HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Master Project

Git-Anon: Anonymous Git with Signatures

Master Informatik
Faculty of Elektrotechnik und Informatik
Hochschule Ravensburg-Weingarten

Author: Erik Escher
Matriculation Number: 33443

Time Period: 2020-05-12 - 2021-03-15
Date: 2021-03-15
Primary Examiner: Prof. Dr. rer. nat. Tobias Eggendorfer

Statement of Originality

I hereby confirm to have created the accompanying document by myself, without contributions from any sources other than those cited in the text and acknowledgments. This document has not been provided in similar or identical form to any other examination authority. It has not been published either.

2021-03-15 in Friedrichshafen ______________________________

Erik Escher

Abstract

Git-Anon is a system that combines anonymity with cryptographically verifiable assertions about roles and attributes of signatories.

The main premise behind git-anon is that commits are signed using unique ephemeral keys and multiple commits from the same author cannot be linked together. This provides stronger anonymity guarantees than traditional pseudonym based approaches.

Furthermore, users can claim a set of attributes and roles that can be certified by certificate authorities and then choose for each commit, which of these attributes and roles they want to share publicly or with a closed group. Both external observers as well as other team members can cryptographically verify that the person creating a commit has the attributes and roles that they make available to the relying party through a full chain of trust from the manually trusted certification authorities through ephemeral identities and to the individual commit.

More information about identities can be claimed and revealed by the keyholder at any time or certified by certificate authorities as needed.

This results in a signature scheme that allows signatories to choose which attributes or roles they reveal about themselves for each signature minimizing the amount of personal data that needs to be shared and retained while still allowing observers to verify that the signatory was authorized to sign the relevant information.

This reduces the risk of exposure, decrease the amount of security required to protect the data and helps fulfill legal requirements imposed by data protection regulation.

It also drastically reduces the amount and granularity of information that is publicly available about contributors working on open source projects, protecting their privacy while still providing the information needed to collaborate with trusted colleagues.

Contents

1. Introduction

1.1. Motivation

Current Challenges

Git is the de facto standard for (distributed) version control [9, 23].

However, its distributed history and the major drawbacks of editing older commits mean that once a commit is published, it is effectively impossible to change any aspect of it. While this is a design goal and useful in many circumstances it also introduces challenges from a privacy perspective. With laws such as the "Right to be forgotten"[21, 29], authors might have the right to have their name removed from their contributions should they so choose potentially years later. Additionally an author might not want to reveal their identity publicly when working on controversial projects, while still allowing other project members to identify them as the author to facilitate everyday interactions.

Existing Solutions and their Challenges

Pseudonyms

While the author could choose a single pseudonym, inform his peers of this pseudonym and then use it for a project, this brings a number of drawbacks. Perhaps the most significant being that this pseudonymization will likely be fairly ineffective unless the pseudonym is frequently and carefully changed. Public interactions or mistakes might easily link a particular commit to a real identity which would then in turn link all other commits using that pseudonym to it as well. On the other hand switching pseudonyms too will often reduce the value of even having an author mentioned in the commit as other project members would frequently have to manually memorize or record the new mapping.

Removing Author Information Completely

Another option would be to change the author information to a value that is used by multiple authors, however this would mean losing all ability to identify and contact the actual author of a given commit and the changes introduced by it.

Digital Signatures

Finally, there is also the issue of commit authenticity. The author information in commits can easily be forged and with a distributed version control system authenticating the connection that introduced the commit is only of limited usefulness as commits are often legitimately pushed by a user that is different than the original author or committer. The way to solve this problem is by having committers digitally sign their commits so that they can later be validated easily, however this introduces the signature key as another identifier that can be used to link commits.

Reasons for Anonymity

There are various reasons for wanting to remain anonymous towards external observers. Such as:

- A company selling work performed by one of their employees, where the company does not want the customer to know which (or how many) employee(s) performed the work.
- An employee working for a company on a public project that he does not want to be publicly attributed for.
- Developers working on a project that is completely acceptable in their country of residence (legally and morally) but not in a country that they might have to travel to one day.
- An organization wanting the ability to limit (or erase) the knowledge of who authored a particular piece of software, whose source code is publicly available.
- An organization wanting to avoid the legal ramifications of creating a public record of exact working times (and indirectly durations) of developers working on public (or internal) projects. Or not wanting to record how many lines each employee contributed.

1.2. Goals

This project aims to create a system that combines the ability to anonymize commits towards the public while maintaining identifyability towards a chosen group of people (project members) and providing cryptographically verifiable assertions about the signatory of a commit. In many cases an external relying party does not need to know the real name of the signatory but only that they were authorized to sign the given commit or release based on their role.

1.3. Scope

Protection against deanonymization based on the contents of the files in the repository or by the entity hosting the repository are explicitly out of scope for this project. Reasons and details are provided in chapter 5.

2. Definitions and Technical Background

2.1. Git

Git [6, 7, 8] is a distributed version control system, which means that in typical operation everyone has a full local copy of the entire shared history and all of these copies are equally valid and important. In theory this means that no central, highly available instance of the repository is needed and instead commits can be exchanged using other decentral systems such as mail or direct connections between workstations. While some teams work this way, most of the time a source-of-truth is defined on a central server and the decentralized nature is used for unfinished versions, offline usage and performance.

To allow for a decentralized history that can be accurately and securely synced, git uses a combination of hash chaining and content addressed storage. The main objects representing the history of a git repository are stored in a content addressed system meaning that the filename/path for each object is determined by a hash of its contents. This hash is then also used when referencing objects from other objects.

To represent the current state of the filesystem at the time of the commit a tree of `tree` objects is created representing the folder structure with its filenames, permissions and references to `blob` objects containing the files contents. These `blob` objects consist of the compressed file contents and are referenced by their hash. To represent a sequence of states of the filesystem combined with additional metadata (also known as the history of the repository), `commit` objects are used. Each commit object contains a reference to a tree object and one or more parent commits that they are based on. It also contains additional metadata in the form of author and committer information as well as a free-form message. Optionally, commits may contain a single gpg signature from the committer. An example of such an object graph is shown in Figure 2.1 with examples for a commit in Listing 2.1 and for a tree object in Listing 2.2.

This content addressed storage system provides a simple and automatic deduplication system because whenever a duplicate object is to be stored, its hash matches that of the existing object, which will then be used instead.

When referring to a commit, its hash can be used. However, when one wants to refer to a version of the repository that can later be changed, a concept of memorable branches and tags can be used. These are effectively pointers to a commit and can be changed easily. Tags can also contain messages and signatures.

The content-addressed objects in a repository can not be changed without changing their hash value and therefore changing all other objects that reference them. This process, often referred to as “rewriting the history”, can cause lots of issues when interacting with other copies of the repository.

