a client $c1$ as key and specifies whose trust relationship we are currently looking at. At the third level of nesting, the mapping uses a client $c2$ and specifies the target of the trust relationship.

This is followed by a tuple that contains a boolean and a natural number. The boolean specifies if that third client is currently trusted by the second client or not. The natural number specifies how many operations have been performed on this entry. For each $AddTrust$ or $RemoveTrust$ operation it is increased by one. This enables us to tell which entry is newer, since only $c1$ can modify this entry and we assume all operations of $c1$ are sequential. For our initialization, we set all booleans to FALSE and all our natural numbers to 0. We create this mapping for all clients that exist in the model. This can be seen in Figure 3.1. Here $x \in Set \mapsto Elements$ means all x in Set map to all $Elements$ [15, pp. 301–304]. Therefore there is a mapping from every x in Set to every $Element$.

$$
\begin{aligned}
Init \triangleq\ & TG = [r \in Clients \mapsto \\
& \quad [c1 \in Clients \mapsto \\
& \quad\ [trusts \mapsto \\
& \quad\quad [c2 \in Clients \mapsto \\
& \quad\quad\ \langle \text{FALSE}, 0 \rangle]]]]
\end{aligned}
$$

Figure 3.1: Initialization of the TLA+ model

3.3 User Operations as TLA+ Actions

A client can only update whom they trust from their own replica first. They can add or remove their trust to another client or they can merge the replica of another client onto their own replica. This corresponds to three TLA+ actions in the model:

- $AddTrust$ - A client adds trust from itself to another client.
- $RemoveTrust$ - A client removes trust from itself to another client.
- $Merge$ - A client merges the trust graph of another client onto their own replica.

In TLA+ there are two things that an action has. One of which is the precondition which enables the action and the other is the effect of the action. An Action is applied if all statements it contains are true [15, pp. 312–314].

You can combine these statements in TLA+ by using a bullet point in front of each statement. We are using $\land$ (conjunction) but $\lor$ (disjunction) is also allowed. The logical operator used for the bullet points decides how the statements are logically combined [15, p. 293].

These actions provide all the functionality needed to model the system. Merging enables the propagation of trust. This is important because if a client $c1$, who already has push-pull capabilities, trusts any other client $c2$, then $c2$ should obtain pull capabilities. In a real world environment where there are several servers, the updates can also be propagated transitively without requiring that the client that added or removed trust to be directly involved.

3.3.1 Adding and Removing Trust

There are only two ways of modifying trust relationships, either by adding or removing trust. We only explain the action for adding trust in detail since the action for removing trust works symmetrically.

The action *AddTrust* in Figure 3.2 chooses two arbitrary clients which is done with $\exists\, c1, c2 \in \mathit{Clients}$ [15, pp. 314–316]. Here $c1$ does not trust $c2$ is the precondition which is the first bullet point. This is simply done by checking in the replica of $c1$ if the boolean value of the tuple that encodes the relationship from $c1$ to $c2$ is FALSE.

The second bullet point specifies the effect of the action. Here TG' signifies what the trust graph should be after the action has been applied. We specify that it is supposed to be the same except for the tuple that encodes the relationship from $c1$ to $c2$ in the replica of $c1$. The keywords TG EXCEPT ! allow us to specify which entry in TG is supposed to be different while leaving everything else untouched [15, pp. 305–306]. We then set the boolean value of the tuple to TRUE and increment the counter by 1.

$$
\begin{array}{l}
\mathit{AddTrust} \triangleq \exists\, c1,\, c2 \in \mathit{Clients} : \\
\quad \land\ TG[c1][c1].\mathit{trusts}[c2][1] = \textsc{false} \\
\quad \land\ TG' = [TG \ \textsc{except}\ ![c1][c1].\mathit{trusts}[c2] = \langle \textsc{true},\ TG[c1][c1].\mathit{trusts}[c2][2] + 1 \rangle]
\end{array}
$$

Figure 3.2: Action *AddTrust* from the TLA+ specification

The action *RemoveTrust* works the exact same way except that $c1$ has to already trust $c2$ and it sets the trust value to FALSE.

3.3.2 Merging

Similar to *AddTrust* or *RemoveTrust*, *Merge* in Figure 3.3 starts by choosing two arbitrary clients $c1$ and $c2$. The replica of $c1$ cannot be the same as the replica of $c2$ for this action to be enabled. It then updates the replica of $c1$ with anything in the replica of $c2$ that is newer. This is because we are again using the notation $x \in \mathit{Set} \mapsto \ldots$ therefore doing this for all elements in the set. Which is newer is indicated by the number in each entry. If it is higher the entry is newer. This is done by the IF ... THEN ... ELSE statement that works like in any other programming language [15, p. 298].

3.4 Helper Functions

One helper function that was created is $\mathit{CanPull}(c, r)$ in Figure 3.4 which determines if client c can pull from the replica of client r given the state of the trust graph currently

$$
\begin{array}{l}
Merge \triangleq \exists\, c1,\, c2 \in Clients : \\
\quad \land\, TG[c1] \neq TG[c2] \\
\quad \land\, TG' = [TG \text{ EXCEPT } ![c1] = \\
\qquad\qquad [c3 \in Clients \mapsto \\
\qquad\qquad\quad [trusts \mapsto [c4 \in Clients \mapsto \\
\qquad\qquad\qquad \text{IF } TG[c1][c3].trusts[c4][2] < TG[c2][c3].trusts[c4][2] \\
\qquad\qquad\qquad\quad \text{THEN } TG[c2][c3].trusts[c4] \\
\qquad\qquad\qquad\quad \text{ELSE } TG[c1][c3].trusts[c4]]]]]
\end{array}
$$

Figure 3.3: Action $Merge$ from the TLA+ specification

replicated by r. This is the case if either $c = r$, r directly trusts c or if there exists a client $c2$ that trusts c and is trusted by r.

$$
\begin{array}{l}
CanPull(c,\, r) \triangleq \\
\quad \land\, c \in Clients \\
\quad \land\, r \in Clients \\
\quad \land\, \lor\, c = r \\
\qquad \lor\, TG[r][r].trusts[c][1] \\
\qquad \lor \exists\, c2 \in Clients : \\
\qquad\quad \land\, TG[r][c2].trusts[c][1] \\
\qquad\quad \land\, TG[r][r].trusts[c2][1]
\end{array}
$$

Figure 3.4: Helper Function $CanPull(c, r)$ from the TLA+ specification

There is also a helper function $CanPush(c, r)$ but it is a bit simpler since only the owner of a replica and the people directly trusted by him can push to a replica. It is not used in the model but it is included for completeness. Therefore we will not discuss it further.

