Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor thesis

# Privacy implications of exposing Git meta data

presented by

Arne Beer
born on the 21th of December 1992 in Hadamar
Matriculation number: 6489196
Department of Computer science

submitted on May 2, 2018

Supervisor: Dipl.-Inform. Christian Burkert
Primary Referee: Prof. Dr.-Ing. Hannes Federrath
Secondary Referee: Prof. Dr. Dominik Herrmann

# Abstract


'As long as we use modern technologies we always expose data about ourselves'. This is a statement I truly believe in.

Recent events, such as the Facebook scandal in which the data of several million people has been exposed to a consulting company [1] show how large amounts of data can be abused to extract valuable knowledge and used for malicious purposes.

This thesis aims to give an example of how much information can be exposed by simply using the popular version control system Git. Simple meta data such as UNIX timestamps and email addresses might be enough to extract sensitive information about Git users or organizations using Git. This paper covers the whole process of gathering the data from a vast amount of git repositories through to preprocessing and interpreting the results of the analyses. With this thesis I hope to raise the awareness how dangerous it can be to expose even simple meta data and that it might be used maliciously.


[1] 'Facebook scandal hits 87 million users' BBC.com, http://www.bbc.com/news/technology-43649018 (accessed, 24.04.2018)

*Even if you're not doing anything wrong, you are being watched and recorded.*

*Edward Snowden*

# Contents

# Acronyms

**API** Application Programming Interface

**CPU** central processing unit

**FS** file system

**GB** Gigabyte

**HTTP** Hypertext Transfer Protocol

**JSON** JavaScript Object Notification

**ORM** Object-Relational Mapping

**OS** Operating System

**SHA-1** Secure Hash Algorithm 1

**SSH** Secure Shell

**SQL** Structured Query Language

**URL** Uniform Resource Locator

**UTC** Coordinated Universal Time

**VCS** version control system

# Chapter 1

# Introduction

Git is a code version control system which is used by most programmers on a daily basis these days. According to the Eclipse Community Survey about 42.9% of professional software developers used git in 2014 with an upward tendency [7]. It is deployed in many if not most commercial and private projects and generally valued by its users. It allows quick jumps between different versions of a project's code base and to manage and merge code from different sources to one upstream.

Several million users send new commits to their Git repositories every day. On Github alone, the currently biggest open source platform, there exist about 25 million active repositories, a total of 67 million repositories and about 24 million users [3].

Some well known projects and organizations use Git, for example Linux, Microsoft, Ansible and Facebook [3]. Every repository contains the complete contribution history of every contributing user. Each commit contains the full directory structure, a link to a blob for every file, a timestamp, a commit message from the author and more additional metadata.

This raises the question how much information is hidden in the metadata of a Git repository and which attack vectors could be introduced by mining this information, regarding a contributer or the owner of the repository.

The newly gained knowledge could be utilized by employers to spy on their employees. It could be used by an unknown attacker who aims to obtain sensitive information about a company and its employees trough their open-source projects. It is even possible that a privat person wants to monitor another person that regularly contributes to open-source repositories.

As there have not been any papers published about this specific topic or at least no public paper and Git plays such a crucial role in todays information technology, I want to investigate and evaluate this potential threat.

## 1.1 Attack models

This section introduces three attacker models and their respective goals.

The required data to achieve and evaluate the goal will be listed and explained in the process. These attack models serve as a guideline for the data aggregation process, which will be covered in the next chapter.

### 1.1.1 The Employer

This attack model deals with the scenario of an employer, which wants to monitor their employees. The attacker's motivation is to spot irregularities in working behavior as well as unmotivated or unproductive employees.

**Productivity of Employees**

Ensure employees produces enough code. For this purpose the changes in lines of code over a specific time span will be evaluated.

Required data:

- Commits of the employer's repositories.
- Commit timestamps
- Additions of each commit
- Deletions of each commit

**Compliance of Working Hours**

Check if an employee is productive in the defined working hours. This is especially useful to supervise employees, which work remotely.

Required data:

- Commits of employer's repositories.
- Commit timestamps

**External Projects during Working Hours**

Inspect if an employee is working on an external project during working hours. This only works if the employer has access to the external project, for example open source projects.