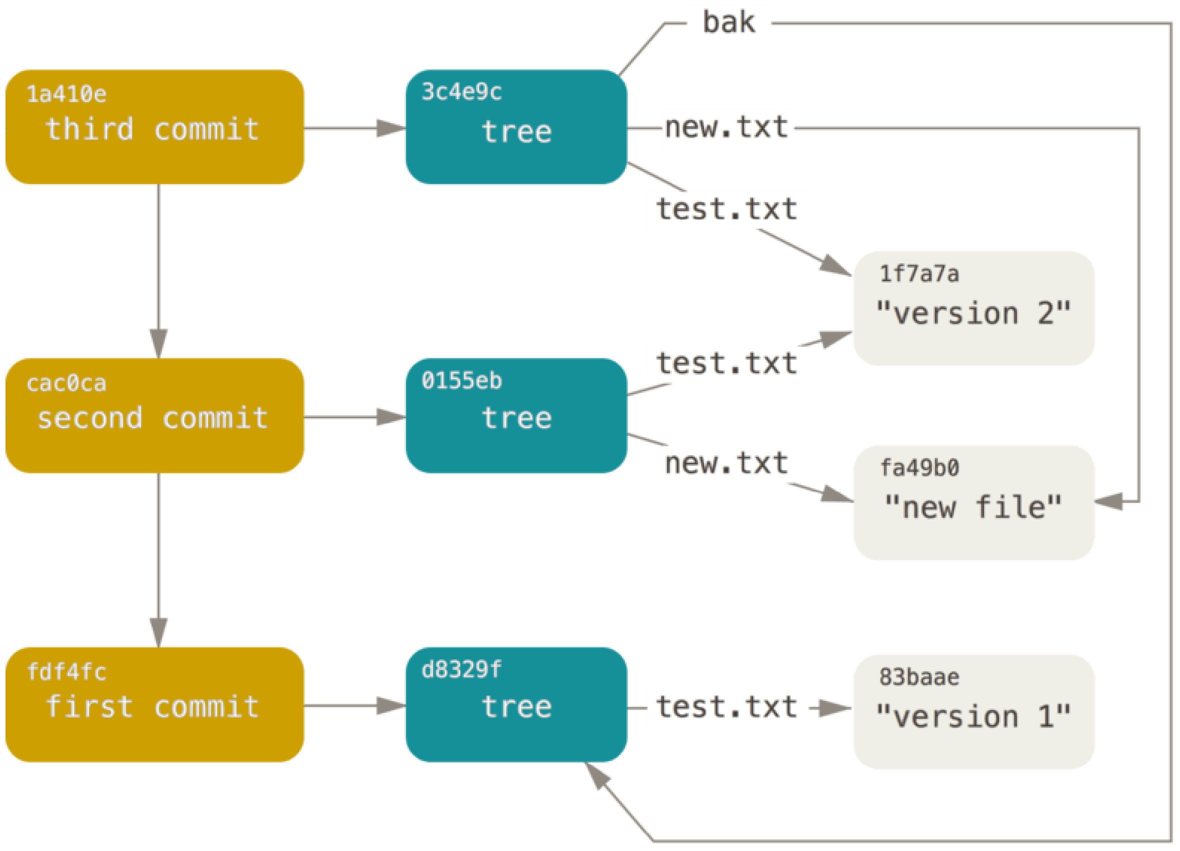

Figure 2.1.: Example of an object graph as used by git. [8]

```
tree 3934e6775f96fec7737d5f837fec62c4dcf15bbd
parent f2ff4fcb03f6547670f17cd43ae00b483633ddc3
author Erik Escher <git@erikescher.de> 1595006736 +0200
committer Erik Escher <git@erikescher.de> 1595006736 +0200
gpgsig -----BEGIN PGP SIGNATURE-----
 [[SIGNATURE DATA REDACTED FOR PRESENTATION]]
 -----END PGP SIGNATURE-----

This is a sample commit message.
```

Listing 2.1: Example of a commit object with an included signature.

```
100644 blob bd83801f897f05a09e1c286e20d4c50da0890952    testfile
040000 tree e4af7700f8c091d18cc15f39c184490125fb0d17    myfolder
```

Listing 2.2: Example of a tree object in git.

2.2. GPG/OpenPGP

OpenPGP [13] is a standard/specification describing a system for asymmetric encryption and signatures of messages and other objects using a system of public and private keys.

The most common implementation of OpenPGP is the “GNU Privacy Guard” [15] (GPG) program. In the following the terms OpenPGP and GPG will be used interchangeably and typically refer to the specification.

Since this project uses GPG extensively, readers should be familiar with it and some of its advanced features. A quick summary of important aspects is provided below.

Key Composition

GPG keys are composed of multiple components, that are bound together using special signatures. The main component of a gpg key is its public key material, which is hashed to produce the keys fingerprint and is the only part of the key that cannot be changed. Other important parts include User IDs, subkeys and the signatures that bind all of these together. These components can be extracted and stored or transmitted independently. In particular different versions containing different User IDs and certifications can be created and distributed.

An example of a gpg key consisting of multiple packets is provided in Appendix A.

User IDs

A GPG key can contain one or more User IDs (strings that represent the key holder). While the OpenPGP specification does not require the presence of a User ID, gpg seems to do so and can refuse to work with a key that does not provide one. Typically, this feature is used to include the key holders name in combination with multiple email addresses to simplify key discovery in clients.

Self-Signatures

To ensure that an attacker can not take advantage of the modular nature of gpg keys to inject malicious information, all parts of the key that can only be added by the key holder must be signed using the primary key material. While these signatures could be handled separately, for the purposes of this project they can be considered part of the User ID or subkey that they sign.

Certifications

Certifications are signatures from third-parties binding a UserID to a key and certifying that the entity described in the UserID is the holder of the key. A key can have an unlimited number of certifications, however at some point performance issues start to arise. While Certificate Authorities akin to the system in X.509 could exist, they don't seem to be used in practice. One notable exception is the signing service [16] operated by Governikus [17] on behalf of the Bundesamt für Sicherheit in der Informationstechnik [4].

Subkeys

A gpg key can contain additional public key material to be used instead of the main key material for various reasons. For this project these subkeys are irrelevant as only signature and certification capabilities are used and these can easily be attached to the main key material.

Trust and Verification

One of the most challenging problems with public key cryptography is verifying that a particular public key belongs to (was created by) a particular entity or person. Various systems exist to solve this using a hierarchical system (DANE), numerous certificate authorities that are ultimately trusted (X.509), trust on first use (SSH) or trust always (many other systems).

GPG itself offers multiple different models for verifying keys (specifically User IDs on keys) and leaves these choices up to the user. In practice this mostly means obtaining the key from a semi trusted source (Websites, DNS, ...) and optionally comparing its fingerprint with the key holder when meeting them in person.