3.5 Invariants

3.5.1 Typing

Since in TLA+ variables are untyped we need to specify our own invariant that specifies the set of values that are allowed for each variable [15, p. 67]. It is very similar as what was already done in the initialization except this time we are not assigning any values but we are checking if TG is an element of the set it may be in. Namely being a mapping from $Clients$ to $Clients$ to the label $trusts$ to $Clients$ to a tuple of a boolean and a natural number [15, pp. 306–307]. This is done in Figure 3.5. $\rightarrow$ signifies that it is that set of functions with these types. For example $[S \rightarrow T]$ is the set of functions f with $f(x) \in T$ for $x \in S$ [15, pp. 301–304].

$$TypeOK \triangleq TG \in [Clients \rightarrow [Clients \rightarrow [trusts : [Clients \rightarrow \text{BOOLEAN} \times Nat]]]]$$

Figure 3.5: Invariant $TypeOK$ from the TLA+ specification

3.5.2 Local Modifications First

We also have another invariant in Figure 3.6 to make sure that a client only modifies the trust values in his own replica first before they are propagated to replicas of other clients. This is done by comparing the iteration numbers between two replicas. The entry either needs to be completely the same, which means the other replica is up-to-date, or the revision number

of the remote replica should be lower, which means it has not replicated the latest state yet. Here we are now using bullet-pointed $\vee$ statements in contrast to the bullet-pointed $\wedge$ statements from before. This has always to be true for any three arbitrary clients which in TLA+ can be written with the $\forall$ operator [15, pp. 293–294].

$$\begin{aligned} &\mathit{LocalTrustFirst} \triangleq \forall\, c1,\, c2,\, c3 \in \mathit{Clients} : \\ &\quad \vee\, TG[c1][c1].\mathit{trusts}[c3] = TG[c2][c1].\mathit{trusts}[c3] \\ &\quad \vee\, TG[c1][c1].\mathit{trusts}[c3][2] > TG[c2][c1].\mathit{trusts}[c3][2] \end{aligned}$$

Figure 3.6: Invariant $\mathit{LocalTrustFirst}$ from the TLA+ specification

3.6 Liveness

We specify some liveness properties for eventual consistency. The first formula in Figure 3.7 states that for two arbitrary clients $c1$ and $c2$, and for every client $c3$, if $c1$ locally trusts $c2$ and $c3$ locally trusts $c1$, then it follows that $c2$ can pull from $c3$ which in TLA+ can be expressed with the implication ($\Rightarrow$) operator [15, pp. 293–294].

This may not always be the case. If the change in $c1$'s replica has not been propagated to $c3$ yet, the first part may be true but not the second part. Because of this we specify that it has to be eventually true and can infinitely often switch between being false or true. This can be expressed by the symbols $\Box\Diamond$ [15, pp. 314–316].

$$\begin{aligned} &\mathit{TransCanPull} \triangleq \exists\, c1,\, c2 \in \mathit{Clients} : \\ &\qquad\qquad \forall\, c3 \in \mathit{Clients} : \\ &\qquad\qquad\quad (TG[c3][c3].\mathit{trusts}[c1][1] \wedge TG[c1][c1].\mathit{trusts}[c2][1]) \\ &\qquad\qquad\quad \Rightarrow \mathit{CanPull}(c2,\, c3) \\ &\mathit{InfOftenTransCanPull} \triangleq \Box\Diamond \mathit{TransCanPull} \end{aligned}$$

Figure 3.7: Liveness property $\mathit{InfOftenTransCanPull}$ from the TLA+ specification

There is also a second liveness property in Figure 3.8 that specifies the same concept but when trust is removed instead. Here we need to specify this a bit differently. We start by saying that there exists a client $c2$ and for all clients $c1$ we specify that $c1$ may not trust $c2$ locally and they may not be the same. Then we specify that there may not be any transitive trust through any client $c3$ locally on $c1$'s and $c3$'s replica respectively. If all these conditions hold we imply that $c2$ can not pull from $c1$. The inverse of a statement can simply be expressed by using the $\neg$ operator in front of the statement [15, pp. 293–294].

We then specify that it can switch infinitely often between being false or true because of the same reasons as before.

$$\begin{aligned} &\mathit{TransCanNotPull} \triangleq \exists\, c2 \in \mathit{Clients} : \\ &\qquad\qquad \forall\, c1 \in \mathit{Clients} : \\ &\qquad\qquad\quad (\wedge \neg TG[c1][c1].\mathit{trusts}[c2][1] \\ &\qquad\qquad\quad\ \wedge c1 \neq c2 \\ &\qquad\qquad\quad\ \wedge \forall\, c3 \in \mathit{Clients} : \\ &\qquad\qquad\quad\qquad \vee \neg TG[c1][c1].\mathit{trusts}[c3][1] \\ &\qquad\qquad\quad\qquad \vee \neg TG[c3][c3].\mathit{trusts}[c2][1]) \\ &\qquad\qquad\quad \Rightarrow \neg \mathit{CanPull}(c2,\, c1) \\ &\mathit{InfOftenTransCanNotPull} \triangleq \Box\Diamond \mathit{TransCanNotPull} \end{aligned}$$

Figure 3.8: Liveness property $\mathit{InfOftenTransNotCanPull}$ from the TLA+ specification

4

Implementation

This chapter is about enforcing the security properties that were implied in the model. This is done by using self-certification and a pipeline of verification steps to ensure that only valid operations may modify the state of the repository. These properties are: 1) the owner always has capabilities; 2) only those who can push or pull on the model may do so in the implementation; and 3) each participant can only modify their own trust relations.

It is important to note that the owner of a replica needs to be trusted since he has full control over the machine and could modify all trust relations. This is fine since this software aims to let the owner delegate work and not make him relinquish power over his machine.

4.1 User Perspective

Adding or removing trust in this software is encoded in Git commits. Each user has their own branch on which they make commits that either encode adding or removing trust. We will call these their *personal access control* branches. Each branch's name follows the structure `<root-hash>_<base-32-encoded-pub-key-of-user>`.

The root hash is the Git hash of a signed Git commit that contains all software needed to run the access control software and we will refer to it as the *root commit*. This allows us to identify which version of the software is being used and if this branch even belongs to this access control software.

The base 32 encoded pubkey allows us to tell exactly to whom this branch belongs to. It is base 32 encoded to avoid any characters that could be be invalid or be interpreted by Git or the operating system as a path separator, such as a backslash.

4.1.1 User Operations

A user of the system can simply interact with the system by interacting with the `gac.py` [16] script that will be on their personal access control branch as well as on the branch called `<root-hash>_gac` which is the branch that contains all software used for the access control. How to setup a repository to work with the software will be discussed later in subsection 4.5.3. If a user wants to add trust for a person he can do so simply by executing the following command:

```
./gac.py add_trust <trustee-pub-key>
```

This corresponds directly to the *AddTrust* action discussed in the model earlier. The following command corresponds to the *RemoveTrust* action:

```
./gac.py remove_trust <trustee-pub-key>
```

The *Merge* action corresponds to pushing the changes on the local machine to the server that hosts the repository. This will merge the pushed replica with the replica of the owner of the server. If the pushed replica belongs to the owner of the server it updates the server to the current state of the replica.