Required data:

- All commits of any available repository to come into question
- Commit timestamps

**Code Quality Between Employees**

Compare the quality of contributed code between different employees. With this metric the quality of an employee could be measured. To compare the quality we would need an external tool for code analysis.

Required data:

- Commits of the employer's repositories.
- Complete commit patch
- Commit timestamps

### 1.1.2 The Individual

This scenario describes a single person, which wants to harm, monitor or gain information about an open source developer.

An example goal of an attacker could be to either stalk the victim, harm him in any way or to manipulate him or one of his acquaintances. The motivation of this attacker is mostly personal and on an emotional level.

Another non emotional attacker could be a robber trying to find the perfect time window to rob a house or the tracking of a high profile target.

A third attacker could be a headhunter which tries to get information about the skills and reliability of a developer.

**Sleeping Rhythm and Daily Routine**

Learn about the persons sleep rhythm and obvious patterns in his daily routine. This attack aims to understand and predict the victim's behaviour.

Required data:

- Victim's commits.
- Commit timestamps.

**Personal Relationships to Various Programmers**

Show how many people work together (Same time window). Time correlation.

Required data:

- Victim's commits.
- Commit timestamps.

**Sick Leave and Holiday**

Detect breaks in his typical work behaviour, which could represent holiday breaks or sick leave. This attack could give information about whether a developer is at home right now or if he tends to be sick alot.

Required data:

- Victim's commits.
- Commit timestamps.

### 1.1.3 The Industrial Spy

This attack model covers the scenario of an external person, which wants to gain as much private or malicious information about a company as possible. The attacker's motivation is either to harm the company, gain an advantage as an competitor or in the stock market or to sell secret information to a third party. This attack vector only works if the targeted company is providing their product or at least parts of their product as open-source software.

**Company Employees**

The most important target is to detect the company's employees as three other goals for this attacker model depend on this information. Another motivation could be to detect company members for further social engineering attacks or to headhunt the company's employees.

Required data:

- All commits of the company's repositories.
- Commit history graph.
- As much meta data about the company's employees as possible for evaluation.

**Employee History**

Detect the timespan for which an employee worked at a given company. This could be interesting, as it shows the average employment duration and the employee amount over the history of the company, which could be an indicator of its current financial growth. Social engineering or headhunting could be a motivation here as well.

Required data:

- Company Employees
- Commit timestamps of the company's repositories

**Global Workforce Distribution**

Detect the timezone of all employees and create a global distribution graphic by timezones. This graphic allows you to guess the location of a company's workforce. It is also possible to create this statistic for all contributer, which could show a trend which countries or at least continents are interested the most for the company's product.

Required data:

- Company Employees
- All Commits
- Commit timestamps of the company's repositories

### Internal Team Structures

Try to predict different teams, the role of each team and the respective team members.

Required data:

- Company Employees.
- Commits of the employer's repositories.
- Commit history graph.

# Chapter 2
# Data

## 2.1 Git

This chapter introduces the version control system (VCS) *Git*, as it plays a fundamental role in this thesis. In the following the most relevant parts of Git will be explained such as user roles, technologies and internal data representations. I will also talk about the current cases of application and some interesting scenarios which might be interesting for this thesis.

### 2.1.1 Introduction to Git

At its core, Git is a tool, which is used to manage different versions of files in a specific directory. A directory managed by Git is called a *repository*. Each version of the project is saved as a so called *commit*, which represents a specific state of all files and directories in the project. Users are able to meticulously specify files or changes in files that should be added to a commit. For instance a developer can only commit a subset of the changes, which were applied to a repository. By doing so, one can split a large set of possibly completely unrelated changes into several commits, where each commit in itself forms a set of logically related changes. After creating several commits Git is capable of showing the exact changes between different commits, which is called a *diff* and jumping between different versions of the project, which is called a *checkout*.

Git is the currently most popular tool to control a project's code with a still upward tendency [7]. It enables to work with multiple developers on a single code base, as it provides several different techniques, namely the *history tree*, the *branch* and the *merge*. The commit history of Git is internally represented as an directed, non-cyclic, connected graph of commits or a tree. The commits act as *nodes* and the connection to their parent commits as *edges*. Every time two edges leave a single node, a new branch is created. Git provides the feature to name branches, whereas the main branch is per default named *master*.