For this project we will assume, that the public keys of anyone, whose signatures should be verified are available and trusted.

Contents of Signatures

Signatures produced by GPG are very bare. In particular and as opposed to the way signatures are handled in the X.509 ecosystem, gpg signatures do not contain the key that created them and cannot be verified on their own. Instead, they contain the last 8 bytes of the key fingerprint and optionally the full fingerprint in an additional field.

This means that to verify a signature the signatories public key must be present (or discoverable using mechanisms such as keyservers) and trusted as described above.

2.3. Anonymity versus Pseudonymity

Anonymity [5, 30, 27] is the concept of no name being known for an entity, which in particular makes it difficult to even link different actions from the same entity together.

Pseudonymity [28, 31] is the concept of an entities “real” name and identity being unknown with only an alternative name (a so-called “pseudonym”) known instead. This means that different actions performed by the same pseudonym can be linked together and are known (or suspected) to have been performed by the same entity. This allows an observer to learn more about the entity behind the pseudonym and increases his chances of linking the pseudonym to the entity by correlating known information.

Neither term implies that nobody knows the real identity of the entity taking the action only that the public or an external observer does not know. However, in a pseudonymous system, one can only choose to link the pseudonym and all associated actions to oneself whereas in an anonymous setting this decision can be taken for each action independently.

These definitions are compatible with those defined in [20].

2.4. Unlinkability

The term or concept of “unlinkability” is used multiple times in this project and refers to the inability of an external observer to link any two commits to the same author provided that at least one of them uses the anonymity system defined here and that sufficient other contributions using the anonymity system exist from potentially distinct authors.

2.5. Real/Civil versus Anonymous Identity

An identity for the purposes of this document consists of a name, optionally an email address that might be able to receive mail and potentially an associated gpg key. The gpg key can in turn have user ids that contain names and email addresses.

They are typically represented in the format of `Name Consisting of Multiple Parts <localpart@domain>` as specified for e-mail messages [22].

The terms real and civil identity are both used to describe an identity that can be linked to a person (or in fact any identity). Typically, this means their legal name but it can also refer to nicknames or widely known pseudonyms.

An anonymous identity for contrast would not allow this connection to be made.

3. Existing Solutions

There are multiple different solutions that aim to provide anonymity in relation to git, however their goals and functionality are quite different.

3.1. Long-Term Pseudonyms

An author might simply choose to use long-term pseudonyms to conduct their work, however this often provides very little protection against an adversary with some additional information about any event linking one of the commits to the author.

3.2. External Anonymization Systems

Systems exist to mask the identity of an author in a pull request where they replace any references to the author with their own identity [19, 11, 12].

The goal with these is typically to avoid potential bias based on attributes of the author (gender, nationality, perceived experience, etc.) during code reviews [19]. Another goal might be to allow double-blind review of the proposed changes or project [11, 12]. For the latter case entire repositories can be stripped of their history and a list of blacklisted words [11, 12].

One of the main drawbacks is that the anonymity is often not very strong and relies on the maintainer not actively seeking out the actual author. Another issue is that this typically leaves no information about the author in the actual commits in the repository making it much harder to contact the author of respective subsections for legitimate purposes.

Finally, the fact that these systems modify the commits at all, changing their hash, makes it harder to verify that no malicious changes have been introduced in the process of anonymization. In a project where the person looking at the commit trusts and knows the author, they might otherwise ask them for the hash of a suspicious commit or rely on the authors signature. Both are unusable when such an anonymization system is used as the commits are modified by the anonymization system on a typically untrusted host. The process also invalidates (and strips) any signatures, as these would otherwise have to be made in and by the anonymization system.

4. Requirements

The following is a list of requirements that the anonymization system described in this document should fulfill. They are listed in order of decreasing importance.

Inability for External Observers to Group Commits from the same Author

An external observer given full knowledge of how this system works must not be able to group two commits (or tags) created by the same person as long as the anonymization system was used for at least one of them.

This is to ensure that attacks based on knowledge about the authorship of a particular commit can not be leveraged to infer the authorship of another commit.

Backward Compatibility and Interoperability with Unaware Clients

Clients that are unaware of this new feature should be able to perform all operations as usual. For users whose identities they don't know they should simply see a pseudonym instead of their name. For signatures created under this scheme old clients should still see the signature, but verification may be harder and will not result in the name of the author but only sufficient information to ensure the signatory's role was authorized to sign the given commit or tag (for example a release tag).

This requirement also means that anonymous users, users that choose to reveal their identity and users that are unaware of the entire concept should be able to cooperate normally. From the perspective of an unaware user it will simply seem like the repository has a lot of additional users that contributed very little. The situation should be the same (or similar) for users unaware of the protocol and users that do not know the mapping between pseudonyms and real authors.

Anonymization is Reversible

When desired the author of a commit should be able to assert (and optionally prove) his authorship by publishing a piece of information. It should be possible to restrict the group of people that are aware of the authorship based on who this information is provided to. Others with knowledge of authorship may be able to forward this information to others.

Git-Tools Show the Real Author if the Corresponding Mapping is Known

Git command line tools and other tools built on top of them such as integrated development environments should display the real name of the author if the appropriate mapping is known. This includes functions such as `log`, `blame`, `tag`, `commit`, etc..

Support for GPG-Signatures Embedded in Commits

The scheme should support signing and verifying commits using attached gpg signatures without compromising the anonymity guarantees and while providing external users (that don't know the real author of the commit) with tools to verify certain attributes of the author that the author chooses to reveal. Such attributes could include their role as a member of the project ("maintainer", "release-architect", etc.) or even their full name if they so choose. These roles should be cryptographically verifiable given trust in a certificate authority (a public key that certifies the attribute).

This will be one of the most difficult goals to achieve especially in combination with the requirement of unlinkability between different commits.

Does not Clutter Existing GPG Configuration

The tool developed here should not clutter the users existing gpg setup with additional keys (especially not the thousands of keys that might be necessary for such a system).

Decentralized

One of the main advantages of git over some of the other version control systems is that it can be used decentrally without a central entity. An anonymization system for git should not require central entities for important functions. Where central entities can not be avoided, basic functions must still be possible even when the central entities are temporarily not available. In particular this includes creating commits whose commit id does not need to be changed later to ensure that the correct author name is shown when needed. Central entities are deemed acceptable but undesirable for the synchronization of information related to the mapping of pseudonyms to real identities.

5. Concessions

The following problems with the system are accepted as out of scope or impractical to fix due the other issues and side-effects that a solution would cause.