4.2 Commit Structure

Each commit on an access control branch encodes three things. The commit message is either `add_trust` or `remove_trust` which encodes the operation. A git commit also has fields for the name of the committer and their email. These are rarely used so in this software we will use them to encode some other things. The field for the committer name is used to encode the public key of the person for whom we want to add or remove trust. The field for the committer email contains the base 32 encoded pubkey of the person executing this trust operation. We are using the base 32 encoded pubkey here for consistency reasons.

But in Git anybody can create a commit on any branch they want. This is why we are using self certifying commits. By signing the commits with the same key that is encoded in the branch name and the committer email we can verify that the user that made a commit on this branch is the user this branch belongs to.

For a branch to be a valid access control branch it also needs to have the root commit as the root of the branch. By checking for this we can again verify if this branch belongs to the access control software and the current version of the software.

To enable us to have a clear understanding of what trust operations happened, and in which order, we require each access control branch to be a linked list (sequence) of commits. This means there can be no merge commits.

We do not need to be able to tell when an operation happened relative to an other access control branch because we are only interested in who is trusted by each user at the current state. However the order of trust operations that each user executed are important for us. Since the commits that encode these trust operations can be created without any prerequisites it is entirely possible for a user to first remove trust for a user that he did not trust before and then add it again. If these operations are in the reversed order the outcome would be the opposite of what is supposed to happen. Therefore it is imperative for us to have a linear timeline that cannot be altered and we need to disallow deleting these branches or commits on these branches.

4.3 Architecture

There are 3 main parts to the architecture of this software, which are illustrated in Figure 4.1.

The first is the script `gac.py` [16] which is mainly there to make interacting with the software easier for the user. All things done by this part of the software could also be done by the user by typing the corresponding Git commands and creating certain files. It would however

make interacting with software rather clunky. The second part is comprised of what we write into the `authorized_keys` file and the two scripts `git_pull.sh` and git_push_pull.sh [16]. This part is responsible for limiting the capabilities of users added by the system. The third part is the git hook `pre-receive.py` [16] that is responsible for most of the logic of the software.

If a user pushes commits to a server there are usually three checks if this is allowed to happen. First, if they connect using SSH and their public key is not in the `authorized_keys` file, the push will be rejected. Second, the command sent to the server to be executed is checked by either `git_pull.sh` or `git_push_pull.sh` [16] and may be rejected if it is not allowed. There is one exception to this rule and that is if the owner of the server pushes any changes: in this case, it will skip these scripts since the owner is given full SSH access maintain the server anyway. Third, the `pre-receive.py` [16] Git hook will check the cryptographic properties of all incoming commits and reject them if they are violated. If the commits pass all these checks the changes they make to the trust graph on the server are applied.

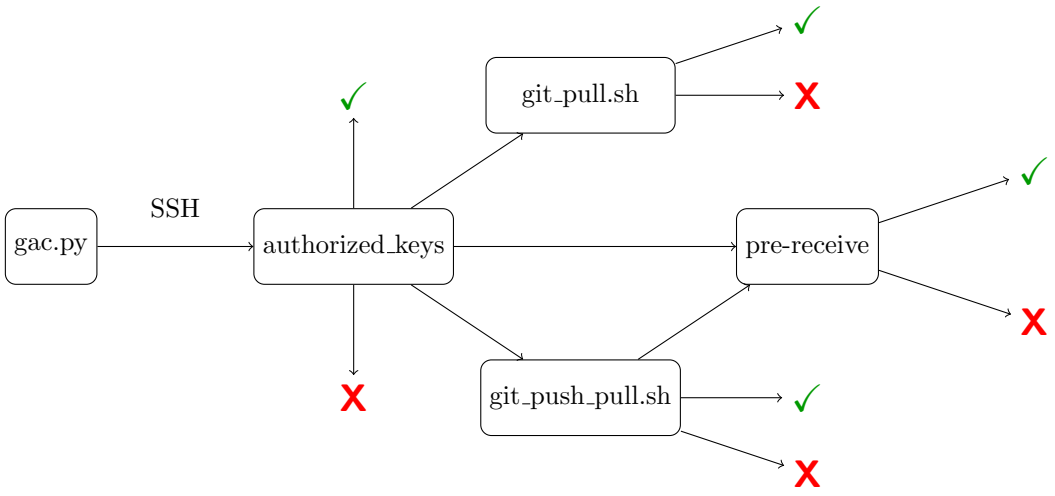

Figure 4.1: Architecture of the Git access control software

4.4 Code

In this section, we explain in more detail how each component of the architecture is implemented.

4.4.1 Restricting SSH Access with the Authorized Key Options

It is possible to restrict users that access a server using SSH with a public key in a few different ways. It is possible to disable many features of SSH such as port forwarding or X11 forwarding by adding certain arguments to the entry in the `authorized_keys` file [6, pp. 113–116]. In our case this is:

```
no-agent-forwarding,no-port-forwarding,no-pty,no-user-rc,no-X11-forwarding
```

These arguments disable all other features of SSH that a user could use to make sure they only execute the commands that they are supposed to. It is also possible to specify a

command to be executed when the user starts an SSH session by simply adding this [17] in front of the other arguments:

```
command="<command-to-be-executed>"
```

This allows us to restrict any user further. We either execute the `git_pull.sh` or the `git_push_pull.sh` script [16]. These scripts get passed the original SSH command and then check if it matches a certain pattern. This is possible because `git pull` invokes `git-upload-pack` and `git push` invokes `git-receive-pack` for a specific repository on the server side. The patterns get modified when a server is created so that they only work for one repository. If the pattern matches we execute the command in a git shell. This shell only allows git commands which further improves security.

Unfortunately one can only run one repository per user per server at a time. This is due to how SSH reads the public keys in the `authorized_keys` file. It always applies the first key that matches and if that is not the entry that belongs to the repository a user wants to access the access control system will deny access. This is a limitation of SSH and making it work with multiple repositories is not part of this thesis. If one wants to run multiple repositories one can either run each repository on a different user or run them in Docker containers.

4.4.2 Git Hook

The pre-receive git hook is where all the logic of this software resides and it can be found on Github and is called `pre-receive.py` [16]. This is the hook that gets invoked when a repository receives new commit(s) through a user invoking `git push` on their machine [9]. The access control software will be run on a server which only receives new commits through push operations made by clients. This means we only need to process commits received this way.

This script receives the hash of the old revision, the hash of the new revision and the reference name on standard input. Firstly it loads the current trust graph from disk. It then checks if it is an access control branch by looking for the root commit. If it does not find it, it will just continue with the next branch that was pushed to the server. It then checks if the branch is named correctly and if it is not the branch called `<root-hash>_gac` which may not be updated since it contains the access control software. It then checks for deletion, force pushes and merges which are all things we do not allow. If these checks all pass it starts processing commits one at a time. It checks the cryptographic properties of each commit on an access control branch and correctly modifies the trust graph.

After processing all commits on all branches that were pushed to the server, the script saves the trust graph to the disk, resolves the trust graph into user-specific privileges and modifies the `authorized_keys` file accordingly.