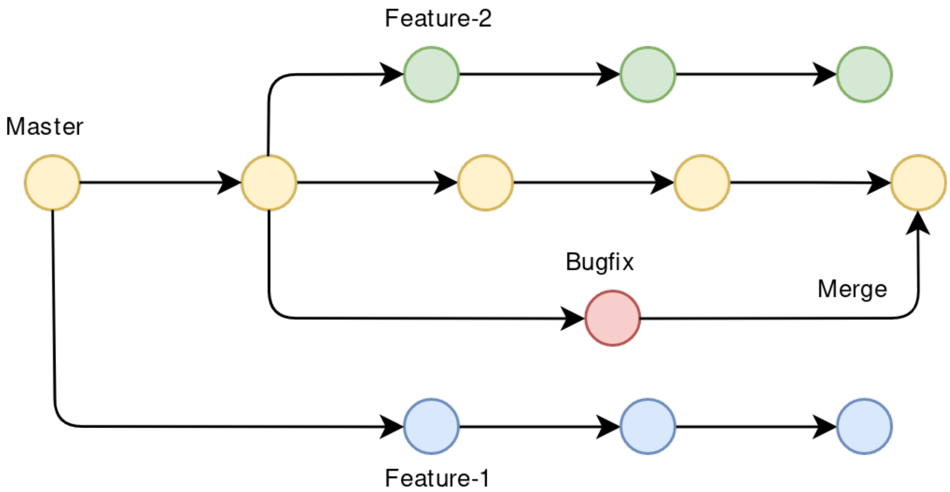


Figure 2.1: A Git commit history tree.

As shown in figure 2.1 two developer can for example create their own branch on which they can work unimpeded. If they finished a task and want to add their work to the master branch, they can now merge their changes. Git then tries to automatically resolve any conflicts which might have emerged from editing the same lines in a file, if that is not possible, it marks the conflicts and allows the user to manually correct them. After this resolution a new *merge commit* is created. This merge commit represents the merge of the changes of two different branches.

With this methodology it is possible to work with many people or teams on the same project, without accidentally overwriting changes of another developer, whilst maintaining a clear history of all changes in the project.

Another important feature of Git is the possibility to set up a *remote*. A remote acts as a single source of truth a developer can *push* their changes to or *pull* changes from other developers. It can for example be a distinct server, which is attached to some kind of network that is accessible by the developers. This feature allows developers to distribute to a project, as long as they have access to this network. Git also supports several protocols such as Hypertext Transfer Protocol (HTTP) or Secure Shell (SSH) to connect to the remote and to provide a simple user management layer.

### 2.1.2 Git User Roles

There exist two roles in Git, namely the *committer* and the *author*. Every commit in Git contains the email addresses and the names of these two people. The author of a commit is the person which actually contributed the changes in the files. The committer is the person, which created the git commit. This is important to keep track of the original author of the changes.

Lets look at the case of an author contributing code to a project in an email with an attached patch file. If a maintainer of the project now applies the patch file and commits without setting the author, the information about the original author would be lost. Collected data indicates that in about 89% the author and the committer are the same person.

### 2.1.3 Internal Representation

Git underlying storage and management solution for files is commonly described as an mini filesystem [1, p. 9], thereby I will refer to this as an file system (FS) from now on. Git provides a collection of high level abstraction tools to work with it's underlying FS. In the following I will explain the most important aspects of Git's FS structure and management.

The representation of a single file in Git is a called *blob* object [1, p. 56]. A blob object is a file, which has been added to a Git FS. It is compressed and saved in the `.git/objects` directory under the respective Secure Hash Algorithm 1 (SHA-1) hash of the uncompressed file. As follows there exists a blob object for every version of every file of the project.

The SHA-1 hashing for unique file identifier might seem unsafe at first, but the probability of a SHA-1 collision is really low, roughly $10^{-45}$. Lately Google managed to force a collision in an controlled environment in 2017, but it is really unlikely to encounter a collision under normal circumstances [4]. This characteristic of SHA-1 hashing will become quite important in the design of the database later on.