Anonymizing an existing Commit changes its Commit ID

An integral part of how git works is that each object (including commits) is referenced by its hash. Since commits include the authors name in full as configured in their client, this cannot be changed or removed without changing the cryptographic hash which functions as its identifier. Similarly, each commit includes the hash of its parent commit making it impossible to change anything about a commit without also changing all succeeding commits, which can in turn cause issues for collaborators.

Anonymity towards the Git-Hoster

Many hosting providers for git repositories log authentication and push events to repositories and some make this information available using a separate API to unauthenticated users. This is not a feature of git and can only be solved by disabling the logging (if possible) or pushing commits through a shared account so that they can not be distinguished by the account that pushed the commit. This would be easy to achieve with tools such as gitMask [19], git proxies or by pushing commits to a trusted hoster and configuring the repository there to be pushed to an untrusted git hoster [10].

This tool will also do nothing to protect interactions with the hoster using other features such as issue trackers or pull requests.

Protection against Deanonymization based on the File Tree or Commit Message

The file tree representing the files in the repository at the time of the commit and the message associated with the commit are typically considered the main parts of a commit. Both can contain sensitive information or information that deanonymizes the author of the commit but filtering this information would be very difficult and cause lots of issues in normal usage of a repository. It is therefore left up to the committer to ensure that these can not be used to deanonymize them.

Similarly, the content of commit messages is not considered. Removing identifying information from these is left up to the committer.

For some files in a repository containing sensitive information it might make sense to transparently encrypt them, which is offered by tools such as git-crypt [1, 2, 3, 25].

Masking of Metadata other than the Author and Committer

Commits include additional metadata such as the time the commit was authored and the time it was added ("committed") to the repository. Depending on the workflow chosen both might be equal or distinct and both could be used to aid deanonymization. Options for normalizing this data include setting the timezone to a predefined value such as "UTC" or rounding the time to the nearest minute, hour or day. However, they also provide additional value so care must be taken to not normalize them too much. For the initial version of this project neither will be done because both might harm the usability and the main focus of the project lies on masking and verifying identities. Later versions should consider this, though it might require a shared setting to control the amount of normalization.

Limited Configurability

When it comes to making people indistinguishable, any unique or even just differing behavior is potentially dangerous. For example if everyone uses the correct timestamp on their commits and one person always sets it to a specific value or the first second of the given day then this behavior alone can be sufficient to link multiple commits to the same author.

This means that an anonymization system can only offer configuration options that are synchronized with all other users of the system in a given repository to ensure that they can't be used to distinguish them.

Support for Operating Systems other than Linux

While other operating systems can of course be supported given enough work, this is also considered to be out-of-scope for this project.

The goal of this project is to prove the viability of the proposed system by implementing a basic version for typical Linux installations. Porting this system to other environments (Windows, MacOS, BSD or less typical Linux installations) or even integrating it into git directly should then be fairly simple.

Only the Identity of the Committer can be verified

With the way git signs commits only one signature can be applied to a commit and this signature can only be created once the committers identity and commit time are known.

In cases where committer and author are identical, this is not an issue however when a commit is later changed (for example to modify the commit message or the parent commit) the committer might well be different from the author. In this case the author information becomes a claim of the committer that can no longer be cryptographically verified.

There are proposed systems [18, 26] that allow for multiple detached signatures on each commit, however these also obviously can't cope with changes to the file tree, message or author and committer information as they also sign the entire commit.

In typical scenarios this is not too much of a problem but it is important to be aware of the fact that only the committer can be verified and information about the author is only a claim from the committer. This is despite the fact that git rarely displays information about the committer in everyday use unless configured otherwise.

6. Potential Solutions

6.1. Protecting Identities

Commits contain information about the person creating the commit in two distinct fields (`Author` and `Committer`). These fields contain a name and email address combination as well as information about the time and timezone when the commit was created or authored respectively. To control who this information is provided to, the name and address in this field need to be masked. There are multiple ways of achieving this, but they have in common that the masked value needs to be different every time even if the original value is the same. This is to ensure that the masked name is not merely a pseudonym and does not allow commits from the same person to be linked together.

What these systems have in common, is that they don't mask the signature, which includes information about the key and therefore identity that created it. Signatures would either have to be protected separately or removed.

Simply Encrypting Identity Information

The simplest solution would be to encrypt the authors name using a shared key. This would be a very fast and scalable solution, however only a limited number of keys could be used reducing the granularity with which information about the actual author can be shared. This also makes it more difficult to reveal the author of individual commits after their creation, as all identities encrypted under the same key would have to be revealed as well.

Uniqueness of masked identities could easily be achieved using random (or at least unique) initialization vectors.

Unique Pseudonyms and a List of Mappings

Another option would be to pick random pseudonyms and distribute them in one or multiple lists of mappings. These lists would contain the random pseudonym and any information that should be shared with anyone that has access to the respective list.

This provides a lot more granularity on which information to share and allows attributes about the identity to be shared.

Lists would have to either be shared out-of-band (using blockchains, databases, mails, ...) or encrypted and stored in the repository.

However, anyone capable of adding information to the list would also be able to add or modify information about identities other than their own.

6.2. Facilitating Signatures

Signatures cause a different set of challenges.

- Signature keys must be available to verify signatures.
- There needs to be a chain of trust from an already trusted key to the signature key. The initial set of keys is typically trusted manually.
- Signature keys must be connected securely to attributes of their owner.

Shared Signature Keys

The simplest signature system to prove membership in a certain group is sharing a signature key.

One of the advantages of this system is that signature verification on commits becomes very easy even for users that don't use the anonymization system. They can simply be provided with the signature key that is shared by all members allowed to perform a certain action (such as creating a commit) and use existing git and gpg tooling to verify the signature.

One of the problems with this system is that git only supports a single signature on commits. This means that if the user has multiple roles and therefore multiple shared signature keys, they must choose one of those roles for each commit manually.

Another problem is how to securely make this key available. As the number of members of a certain role increases, more and more copies of the key will be created, making it more likely that one of the copies will be compromised. For larger organizations this would mean sharing the "member of the organization" key with hundreds or thousands of people making it almost certain that it will be compromised. To counter this, access could be limited by providing the key in tamper-resistant hardware or as a networked signing service rather than directly.

Another big problem, that is inherent to digital signature schemes without notarized timestamps or freshness guarantees on signatures is once a key gets compromised all signatures ever created using it must be considered invalid unless it can be proven, that they were created before the compromise [24].

Separate Signature Keys per User ID

Another option is to generate new signature keys for each masked identity (and therefore effectively for each signature) removing any links between the commits.

These automatically generated signature keys would then be shared together with the other identity information as described above.

However, this would make them prone to manipulation by anyone with write access to the respective list and raises the question of how these keys can be validated to belong to someone with the claimed name or attributes.