The trust graph is saved as two nested lists. The outer list contains an entry for each user and an inner list of the users trusted by this user. Parsing the trust graph to privileges is done by adding all keys trusted by the owner to the list of users that may push and pull. After this it goes through all keys of users on that list and adds the keys that are not the owner or in that list to a list of people that may only pull.

4.4.3 Installation

The software is in a commit and we use this commit to identify branches of the software. The cleanest way to install this software is therefore to insert this commit into a Git repository and check it out. This is what our setup script `gac.sh` [16] does. It contains a base 64 encoded bundle which contains this single commit. This commit is inserted into a new branch called `<root-hash>_gac`.

The commit hash of this commit is essential for the software to work since we need to check for this root commit and the easiest way to do this is to check the commit hash. This creates a problem because the hash of the root commit has to be in the input for the hash function that creates this hash. Therefore the hash depends on itself. This means we need to patch the commit hash into the software while installing it since there is no way to create a commit that contains its own hash.

After patching the commit hash into the software `gac.sh` calls `gac.py` [16] with the operation `create_server` and the first argument it was given. This argument should be the public key of the owner of the repository. `gac.py` now changes a few lines in `git-pull.sh` and `git-push-pull.sh` [16] so that they only allow access to the specific repository that is currently being setup and log their output to a file in the `.git` folder of this repository. After all these modifications are done all files get moved to the places they need be. `git-pull.sh` and `git-push-pull.sh` get moved to the `.git` folder and `pre-receive.py` gets moved to the `.git/hooks` folder as well as renamed to `pre-receive`. Having made all the changes necessary the script now commits the changes to the `<root-hash>_gac` branch and the server is ready.

4.5 Deployment

This section describes how to use the code that was developed in this thesis to host a Git repository with access control. It will provide two different ways to run it. One which allows a developer to easily make modifications on his development machine, which we call the *host machine*, and one in a Docker container to simplify deployments on remote servers.

4.5.1 Deploying the Code on the Host Machine

Deploying the code on the host machine itself is the simplest way to use it. One needs to have a non bare initialized Git repository and just needs to copy the `gac.sh` [16] file into the root folder of the repository. Then one needs to run it with the following command, replacing `<ed25519-pub-key>` with the ED25519 public key that one wants to be the owner of the repository:

```
./gac.sh <ed25519-pub-key>
```

Make sure that this public key is also in the authorized keys file with full SSH access to the server. This can simply be done by adding the following line to your `authorized_keys` file:

```
ssh-ed25519 <ed25519-pub-key> <optional-identifier>
```

The identifier is completely optional and can just be removed.

4.5.2 Deploying the Code in Docker

Deploying the code in Docker is a clean and simple solution to host a repository but provides a few challenges. Since the code is not being run on the host machine itself but in a container there still needs to be a way to access the host machine.

One can not expose port 22 of the container to the outside since this port is needed to access the host machine. It is also not possible for the host machine to tell which hostname was used to access it since there is no protocol extension like Server Name Identification (SNI) in HTTPS for SSH as discussed in a Stack Overflow post [18]. Otherwise it would be possible to tell by the hostname that was used if a user wanted to access the container running the Git repository or if he wanted to use SSH to access the host machine. This is a problem because it would be the cleanest solution to access the repository. We can get around this by exposing a different port to the outside and then mapping this port to port 22 on the container. This setup keeps the host machine accessible and allows access to the Git repository inside the container. However, since the SSH port is non-standard, the port must be explicitly specified when accessing the repository.

If one wants to run the container Docker must be installed and one needs to edit the last line in the Dockerfile to use the ED25519 public key that should be the owner of the repository. Currently it contains an example key. It should look like this:

```
ENTRYPOINT ["/entrypoint.sh", "<your-key>"]
```

The example Dockerfile that is included maps port 22 of the container to port 2222 of the host machine. One can build the docker image by executing the following command in the directory that contains the `Dockerfile`, `docker-compose.yml`, `gac.sh` and `entrypoint.sh` [16] files:

```
docker-compose build
```

One can then run the container in the background by executing the following command in the same directory:

```
docker-compose up -d
```

To stop the container one can execute the following command in the same directory:

```
docker-compose down
```

If one wants to run multiple containers one can do so by exposing a different port for each container. This would only require changing the port mapping in the `docker-compose.yml` file. In the example `docker-compose.yml` port 22 on the container is mapped to port 2222 on the the host machine. One can change this to any port that is not already in use. One will also need to change the names of the volumes in the `docker-compose.yml` file to something that is unique for each container. Otherwise the container will just contain the same data as the other container.

4.5.3 Accessing the Repository

To access the repository one can use the following command, replacing the placeholders as needed:

```
git clone ssh://<user>@<host>:<port><path-to-repository>.git
```

If one is running the code in Docker with the example Dockerfile the command will look like this and one only needs to replace `<host>` with the hostname or IP address of the host machine:

```
git clone ssh://git@<host>:2222/home/git/repo/.git
```

This command will clone the repository to the machine it is being executed on. To start using the access control system one needs to set it up first. This can be done by executing a single command in the root directory of the repository right after cloning it. This command will setup the access control system on the local machine. One will need to provide the path to one's own ED25519 public key. This key must be the same that was used to create the repository on the host machine if one is the owner of the repository. Otherwise it can just be the ED25519 public key that one wants to use with this system. The command is as follows:

```
./gac.py create_repo <path-to-your-ed25519-public-key>
```

Depending on how the host machine is usually accessed one might need to setup the SSH config file to use the correct key. This can be done by adding the following lines to the `~/.ssh/config` file and replacing the placeholders with the correct values:

```
Host <hostname>
    HostName <hostname>
    Port <port>
    User <user-on-server>
    IdentityFile <path-to-your-ed25519-public-key>
    IdentitiesOnly yes
```

The hostname is the hostname or IP address of the host machine. If the software is running in Docker with the example Dockerfile the port is 2222. If one is running the software on the host machine the port is 22. The user is the user on the server which is hosting the repository. If one is hosting the repository on Docker it is `Git`. The path to ones ED25519 public key is the path to the public key that one wants to use with the access control software. This setup will allow one to access the repository without having to specify the key every time.

5 Evaluation

5.1 Correctness of the Specification

To guarantee the correctness of a TLA+ model it can be checked by the TLC model checker.

5.1.1 Limiting the State Space

If we just let the TLC model checker run on this model it will never finish since the amount of trust operations is not limited in any way. To have the model checker finish we need to limit the amount of $AddTrust$ and $RemoveTrust$ operations possible.

For this we have defined the functions $Max3$ which computes the maximum of three natural numbers and $Add3$ which adds 3 natural numbers together that are rather simple and will not be discussed further here.

We also have the function $Fold(Fn3)$ in Figure 5.1 that takes a function that takes three arguments as an argument. It then calculates the given function for all the counters that signify how often either $AddTrust$ or $RemoveTrust$ have been executed on a tuple.