As mentioned in the introduction 2.1.1 Git is used to store the state of a specific directory on any underlying Operating System (OS) FS. To represent a FS or to simply bundle multiple Git blob objects together, Git uses the tree object.

A tree object is a file, which has a SHA-1 hash reference to all underlying blob and tree objects as well as their names and file permissions. To represent a subdirectory a tree simply holds a reference to another tree object. With these tools git manages to build it's own basic representation of a file system.

```
100644 blob 11d1ee77f9a23ffcb4afa860dd4b59187a9104e9  .gitignore
040000 tree ac0f5960d9c5f662f18697029eca67fcea09a58c  expose
100644 blob 61b5b2808cc2c8ab21bb9caa7d469e08f875277a  install.sh
040000 tree 8aaf336db307bdcab2f082bd710b31ddb5f9ebd4  thesis
```

Listing 1: A tree file example.

As stated before the commit is utilized to provide an exact representation of a state of the repository's files and directories. Just as blob object, the tree and commit files are also stored in the `.git/objects` directory under their respective hash.

```
tree      cd7d001b696db430b898b75c633686067e6f0b76
parent    c19b969705e5eae0ccca2cde1d8a98be1a1eab4d
author    Arne Beer <arne@twobeer.de> 1513434723 +0100
committer Arne Beer <arne@twobeer.de> 1513434723 +0100

Chapter 2, acronyms
```

Listing 2: A commit file example.

As you can see in listing 2, the commit is just another kind of file utilized by Git, which contains some metadata about a repository version:

- The reference to a tree object, which represents the root directory of the commit's version of the project.
- A reference to one or multiple parent commits, to maintain a version history.
- The name and email address of the author.
- The name and email address of the committer.
- The Coordinated Universal Time (UTC) timestamps with UTC offset for the commit and author date.
- The commit message. A message with arbitrary text from the committer.

The commit is the most important object for this thesis. It contains crucial information such as the date of the commit as well as the email, which is needed to identify a contributor across several commits.

### 2.1.4 More features

Git provides many more features, which are not necessarily important for data analysis, but which might need be taken into account when aggregating the data. In the following some functionalities will be shown, which can lead to problems or irregularities in the gathered data.

**rebase**
: It is possible to *rebase* branches. For instance a rebase can rewrite the commit history and change the branch point of a branch to another commit. This is for example a very powerful but also dangerous tool, as it implicitly changes the timestamps of the commits of the rebased branch.

**force push**
: Git allows to push to a remote with the `–force` flag, which is called a *force push*. This enables the users to rewrite every commit in the whole history tree. If another user has a git repository version

## 2.2 Data source

The biggest initial task for this thesis was the acquisition of data. Selecting a data source was a crucial step, as good data for analysis and evaluation is the backbone of this thesis. This section will list these requirements in detail and evaluate why I chose to use Github as a data source. Furthermore some functionalities of Github will be explained and a brief overview of the data provided by Github's Application Programming Interface (API) will be given.

### 2.2.1 Requirements

The data source had to satisfy as many requirements, which were specified in Chapter 1.1, as possible.

To accomplish a meaningful analysis of commit behaviour one needs a sufficient amount of commits. For instance it is necessary to have a few commits per weekday over a timespan of a month for a simple sleep rhythm analysis. If there are only 20 commits for a user over the past month there is probably not enough data for a representative analysis. To gather as many commits as possible we have to get access to as many repositories to which the targeted user contributed to as possible. Thereby the data source has to provide a way to dynamically explore repositories around a single user.

For analysis of companioned persons as described in Section 1.1.3 it is crucial to find users, which are likely to know each other. Optimally the data source provides a functionality for users to actively mark other user as their friends or colleagues. There should at least be the possibility for some kind of link between different users which.

To attack a company, as described in Section 1.1.3, or to spy on company members, as described in Section 1.1.1, we want in the best case all repositories owned by the company. The data source thereby needs to provide some kind of representation for a company should be available. Ideally there should also be a list of all company members for evaluation purposes of data mining findings.