The solution would be to certify (sign) them using long term keys that are already trusted. The certifying key would confirm that a certain attribute belongs to the holder of a certain signature key. These attributes and certifications could then be made available independently of one another providing fine-grained control about which information to share with whom.

The simplest option would be using existing long term gpg keys to sign anonymous signature keys thereby confirming to anyone that gets the certification that the signature key belongs to the same person as the long term key.

Another option would be to sign them using a shared key (which could then be stored offline with ephemeral keys generated and certified in advance). This shared key could then be used to verify, that the owner of a certified key belongs to a certain role or group. These assertions could in turn be verified without needing to know anything about the person that signed the commit.

6.3. Combined Solution

Another option is to base the anonymization system on the signature keys rather than trying to somehow get a workable signature system on top of the pseudonym system. To prevent signature keys from becoming links between different commits, they will have to be ephemeral and can only be used for one commit. At this point there will be pretty much a one to one mapping between pseudonyms used in author and committer identities and signing keys used to sign commits, which creates unnecessary overhead. Instead one could simply use references to the signing key (specifically its fingerprint or key id) as a pseudonym for the real identity behind the signature key. The signature key has to be available to anyone attempting to verify a signature anyways and at this point it does not contain any usable information about the real identity so it can simply be added to the repository and distributed alongside it.

Now two very useful features of GPG come into play. The first feature is that gpg keys can contain more than one user id and that each user id can be certified independently (and by multiple keys). This means that the key holder can create separate user ids for each attribute of themselves that they might want to reveal later, add them to their key and have each attribute certified individually by others. The other feature is that keys consist of multiple packets that can be separated and recombined securely as needed, which allows the key holder to separate the individual user ids they created earlier and distribute them independently. If the key holder wants to inform someone about a certain attribute of themselves, all they have to do is make the relevant user id and it's associated signatures available to them. It can then be verified that the user id belongs to the key by verifying its self-signature and then validating the certifications to prove that a third party attested to this attribute.

This means that the key holder can reveal individual attributes such as their role in the project to outsiders without revealing their full identity.

There are a multitude of options when it comes to distributing the user id packets. They range from databases with or without authentication to mails or various forms of encrypted publishing. For the initial version of this project the packets will simple be encrypted with a single shared key and stored in the repository.

For simple systems where a group wants to reveal their identities to each other while only revealing selected attributes to the public, this approach will be sufficient. Of course more flexible approaches such as asymmetric cryptography and different levels of access could be implemented. Storage systems outside the repository would allow data to be deleted more effectively later.

6.4. Verdict

The combined solution provides cryptographically verifiable paths from certification authority keys that the user manually trusts to each ephemeral identity, its attributes and finally the signed commits.

Attributes can be published in a number of ways and at any time as desired and signatures can only be forged by certificate authorities creating new keys and certifying the attributes they can certify.

The mapping between anonymous identities in author and committer fields and their associated real names can be handled as a byproduct of the signature scheme and can similarly be revealed as desired at any point in time.

7. Implementation

The proposed system will be based around GPG and use other existing functionality where possible. GPG already provides a versatile system of certifications and trust and allows for individual parts of public keys to be distributed independently. Combined with the existing integration into git and its useful features (described in section 7.3) this will be the foundation of this project.

7.1. GPG Key Properties

Ephemeral Keys

To avoid linkability between commits, each of them will be signed with an individual gpg key with author and committer identities set to the keys fingerprint or key id.

User IDs

One of the main feature used in this system is that GPG keys can contain multiple User IDs that can be used independently. This will be used by adding one user id per assertion about themselves that the key holders want to make.

Examples:

- Full name and e-mail address
- One for each role in the current project (Maintainer, Developer, Release-Manager, Senior-Developer, Sysadmin, Project-Lead, ...)
- Membership in organizations (Eclipse Foundation, ACME Corp., ...) (potentially including roles therein)
- Assertions about usernames on common platforms (Github, Gitlab, Bugtrackers, ...)

By selectively sharing or publishing these User IDs the keyholder can then decide, which assertions they want to make known to whom.

Certifications

At this point the assertions made using User IDs are nothing more than claims.

To make them cryptographically verifiable, a system of Certificate Authorities (CA) is proposed. These will be regular GPG Keys and can certify User IDs on other keys using the regular mechanisms of GPG. Contrary to the model of CAs employed with the X.509 system, CAs are deemed authoritative only for User IDs that match one of their own User IDs. The CA-UserIDs in turn fall under the regular trust model of GPG and must be considered trusted in the users regular keyring or imported manually.

This allows a regular gpg key trusted in the users normal keystore to certify UserIDs on ephemeral keys by the same user, binding them together and creating self-certifications. To decrease the number of keys and User IDs needed a pattern matching could be defined instead of a strict matching. To avoid misuse, transitive certifications inside the ephemeral keyring will not be allowed and CAs must always be from the users normal keyring.

Potential Issues

- An author could claim multiple contradictory assertions if the associated CAs cooperate. For example, they could claim multiple full names and have them certified by the legitimate holders of these names. The effects of this are unclear.
- Lots of keys need to be generated (one per commit), which might cause issues with performance and availability of entropy.
- It might be possible to brute-force User IDs based on their certifications. This can be solved easily by distributing certifications and User IDs together and treating both as sensitive.

Implementation State

Currently, trusted CAs must be imported manually into the system. Future versions could import them automatically from the users regular gpg keystore.

7.2. Synchronizing Key Material

The aforementioned information about keys must be distributed and available in combination with the repository. While some parts of the key must be available to everyone (public parameters of the actual key material), others should be made available selectively (User IDs and their certifications).

Many options exist for this including the following. Each has their benefits and drawbacks:

- A folder at a predefined location in the file tree of the repository
- Objects in the repositories object storage alongside commits, tags, trees and binary blobs for files.

- Out-Of-Band using central systems such as keyservers, LDAP, databases, ...
- Out-Of-Band using unmanaged systems such as mail

Part of the information can of course also be encrypted to further limit the number of entities that can use it.

The most practical solution seems to be to include the information at a predefined location in the repository and encrypt it symmetrically to limit its recipients.

Future versions will likely offer options for out-of-band synchronization using central systems such as databases where access can be limited and that offer more realistic options for deleting data than git repositories.

7.3. Useful Features and Tools

The following special features of git are used:

Mailmaps

Git offers a feature called mailmaps, where if a file called `.mailmap` exists in a repository, it can be used to change the displayed name and e-mail address of authors and committers without rewriting older commits. A central mailmap file outside of the repository is also supported.

This is used to translate from pseudonyms to real names by maintaining a mailmap file outside the repository that gets updated whenever new pseudonyms become known.

Git-Hooks

Git allows scripts to be defined that are executed during certain events. Such events include `commits`, `pushes`, `pulls` and more. These are used to analyze and process incoming commits, update the list of pseudonyms and to generate new identities for the next commit.