The function defines a helper function called $Helper[C \in \text{SUBSET } Clients, C2 \in \text{SUBSET } Clients]$ using the LET keyword [15, p. 299]. Here $x \in \text{SUBSET } Set$ means that x is a subset of Set [15, pp. 299–301]. This function returns 0 if either C or $C2$ are the empty set. Otherwise it chooses a c in C and $c2$ in $C2$ using the CHOOSE keyword [15, pp. 294–295]. This keyword is needed because the recursion needs to apply to a smaller set to terminate.

It then calls the given function $Fn3$ with two arguments being the helper function again. Once the helper function is called with C without c and $C2$ and once with C and $C2$ without $c2$. The third argument being the natural number that signifies how often the trust relation from c to $c2$ has been modified in the replica of c. This has to be the highest number for this relationship because the relationship has to be modified in the local replica first before it can be propagated to other replicas.

In the end $Fold(Fn3)$ calls the helper function with the arguments $Clients$ and $Clients$. This means that if we call this function with the argument $Add3$ it returns the number of $AddTrust$ or $RemoveTrust$ actions that have been performed. It however will not count the number of $Merge$ actions.

This allows us to limit how many of those actions can be performed. By limiting the number of $AddTrust$ and $RemoveTrust$ operations, the only other operation that can be performed is a $Merge$ operation, which eventually results in all replicas having the same trust graph and no further state changes. This way, this results in a finite state space for the model checker to verify.

$$
\begin{array}{l}
Fold(Fn3) \triangleq \\
\quad \text{LET } Helper[C \in \text{SUBSET } Clients, \\
\qquad\qquad C2 \in \text{SUBSET } Clients] \triangleq \\
\qquad \text{IF } C = \{\} \vee C2 = \{\} \text{ THEN } 0 \\
\qquad\quad \text{ELSE LET } c \triangleq \text{CHOOSE } c \in C : \text{TRUE} \\
\qquad\qquad\qquad\quad c2 \triangleq \text{CHOOSE } c2 \in C2 : \text{TRUE} \\
\qquad\qquad\quad \text{IN } \quad Fn3[Helper[C \setminus \{c\},\ C2], \\
\qquad\qquad\qquad\qquad\quad Helper[C,\ C2 \setminus \{c2\}], \\
\qquad\qquad\qquad\qquad\quad TG[c][c].trusts[c2][2]] \\
\quad \text{IN } \quad Helper[Clients,\ Clients]
\end{array}
$$

Figure 5.1: Helper Function $Fold(Fn3)$ from the TLA specification

5.1.2 Running the Model Checker

The model checker needs to be limited rather heavily since the number of states in the model increase exponentially. For the results of the model checker to be useful we need to run the model checker with at least three different model values. Otherwise it is impossible to reach states in which a client gains trust through another client transitively. Because the model checking would take too long otherwise it was run with three clients as model values and restricted with either $Fold(Max3) \leq 1$ or $Fold(Add3) \leq 7$. $Fold(Max3) \leq 1$ allows 1 trust operations to be performed for each client by each client. $Fold(Add3) \leq 7$ allows each client to do a maximum of 7 operations but is not restricted for which client these operations are done. With these constraints the TLC model checker was not able to find any errors in the model and produced the results in Table 5.1.

Table 5.1: Results of the TLC model checker

State Constraint	Distinct States	Runtime
$Fold(Max3) \leq 1$	928 995	05:01
$Fold(Add3) \leq 7$	688 612	04:12

5.1.3 Errors with Model Checking

Checking the liveness while restricting the model checker is challenging since the model checker has along runtime. For this reason it was only run with three clients. Because of this an error in the liveness function *InfOftenTransNotCanPull* seen in Figure 3.8 went uncaught until later in the process. The function used to be written as seen in Figure 5.2. This is wrong because here it specifies that the transitive property does not hold for one arbitrary client. For the function to be correct the transitive trust has be false for all clients.

$$
\begin{aligned}
\mathit{TransCanNotPull} \triangleq\ & \exists\, c1,\ c2 \in \mathit{Clients} : \\
& \forall\, c3\ \ \in \mathit{Clients} : \\
& \quad (\ \wedge \neg TG[c1][c1].\mathit{trusts}[c2][1] \vee \neg TG[c3][c3].\mathit{trusts}[c1][1] \\
& \quad\ \wedge \neg TG[c3][c3].\mathit{trusts}[c2][1] \\
& \quad\ \wedge c2 \neq c3) \\
& \qquad \Rightarrow \neg \mathit{CanPull}(c2,\ c3) \\
\mathit{InfOftenTransCanNotPull} \triangleq\ & \Box\Diamond \mathit{TransCanNotPull}
\end{aligned}
$$

Figure 5.2: Wrong liveness property *InfOftenTransNotCanPull*

5.2 Security

This section aims to evaluate the security properties of the produced software. This is done by reviewing the implemented security measures.

5.2.1 System Assumptions

It is assumed that users on this system do not share their private key with anybody and that their machines are not infected with malicious software. We also assume that the owner of a replica is trusted and will not modify the trust graph manually for entries other than his own. Furthermore we assume that a user does not fork his personal access control branch and pushes the two different versions to two different replicas.

5.2.2 Assumptions on SSH and Git

This application is built on top of Git and SSH which are widely used and actively developed. For this security evaluation we assume that there are no exploits in these technologies. It is assumed that one can not log into a server using SSH if that person is not in possession of the private key that corresponds to the deposited public key. Furthermore it is assumed that one can not create Git commits with a valid signature without being in the possession of the private key that corresponds to the fingerprint of the public key Git reports as being used to sign the commit.

5.2.3 Unforgeable Commits

No one that is not in possession of the private key of a user can forge a commit that adds or removes trust from this user to anybody. This is due to the self certification of each commit as mentioned in section 4.2. Because each commit is signed with the private key corresponding to the user that supposedly made it. This allows us to easily verify that the commit was made by this user. This means it is impossible to create a commit that changes trust relationships for a user if one does not possess the corresponding private key.

5.2.4 Ignoring Invalid Reference Modifications

We do not allow deletion of access control branches or force pushes since these would lead to inconsistencies between the stored trust graph and the trust graph encoded in the commits. This is done by checking if the new revision passed to the script is only zeros which indicates branch deletion. After ensuring that there is no branch deletion we make sure that the old revision is an ancestor of the new revision. This is to make sure there are no forks or deletion of commits.

5.2.5 Non-replication of Invalid Commits

For each commit pushed to an access control branch the `pre-receive` script [16] checks if there are no merge commits. This is done by checking if any of the new commits has two or more parents. It then verifies all cryptographic properties of the commit. This means it ensures that the commit was signed by the same key that is in the branch name and the committer email field of the commit.