Git Plugin System

Git offers a simple plugin system where executable files following a certain naming scheme can be stored on the systems path and are then made available as subcommands in git.

When the user then runs a command such as `git anon update` git will execute the file `git-anon` with the argument `update` and provide a suitable environment.

7.4. Architecture

Git-Anon is implemented on top of an existing git installation rather than being integrated into it. This is to simplify development and reduce maintenance effort.

Cryptographic operations are performed using an integrated gpg library as the regular implementation scales badly with the number of keys in the keystore.

The system is written in Python due to its memory safety and suitability for command line applications. A Rust implementation was also considered.

7.5. Distribution

The project is distributed as a source distribution and as a wheel for faster installation through the canonical Python Package Index [14]. The source distribution contains the source code that a user would need to build and install the software themselves. This includes build scripts and source code for compiled languages such as C or Rust if the project uses native code segments for performance. Wheel distributions on the other hand contain these parts as compiled binaries and structure the python files in a way that speeds up installation. While source distributions are platform-independent, wheels containing native code need to be compiled for each architecture. Since git-anon only uses Python the resulting wheels are platform-independent as well.

Both source distributions and wheels are compressed folder structures that contain interpretable and readable Python code combined with metadata.

7.6. Storage Architecture

Git-Anon uses a content addressed storage system similar to the one used in git to quickly and efficiently access keys and information about them and to prevent conflicts when concurrent additions to the keystore occur and need to be merged later.

All key information that should be shared is stored in a subfolder of the repository and synchronized along with it. To avoid collisions and allow quick access to individual keys, each key is stored in a folder, whose location is derived from its keyid and fingerprint. The public key is stored in a file called `primary_key.pub`, while its user ids and their associated signatures are stored in individual files named after their hash and information about whether they are encrypted.

The advantages of this content addressed system are that finding a key takes only as long as it takes the file system to list and return a few small files in a folder at a known location. This information is typically cached by the operating system and further internal caching is planned for future versions of git-anon.

Scalability of GPG Keystores

Initial versions of git-anon relied on the standard gpg installation that comes preinstalled on most Linux distributions for signing, verification and local key handling. However this quickly causes problems as some part of importing a key into a gpg keystore seems to depend on the number of keys already present in the keystore.

While the first few keys could be imported in milliseconds, key number 1000 took about 2 minutes to import, making this far too slow for a system that uses a different signing key for each commit.

This slowdown might be caused by operations related to the Web Of Trust, validation of signatures on existing keys, comparing the new key to existing ones or simply a slow storage implementation. The exact reasons for this slowdown were not analyzed further as importing thousands of keys into gpg should be rare for regular use and the problem for git-anon can simply be solved by replacing gpg with a compatible tool for signature verification and signing that can make use of the existing storage architecture used by git-anon.

8. Potential Extensions

8.1. Forcing Escrow of the Real Identity to the Server

In cases where it is necessary to enforce that any commit added to the repository be traceable to a civil identity, a git hook on the server could be used. This hook would verify that any new commits introduced to the system are signed and that sufficient information is available to cryptographically link the identity of the committer to a civil identity from a known list of authorized persons. The necessary information could be provided to the server in a multitude of ways and might be retained by it for a predetermined time or forever.

Depending on the configuration this could also be used to ensure that the identity information is available to other team members (for example by forcing it to be encrypted with a shared key and added to the repository).

Of course this could also be very useful independent of anonymization schemes.

8.2. Replacing/Augmenting Connection Based Authentication with Content Based Authentication

Signed commits (whether anonymized or not) could also be used to replace or augment the authentication towards the git server. While this traditionally uses connection based authentication, one could imagine only accepting commits signed by authorized team members instead.

One of the important challenges here is determining whether the committer actually intended to push this commit and in particular whether he intended have a particular branch point to it.

A commit that deletes all data on restart might be signed but only intended for local use. Similarly, this change might be appropriate on a branch that gets deployed in a testing environment, but would have disastrous consequences in productive ones.

One potential solution to this would be the feature of singed pushes as offered by git, however this requires further investigation.

8.3. Mail Relay for Anonymous Identities

The anonymous identities could be configured to contain an email address of the format `<keyid>@<relay-domain>`, where `<keyid>` is the hexadecimal, 64 bit key id for the identity (or the full fingerprint). This would allow users or systems that are unaware of the anonymization system to contact the owner of the identity as they usually would.

A relay service could then be operated at `<relay-domain>`, which would extract the keyid from the local part of the email address and check if it can infer the mapping to a real identity (using the anonymization system). If the identity can be inferred, it could forward the message to the actual email address. If the identity is unknown, the system can either accept and store the message until the identity becomes known or refuse it temporarily or permanently. A temporary refusal might be useful here because for messages sent shortly after a commit is published it is fairly likely that the relay service would not have received information about the identity mapping yet.

One concern here would be that tracking elements in mails could be used to detect if, when and where the mail was opened, potentially revealing the real identity. Another concern would be revealing whether the relay service is aware of the real identity and therefore whether the author has made this mapping known to the service.

9. Evaluation

Git-Anon meets all requirements defined in chapter 4, however some parts are still a bit rough around the edges and some optional parts or convenience features are not implemented yet.

Inability for External Observers to Group Commits from the same Author

Each commit is signed with a unique key that shares no key material with any other key and is completely independent.

Public keys contain no information connected to the user and are generated with a predefined set of preferences and algorithms that are common to all keys generated by git-anon.

Attributes and Roles of the keyholder are signified through User IDs. These can be stored encrypted and/or unencrypted and consist of no useful information apart from the UTF-8 encoded string that represents the attribute and a signature that binds it to the public key material. Encrypted attribute packets are padded to a uniform length of 4096 bytes to prevent correlations based on their length.

All timestamps in signatures and keys are set to a fixed date to avoid revealing that a set of keys might have been generated at the same time.

The only differentiating factors between git-anon identities are their publicly revealed attributes and the number of encrypted attributes.

The latter can be counteracted by agreeing on a number of attributes with the other project members. A future version of git-anon might generate decoy-packets at the cost of more storage space. Unfortunately this would also prevent users from adding more encrypted attributes later or having more than the predefined number of encrypted attributes.

Backward Compatibility and Interoperability with Unaware Clients

Git-anon creates standards compliant commits. Users unaware of the system will simply see authors with names such as `ANON 1234567890ABCDEF` and associated email addresses such as `1234567890ABCDEF@git-anon` that created one commit each. Signatures will either be displayed as "No Signature" or as a signature from an unknown key that cannot be obtained easily. Due to the current storage format, verifying signatures without git-anon would be quite tricky. Users would either have to install git-anon or manually combine and verify the relevant key parts themselves.