5.2.6 Enforcing Valid Pull Access

Only people who can pull accordingly to the trust graph definition in section 3.4 can successfully pull from the Git repository. This is done through restricting the SSH access discussed in subsection 4.4.1. The scripts to which the command is passed verify that it is a command that pulls from the respective repository and if it is not deny access. Granting access is done through a Git hook discussed in subsection 4.4.2. This hook only grants access if the user should have access to the repository according to the definitions in section 3.4. This is done by iterating through all users trusted by the owner directly and all users that are trusted by a user who is directly trusted by the owner. Since commits can not be forged this is secure.

5.2.7 Enforcing Valid Push Access

Only people who can push accordingly to the trust graph definition in the function *CanPush(c,r)* (Appendix A) can successfully push to the Git repository. The SSH script discussed in subsection 4.4.1 will verify the command. Access is granted through the Git hook discussed in subsection 4.4.2. Here we only grant access to the users directly trusted by the owner by iterating through them and this is also secure since commits can not be forged.

5.3 Deployment Tests on a Single Machine

During development the software was tested using four virtual machines in a virtual network using Virtual Box. The software was set up in one machine and the others were used to connect to it, one being the owner. It needed to be at least four virtual machines since we needed to test the transitive capabilities. Here the software was mainly setup manually since most of the ease of use improvements were not developed at this point. It was tried to manipulate the trust graphs in various ways by creating the commits in unconventional ways. This revealed certain bugs in the software that could be fixed. Once the software was mostly done it was first deployed in a local docker container for more testing.

5.4 Complexity of the Implementation

The software needs certain dependencies to run. Firstly we need a linux distribution to handle our operating system needs. Alpine Linux was chosen because of its small size [19]. Since the implementation is built on top of Git and SSH we need those as well. All code is written in Bash or Python so those need to be installed as well. We also need a non standard Python package called PyYAML because our trust graph is stored in a YAML file. To install this package we need Pip. We then also need to add our software to the container which is two scripts. The `entrypoint.sh` script that sets everything up correctly in the container

and the `gac.sh` script that installs our software [16]. As shown in Table 5.2 this results in a total size of 90.5 MB for the container. The main contributor to this size is the Python installation. The size could be significantly reduced if the implementation was completely in Bash. One could also install the PyYAML package directly without using Pip which would save space.

Table 5.2: Size of the Docker container

Program	Size
Alpine Linux	7.83 MB
Git	12.22 MB
OpenSSH	5.89 MB
Bash	2.05 MB
Python 3	39.62 MB
Python 3 Pip	19.07 MB
PyYAML Package	3.79 MB
`gac.sh`	0.00818 MB
`entrypoint.sh`	0.00126 MB
Adding a User	0.00302 MB
Total	90.5 MB

5.5 Actual Deployment and Usage for Collaboration on Thesis Writing

The Docker container was deployed to a server for use in backing up and collaborating on this thesis. This demonstrates that the software is functional and suitable for collaborative work and data backup.

5.6 Updating the Software

Updating the software is very clumsy since all branches of the access control software depend on the hash of the root commit. This means when one has a new version of the software this hash changes since the software it contains is not the same anymore. Therefore all old access control branches are not usable anymore. One has to create a new server, redo all access control actions and transfer the branches from the old to the new server.

6
Related Work

This chapter aims to compare the work done in ths thesis to other related software and research projects.

6.1 Gitolite

Gitolite does something similar to what was done in this thesis [20]. It is a way to manage Git repositories on a server but compared to the software presented here supports arbitrary many repositories per user and has finer access control like restricting a user to a branch or tag of a repository.

It also makes use of restricting a user that accesses the server using SSH through the command option in the `authorize_keys` file [21]. Instead of simply restricting the commands a user can execute it is handled in a more flexible way by implementing a shell. This shell gets passed the username and the command the user wishes to execute. It then determines if the user has access to said repository. If a user is restricted to a certain branch or tag it makes use of the update Git hook to determine if said user has access to that branch or tag [20].

However access to a repository hosted with Gitolite is still always granted by an administrator. It is done by editing a configuration file in or adding keys to a special administration repository and pushing these changes to the server. This repository has a special post-receive hook that for example updates the `authorize_keys` file and creates repositories [22].

This is were the software presented in this thesis differs a bit from Gitolite. In our software anybody trusted by the owner can grant read only access to anybody they wish. This is not possible in Gitolite since you could give access to the administration repository to other people but this would also mean that they could change access for anybody else. The reason because of which this is possible is that our implementation uses a trust graph and the operations on that trust graph ar encoded in commits on a Git branch compared to a config file in a repository. This is a key difference to our system.

6.2 Grassroots Systems

Our application fits the definition of a grassroots system given by Ehud Shapiro [23]. This is because there can be several instance of this application on different servers that have nothing to do with each other. If now a user that has pull access to one of these instance gains push access to the another instance they can transfer all parts of the trust graph from one instance to the other integrating the trust graph from the first into the trust graph of the second instance.

6.3 Other Applications on Git

There are also other applications that were implemented on top of Git for example the 2P-BFT-Log by Erick Lavoie [13]. In comparison to the implementation of this thesis the 2P-BFT-Log is less tolerant of malicious actors. If a person forks their log and sends the two concurrent branches to two different replicas they will eventually notice this inconsistency and mark that person as malicious. That log now enters a shrinking phase in which the replicas will find the earliest concurrent messages on the log and mark them. This would go down rather differently in our implementation. If a person forked their log here and pushed the two concurrent branches to two different replicas the replicas would never agree on the log of that person again.

Another application on top of Git is the Delta-GOC-Ledger by Jannick Heisch [24]. His thesis implements a δ-based GOC-Ledger on top of Git. It also encodes some things in commits but in contrast to this thesis the actual information about the transactions is encoded in tree objects which are not used in our implementation.

6.4 Secure Scuttlebutt

Secure Scuttlebutt also uses a self-certifying append-only log and has highly accessible replicas to disseminate messages [25]. In comparison to this implementation the replicas are no different from the clients apart from having a user interface and are usually public. It is designed as pure peer-to-peer system and can function without these highly available replicas. Any user can follow a pub in SSB, which would correspond to pull privileges in our system. To get a pub to follow you back, which would correspond to push privileges in our system, one needs an invite code. How these invite codes are handed out depends on the operator of the pub [26].

7 Conclusions

We have shown a secure and simple system to control the access to server hosted Git repositories. The TLA+ model ensures that the system behaves correctly. Using self-certification we can ensure that only valid updates to the access control system are made. The system was also deployed using docker and used in the real world to show its functionality. It can be used in small teams to collaborate on a project or to back up data. However updating the software is still rather cumbersome and a bit of work since the software depends on the hash of a commit that contains the software. This makes old versions of the software incompatible with newer versions of the software. It is still an interesting approach to handle access control with a trust graph secured by self-certification. This allows the administrator to delegate some work and makes managing access control for a repository simpler. It however lacks finer control like Gitolite for example and is therefore only suited for smaller groups of developers.

7.1 Future Work

Using a shell like it is done in Gitolite seems like a good approach. It could be used in this system as well to allow a single user on a single server to host several replicas while still encoding the trust operations in Git commits. It would however not be possible to run the software only in the repository itself but there would need to be communication between the different repositories. Making future versions of the software backwards compatible with older versions and enabling easy updates is also something that should be explored. Adding the option for other people except the owner to grant read and write privileges is also something that could be added. This would enable the owner to have a substitute. Supporting different key types would also be a good addition to the software presented in this thesis.

Bibliography

[1] GitHub. Pricing · Plans for every Developer (n.d.). URL https://github.com/pricing. Accessed: 2025-05-29, Archived version: https://web.archive.org/web/20250529135248/https://github.com/pricing.

[2] GitLab. Features | GitLab (n.d.). URL https://about.gitlab.com/features/. Accessed: 2025-05-29, Archived version: https://web.archive.org/web/20250519194510/https://about.gitlab.com/features/.

[3] Microsoft. Microsoft acquires GitHub - Stories (n.d.). URL https://news.microsoft.com/announcement/microsoft-acquires-github/. Accessed: 2025-05-29, Archived version: https://web.archive.org/web/20250510112857/https://news.microsoft.com/announcement/microsoft-acquires-github/.

[4] GitHub. GitHub Copilot · Your AI pair programmer (n.d.). URL https://github.com/features/copilot. Accessed: 2025-05-14, Archived version: https://web.archive.org/web/20250514003651/https://github.com/features/copilot.

[5] GitLab. Download and install | GitLab (n.d.). URL https://about.gitlab.com/install/. Accessed: 2025-05-29, Archived version: https://web.archive.org/web/20250525053709/https://about.gitlab.com/install/.

[6] Chacon, S. and Straub, B. *Pro Git*. Apress, 2nd edition (2025). URL https://git-scm.com/book/en/v2. Version 2.1.447.

[7] The Git Development Community. *Git - pretty-formats Documentation* (2025). URL https://git-scm.com/docs/pretty-formats. Accessed: 2025-06-01.

[8] The Git Development Community. *Git - git-commit-tree Documentation* (2025). URL https://git-scm.com/docs/git-commit-tree. Accessed: 2025-06-01.

[9] The Git Development Community. *Git - githooks Documentation* (2024). URL https://git-scm.com/docs/githooks. Accessed: 2025-05-01.

[10] The OpenBSD Development Community. *sshd(8) - OpenBSD manual pages* (2024). URL https://man.openbsd.org/sshd. Accessed: 2025-06-01.

[11] Diffie, W. and Hellman, M. E. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654 (1976). DOI: 10.1109/TIT.1976.1055638.

[12] Mazières, D. Self-Certifying File System. Ph.D. thesis, Massachusetts Institute of Technology (2000). URL https://www.scs.stanford.edu/~dm/home/papers/mazieres:thesis.ps.gz.

[13] Lavoie, E. 2P-BFT-Log: 2-Phase Single-Author Append-Only Log for Adversarial Environments (2023). arXiv:2307.08381.

[14] Lamport, L. A High-Level View of TLA+ (2021). URL https://lamport.azurewebsites.net/tla/high-level-view.html. Accessed: 2025-06-01, Archived version: https://web.archive.org/web/20250526172354/https://lamport.azurewebsites.net/tla/high-level-view.html.

[15] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 1st edition (2002). URL https://lamport.azurewebsites.net/tla/book-21-07-04.pdf. ISBN: 0-321-14306-X.

[16] Walser, P. SSH-based Access Control of Server-Hosted Replicas for Git-based Applications (2025). URL https://github.com/piuswa/gac.

[17] SSH. Configuring Authorized Keys for OpenSSH (n.d.). URL https://www.ssh.com/academy/ssh/authorized-keys-openssh. Accessed: 2025-05-30, Archived version: https://web.archive.org/web/20250515221536/https://www.ssh.com/academy/ssh/authorized-keys-openssh.

[18] fadedbee. Does the SSH protocol send the remote name to the remote machine? Stack Overflow (2018). URL https://stackoverflow.com/questions/52455205/does-the-ssh-protocol-send-the-remote-name-to-the-remote-machine. Accessed: 2025-05-01, Archived version: https://web.archive.org/web/20241220110836/https://stackoverflow.com/questions/52455205/does-the-ssh-protocol-send-the-remote-name-to-the-remote-machine.

[19] Alpine Linux Development Team. about | Alpine Linux (n.d.). URL https://www.alpinelinux.org/about/. Accessed: 2025-06-01, Archived version: https://web.archive.org/web/20250530183254/https://www.alpinelinux.org/about/.

[20] Chamarty, S. overview - Gitolite (n.d.). URL https://gitolite.com/gitolite/overview.html. Accessed: 2025-05-30, Archived version: https://web.archive.org/web/20250310003130/https://gitolite.com/gitolite/overview.html.

[21] Chamarty, S. how gitolite uses ssh - Gitolite (n.d.). URL https://gitolite.com/gitolite/glssh.html. Accessed: 2025-05-30, Archived version: https://web.archive.org/web/20241214003715/https://gitolite.com/gitolite/glssh.html.

[22] Chamarty, S. concepts, conventions, terminology - Gitolite (n.d.). URL https://gitolite.com/gitolite/concepts.html. Accessed: 2025-05-30, Archived version: https://web.archive.org/web/20250527151356/https://gitolite.com/gitolite/concepts.html.

[23] Shapiro, E. Grassroots Systems: Concept, Examples, Implementation and Applications (2023). arXiv:2301.04391.

[24] Heisch, J. Delta-GOC-Ledger: Incremental Checkpointing and Lower Message Sizes for Grow-Only Counters Ledgers with Delta-CRDTs (2024). URL https://cn.dmi.unibas.ch/fileadmin/user_upload/redesign-cn-dmi/pubs/theses/master/Heisch-Delta-GOC-Ledger.pdf. Accessed: 2025-06-16, Archived version: https://web.archive.org/web/20250227032428/https://cn.dmi.unibas.ch/fileadmin/user_upload/redesign-cn-dmi/pubs/theses/master/Heisch-Delta-GOC-Ledger.pdf.

[25] Tarr, D., Meyer, A., Lavoie, E., and Tschudin, C. Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications. *Proceedings of the 6th ACM Conference on Information-Centric Networking* (2019). DOI: 10.1145/3357150.3357396.

[26] The Secure Scuttlebutt Development Community. Scuttlebutt Protocol Guide (2025). URL `https://ssbc.github.io/scuttlebutt-protocol-guide/#pubs`. Accessed: 2025-06-12, Archived version: `https://web.archive.org/web/20250501033448/https://ssbc.github.io/scuttlebutt-protocol-guide/#pubs`.