Anonymization is Reversible

Anyone with access to an unencrypted attribute package can choose to publish it. Each attribute package contains either a self-signature or a certification. A self-signature proves that the keyholder claimed this attribute about themselves whereas a certification proves that the holder of the certification key claimed this attribute about the keyholder. Attribute packages can be generated and published after the commit was created.

Git-Tools Show the Real Author if the Corresponding Mapping is Known

Git-Anon uses the mailmap feature of git to tell it which anonymous identity (see above) corresponds to which real name for a person. This real name is derived from the key with the corresponding key id. In particular the user id with the “primary” flag is chosen. This mapping is typically intended for email address changes and used by git everywhere an identity is shown.

Does not Clutter Existing GPG Configuration

Currently, git-anon does not use the existing gpg system at all. Future versions might interact with it to find trusted keys or certify the users name with their existing gpg key.

Decentralized

Git-Anon supports completely decentralized operation. All data is stored in the repository where it can be shared together with the repository in any way that keeps commit hashes intact.

The certification system can be decentralized if either the CA certificates are shared directly or each user has access to the necessary ones at least once in a while to have batches of pregenerated attributes signed.

Certification system

With Git-Anon teams or organizations create gpg keys that are used to certify attributes about others. These keys are manually imported and trusted in a git-anon installation and from then on all attributes that are certified by these keys are considered valid and marked as trusted. Certifications are only considered if the certification key has a user id that exactly matches the attribute to be certified to prevent a CA from certifying attributes for which they are not authoritative. Future versions might allow matching based on patterns to create a more flexible setup.

The goal with this system is to allow users to pregenerate a large batch of keys, send them off for certification and then use them offline as needed. Currently, this process is not implemented fully and realistically the CA key will be shared among everyone with the attribute. Future versions of git-anon could use certification servers or processes whereby keys are signed in bulk by a separate entity that holds the key.

A users regular gpg key could also be used to sign their name. In combination with trusted keys imported from the regular keyring this could be used to bootstrap trust in the system.

10. Conclusion

Git-Anon is a system that allows a closed group of authors to use git relatively normally, sharing their identity and what they are doing with each other without having to reveal their identity publicly and without even the ability for external observers or other contributors to learn which commits were made by the same person.

While providing a good level of anonymity, git-anon also allows both team members and external observers to verify that signatures on commits were created by someone with a set of roles or attributes.

It allows the signatory to decide which of their roles and attributes they want to share and whether they want to share them only within their closed group or publicly. It even allows additional information to be revealed later without modifying the commit and to a degree information can even be retracted later.

This unique set of properties and options allows developers to work on controversial projects with less fear of repercussions and helps solve legal issues around the concept of a public ledger containing almost every action an open source developer has ever taken.

Git-Anon is available as an open source project on both Github and PyPi (see Appendix B for details).

11. Outlook

A signature scheme that allows flexible decisions about which information to reveal about the signer including the ability to reveal part of the information long after the signature was made could have many uses outside of code signing as well.

In a lot of cases the entity verifying a signature does not need to know the identity of the person signing the document. They only need to know certain attributes or roles of the signatory. Examples include common eID use cases such as verifying someones age for age restricted actions, their status as a student for specific benefits or that they are allowed to drive vehicles of a certain category. While there are existing solutions for most of these problems, many consist of presenting documents that contain much more information than necessary.

A similar system to the one used with git-anon could be used to verify roles or attributes about a person while knowing or storing as little information as possible and therefore reducing both the risk of the information leaking as well as the security measures needed to ensure its protection.

A. Example of a GPG Key Consisting of Multiple Packets

```
:public key packet:
        version 4, algo 1, created 1570280851, expires 0
        pkey: [[4096 bits],[17 bits]]
        keyid: 2DD7C9487DFCC04D
:user ID packet: "Erik Escher <erik@erikescher.de>"
:signature packet: algo 1, keyid 2DD7C9487DFCC04D
        version 4, created 1570300511, md5len 0, sigclass 0x13
        digest algo 10, begin of digest 04 dd
        hashed subpkt 27 len 1 (key flags: 01)
        hashed subpkt 9 len 4 (key expires after 2y0d0h0m)
        hashed subpkt 11 len 4 (pref-sym-algos: 9 8 7 2)
        hashed subpkt 21 len 5 (pref-hash-algos: 10 9 8 11 2)
        hashed subpkt 22 len 3 (pref-zip-algos: 2 3 1)
        hashed subpkt 30 len 1 (features: 01)
        hashed subpkt 23 len 1 (keyserver preferences: 80)
        hashed subpkt 25 len 1 (primary user ID)
:signature packet: algo 1, keyid 5E5CCCB4A4BF43D7
        version 4, created 1570286294, md5len 0, sigclass 0x13
        digest algo 8, begin of digest 65 59
        hashed subpkt 5 len 2 (trust signature of depth 1, value
            60)
:public sub key packet:
        version 4, algo 1, created 1570281070, expires 0
        pkey: [[4096 bits],[17 bits]]
        keyid: DA356DBF1F24EA82
:signature packet: algo 1, keyid 2DD7C9487DFCC04D
        version 4, created 1570281070, md5len 0, sigclass 0x18
        digest algo 10, begin of digest 55 09
        hashed subpkt 27 len 1 (key flags: 02)
        subpkt 32 len 563 (signature: v4, class 0x19, algo 1,
           digest algo 10)
```

Listing A.1: Example of a gpg key consisting of multiple packets.

Output as produced by `gpg --export <key-identifier> | gpg --list-packets`. Shortened and formatted for presentation and clarity.

B. Open Source Release of Git-Anon

Git-Anon is available as a Python source distribution and wheel package for installation using pip at `https://pypi.org/package/git-anon`.

The associated source code is available at `https://github.com/erikescher/git-anon`.

Hashes for the initial commit and version 0.1 are available below and all commits are signed using gpg.

Object	Hash
Commit for v0.1	`cae0e0ba939486850836ceeab0752e8855065df3`
git-anon-0.1.tar.gz	`6b2044c9371bb95075ac56cb827ca474e575d3845`
git_anon-0.1-py3-none-any.whl	`1efe49a079917ee04a85e30b92a43e77e4a1d5a3`
GPG Fingerprint	F961 186F 6EE1 85D7 732F 12FA 2DD7 C948 7DFC C04D

C. List of Figures

D. List of Listings

E. Bibliography

[1] Andrew Ayer. *git-crypt - transparent file encryption in git*. 2020. URL: https://www.agwa.name/projects/git-crypt/ (visited on 2020-07-24).

[2] Andrew Ayer. *Ubuntu Manpage: git-crypt - transparent file encryption in Git*. Ed. by Caconical Ltd. 2020. URL: https://manpages.ubuntu.com/manpages/focal/man1/git-crypt.1.html (visited on 2020-07-24).

[3] Andrew Ayer and other Contributors. *GitHub: git-crypt - transparent file encryption in git*. Version 7c129cdd3830a55a8611eecf82af08cd3301f7f2. 2020-04-28. URL: https://github.com/AGWA/git-crypt (visited on 2020-07-24).

[4] Bundesamt für Sicherheit in der Informationstechnik. *BSI - Bundesamt für Sicherheit in der Informationstechnik*. 2020. URL: https://www.bsi.bund.de/ (visited on 2020-07-31).

[5] Cambridge University Press. *Anonymity - Cambridge English Dictionary*. URL: https://dictionary.cambridge.org/de/worterbuch/englisch/anonymity (visited on 2020-08-04).

[6] Scott Chacon, Jason Long, and Other Contributors. *Git*. Version 7714726. 2021-03-03. URL: https://git-scm.com (visited on 2021-03-13).

[7] Scott Chacon, Jason Long, and Other Contributors. *Git - Reference*. Version 7714726. 2021-03-03. URL: https://git-scm.com/docs (visited on 2021-03-13).

[8] Scott Chacon and Ben Straub. *Pro git: Everything you need to know about Git*. English. Second. Apress, URL: https://git-scm.com/book/en/v2.

[9] Debian Project. *Popularity Contest Statistics – Debian Quality Assurance - qa.debian.org/*. 2020-10-09. URL: https://qa.debian.org/popcon-graph.php?packages=git+mercurial+subversion+bazaar+cvs&show_installed=on&show_vote=on&want_legend=on&want_ticks=on&from_date=&to_date=&hlght_date=&date_fmt=%25Y-%25m&beenhere=1 (visited on 2020-10-09).

[10] Gitlab Developers. *Repository mirroring - Gitlab*. Version ae7269d1eac6d4ab2970a740797cebbe9328ffd1. 2020-07-03. URL: https://docs.gitlab.com/ee/user/project/repository/repository_mirroring.html (visited on 2020-07-24).

[11] Thomas Durieux et al. *Anonymous Github*. 2020-06-10. URL: https://anonymous.4open.science/ (visited on 2020-08-06).

[12] Thomas Durieux et al. *Anonymous Github.* Version add12b45ecaf3bfa41e44cd283b1705ccdd3acee. 2020-06-10. URL: https://github.com/tdurieux/anonymous_github/ (visited on 2020-08-06).

[13] Hal Finney et al. *OpenPGP Message Format.* RFC 4880. 2007-11. DOI: 10.17487/RFC4880. URL: https://rfc-editor.org/rfc/rfc4880.txt (visited on 2020-07-30).

[14] Python Software Foundation. *PyPI - The Python Package Index.* 2020. URL: https://pypi.org (visited on 2021-03-11).

[15] g10 Code GmbH. *The GNU Privacy Guard.* 2020-07-09. URL: https://gnupg.org (visited on 2020-07-30).

[16] Governikus GmbH & Co. KG. *Beglaubigung OpenPGP-Schlüssel.* 2020. URL: https://pgp.governikus.de/pgp/ (visited on 2020-07-31).

[17] Governikus GmbH & Co. KG. *Governikus KG.* 2020. URL: https://www.governikus.de/ (visited on 2020-07-31).

[18] Owen Jacobson. *Notes Towards Detached Signatures in Git.* Initial publishing date likely earlier but unknown. See also : https://github.com/grimoire-ca/bliki/commit/ee3370cfa5cb17261f722c501c94edf0a431f91d. 2020-01-30. URL: https://grimoire.ca/git/detached-sigs/ (visited on 2020-07-28).

[19] Jason Kulatunga. *gitMask - Develop Anonymously.* 2020. URL: https://www.gitmask.com/ (visited on 2020-07-24).

[20] Andreas Pfitzmann and Marit Hansen. *A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management.* http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf. v0.34. 2010-08. URL: http://dud.inf.tu-dresden.de/literatur/Anon%5C_Terminology%5C_v0.34.pdf (visited on 2020-08-05).

[21] Proton Technologies AG. *Everything you need to know about the "Right to be forgotten".* GDPR.eu. Section: GDPR Overview. 2018-11-05. URL: https://gdpr.eu/right-to-be-forgotten/ (visited on 2020-10-09).

[22] Pete Resnick. *Internet Message Format.* RFC 2822. 2001-04. DOI: 10.17487/RFC2822. URL: https://rfc-editor.org/rfc/rfc2822.txt (visited on 2020-10-09).

[23] Synopsis, Inc. *Compare Repositories - Open Hub - www.openhub.net/.* Compare Repositories. URL: https://www.openhub.net/repositories/compare (visited on 2020-10-09).

[24] Peter Todd. *Solving the PGP Revocation Problem with OpenTimestamps for Git Commits.* 2016-09-26. URL: https://petertodd.org/2016/opentimestamps-git-integration (visited on 2021-03-13).

[25] Sebastien Varrette. *Using Git-crypt to Protect Sensitive Data.* 2018-12-07. URL: `https://varrette.gforge.uni.lu/blog/2018/12/07/using-git-crypt-to-protect-sensitive-data/#removing-a-collaborator-from-the-vault` (visited on 2020-07-24).

[26] Lance R. Vick et al. *Github: Git Signatures.* Version `979f207a1c1342b9342aa58be914fc51a0c62f87`. 2018-10-03. URL: `https://github.com/hashbang/git-signatures` (visited on 2020-07-28).

[27] Wikipedia contributors. *Anonymity — Wikipedia, The Free Encyclopedia.* 2020. URL: `https://en.wikipedia.org/w/index.php?title=Anonymity&oldid=969056091` (visited on 2020-08-04).

[28] Wikipedia contributors. *Pseudonymity — Wikipedia, The Free Encyclopedia.* 2020. URL: `https://en.wikipedia.org/w/index.php?title=Pseudonymity&oldid=967108928` (visited on 2020-08-04).

[29] Wikipedia contributors. *Right to be forgotten.* In: *en.wikipedia.org/.* Page Version ID: 979590717. 2020-09-21. URL: `https://en.wikipedia.org/w/index.php?title=Right_to_be_forgotten&oldid=979590717` (visited on 2020-10-09).

[30] Wiktionary. *anonymous — Wiktionary, The Free Dictionary.* 2020. URL: `https://en.wiktionary.org/w/index.php?title=anonymous&oldid=59666697` (visited on 2020-08-04).

[31] Wiktionary. *pseudonymous — Wiktionary, The Free Dictionary.* 2020. URL: `https://en.wiktionary.org/w/index.php?title=pseudonymous&oldid=59415419` (visited on 2020-08-04).