Appendix A: TLA+ Module

MODULE *prerecieveLogic*

EXTENDS *Naturals*
VARIABLES *TG*
CONSTANTS *Clients*

Helper

$$
\begin{array}{l}
Max[x \in Nat,\, y \in Nat] \triangleq \text{IF } x < y \text{ THEN } y \text{ ELSE } x \\
Max3[x \in Nat, \\
\qquad y \in Nat, \\
\qquad z \in Nat] \triangleq Max[x,\, Max[y,\, z]] \\
Add3[x \in Nat, \\
\qquad y \in Nat, \\
\qquad z \in Nat] \triangleq x + y + z
\end{array}
$$

$$
\begin{array}{l}
Fold(Fn3) \triangleq \\
\quad \text{LET } Helper[C \in \text{SUBSET } Clients, \\
\qquad\qquad C2 \in \text{SUBSET } Clients] \triangleq \\
\qquad \text{IF } C = \{\} \vee C2 = \{\} \text{ THEN } 0 \\
\qquad \text{ELSE LET } c \triangleq \text{CHOOSE } c \in C : \text{TRUE} \\
\qquad\qquad\qquad\quad c2 \triangleq \text{CHOOSE } c2 \in C2 : \text{TRUE} \\
\qquad\qquad\quad \text{IN } Fn3[Helper[C \setminus \{c\},\, C2], \\
\qquad\qquad\qquad\qquad Helper[C,\, C2 \setminus \{c2\}], \\
\qquad\qquad\qquad\qquad TG[c][c].trusts[c2][2]] \\
\quad \text{IN } Helper[Clients,\, Clients]
\end{array}
$$

Initialization

We assume one trust graph per client, represented by r

$$
\begin{array}{l}
Init \triangleq TG = [r \in Clients \mapsto \\
\qquad\qquad [c1 \in Clients \mapsto \\
\qquad\qquad\quad [trusts \mapsto \\
\qquad\qquad\qquad [c2 \in Clients \mapsto \\
\qquad\qquad\qquad\quad \langle \text{FALSE},\, 0 \rangle]]]]
\end{array}
$$

$$vars \triangleq \langle TG \rangle$$

Calculations from *Trustgraph*

Can client c pull/push from the other client r ?

$$
\begin{array}{l}
CanPull(c,\, r) \triangleq \\
\quad \wedge\, c \in Clients \\
\quad \wedge\, r \in Clients \\
\quad \wedge\, \vee\, c = r \\
\qquad \vee\, TG[r][r].trusts[c][1]
\end{array}
$$

$$\lor \exists\, c2 \in \mathit{Clients} :$$
$$\quad \land\ TG[r][c2].\mathit{trusts}[c][1]$$
$$\quad \land\ TG[r][r].\mathit{trusts}[c2][1]$$

$$\mathit{CanPush}(c,\ r) \triangleq$$
$$\land\ c \in \mathit{Clients}$$
$$\land\ r \in \mathit{Clients}$$
$$\land\ \lor\ c = r$$
$$\quad \lor\ TG[r][r].\mathit{trusts}[c][1]$$

Actions

Each client $c1$ can only modify their trust relationship to other clients in their own section of the trust graph.

$$\mathit{AddTrust} \triangleq \exists\, c1,\ c2 \in \mathit{Clients} :$$
$$\land\ TG[c1][c1].\mathit{trusts}[c2][1] = \text{FALSE}$$
$$\land\ TG' = [TG \text{ EXCEPT } ![c1][c1].\mathit{trusts}[c2] = \langle \text{TRUE},\ TG[c1][c1].\mathit{trusts}[c2][2] + 1 \rangle]$$

$$\mathit{RemoveTrust} \triangleq \exists\, c1,\ c2 \in \mathit{Clients} :$$
$$\land\ TG[c1][c1].\mathit{trusts}[c2][1] = \text{TRUE}$$
$$\land\ TG' = [TG \text{ EXCEPT } ![c1][c1].\mathit{trusts}[c2] = \langle \text{FALSE},\ TG[c1][c1].\mathit{trusts}[c2][2] + 1 \rangle]$$

$$\mathit{Merge} \triangleq \exists\, c1,\ c2 \in \mathit{Clients} :$$
$$\land\ TG[c1] \neq TG[c2]$$
$$\land\ TG' = [TG \text{ EXCEPT } ![c1] =$$
$$\qquad [c3 \in \mathit{Clients} \mapsto$$
$$\qquad\ [\mathit{trusts} \mapsto [c4 \in \mathit{Clients} \mapsto$$
$$\qquad\quad \text{IF } TG[c1][c3].\mathit{trusts}[c4][2] < TG[c2][c3].\mathit{trusts}[c4][2]$$
$$\qquad\quad\ \text{THEN } TG[c2][c3].\mathit{trusts}[c4]$$
$$\qquad\quad\ \text{ELSE } TG[c1][c3].\mathit{trusts}[c4]]]]]$$

$$\mathit{Next} \triangleq$$
$$\lor\ \mathit{Merge}$$
$$\lor\ \mathit{RemoveTrust}$$
$$\lor\ \mathit{AddTrust}$$

Invariants

$$\mathit{TypeOK} \triangleq TG \in [\mathit{Clients} \to [\mathit{Clients} \to [\mathit{trusts} : [\mathit{Clients} \to \text{BOOLEAN} \times \mathit{Nat}]]]]$$

$$\mathit{LocalTrustFirst} \triangleq \forall\, c1,\ c2,\ c3 \in \mathit{Clients} :$$
$$\lor\ TG[c1][c1].\mathit{trusts}[c3] = TG[c2][c1].\mathit{trusts}[c3]$$
$$\lor\ TG[c1][c1].\mathit{trusts}[c3][2] > TG[c2][c1].\mathit{trusts}[c3][2]$$

Temporal Formulas

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$$

$$Fairness \triangleq \mathrm{WF}_{vars}(Merge)$$

$$FairSpec \triangleq Spec \land Fairness$$

Liveness

$$\begin{aligned} TransCanPull \triangleq\ & \exists\, c1,\ c2 \in Clients : \\ & \forall\, c3\ \in Clients : \\ & (TG[c3][c3].trusts[c1][1] \land TG[c1][c1].trusts[c2][1]) \\ & \Rightarrow CanPull(c2,\ c3) \end{aligned}$$

$$\begin{aligned} TransCanNotPull \triangleq\ & \exists\, c2 \in Clients : \\ & \forall\, c1 \in Clients : \\ & (\land \neg TG[c1][c1].trusts[c2][1] \\ & \land c1 \neq c2 \\ & \land \forall\, c3 \in Clients : \\ & \quad \lor \neg TG[c1][c1].trusts[c3][1] \\ & \quad \lor \neg TG[c3][c3].trusts[c2][1]) \\ & \Rightarrow \neg CanPull(c2,\ c1) \end{aligned}$$

$$InfOftenTransCanPull \triangleq \Box\Diamond TransCanPull$$
$$InfOftenTransCanNotPull \triangleq \Box\Diamond TransCanNotPull$$

* Modification History
* Last modified *Wed Jun* 04 14:12:27 *CEST* 2025 by *pius*
* Last modified *Fri Apr* 11 16:25:52 *CEST* 2025 by *lavoie*
* Created *Mon Apr* 14 10:38:55 *CEST* 2025 by *pius*