A recap of the requirements to the data source:

- Realistic noise
- Real world data
- Large amount of repositories
- Access to all commits of each repository
- Access complete metadata for each commit
- Email to user association
- Methods to discover repositories a user contributed to
- Methods to discover possibly companioned contributer

- A representation of a company
- Access to members of a company

### 2.2.2 Github

I decided to use Github as a data source, for it is not only convenient to find Uniform Resource Locators (URLs) for cloning repositories, but also provides solutions for most of the other requirements. It hosts one of the biggest collections of open source projects [6] with 64 million repositories, 24 million users and 1,5 million organizations [3]. Github also provides a well documented API for querying its metadata and there are libraries for most major languages, which provide an abstraction layer for this API. This API is publicly available and can be used by anyone registered on Github.

For instance Gitlab, one of Github's competitors, has much less data to offer. Gitlab doesn't provide detailed usage statistics, but they state that they only host about 100000 organizations, which is remarkably less than Github [2]. As Gitlab is an open-source project, there also are an unknown number of privately hosted Gitlab instances, which are completely unaccessible for unauthorized users.

On the other hand one of the downsides of using Github is, that we don't have access to all metadata. For example the full list of members for organizations is often unaccessible, as users actively need to opt-in to be publicly displayed as a member of the team. The internal team structures of organizations are not visible, as one needs to be a member of the organization to access those. Another problem are dangling email addresses, which are not related to any account any more. All commits made with this email address cannot be used any more for any analysis which requires a user to commit relationship. Even though some ground truth is missing, I decided to use this approach as it still was the most promising way to gather as much data and real world noise as possible, compared to other open source hosting services.

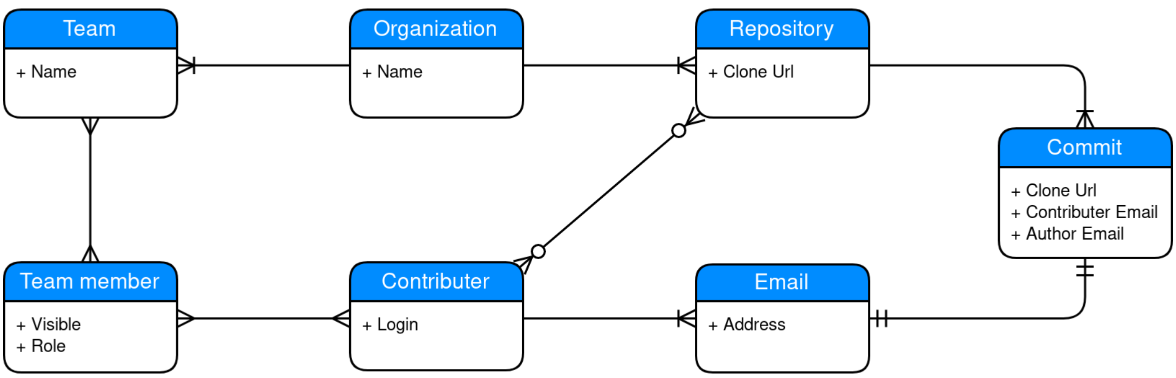


Figure 2.2: Simplified Github relationships.

### 2.2.3 Github's features

In the following I will explain some of the features provided by Github, which cover the requirements listed in Section 2.2.1. Github offers some features, which are convenient to for example find repositories a specific user contributed to or to find contributers which likely personally know each other.

**Stars**

A very crucial feature is *starring*. Every user has the possibility to star a repository to show appreciation or interest in this specific project. Hence popular repositories usually have a comparative big amount of stars. For instance the Github Linux kernel mirror has a star count of over 58000 [1]. Even though Github allows to query all repositories, which are owned or forked by a user, their API doesn't provide a method to get all repositories a user ever contributed to. However Github provides an endpoint to query all starred repositories of an user. In case a user stars a repository he contributed to, whilst not owning it, it is now possible to get this repository with this feature. Of course it is still not a reliable way to get all repositories to which a user ever contributed to, but it is a viable approach to get at least a few of them.

**Follower**

Another important feature is *following*. Every user can follow any other user to get informed, if they do specific things like creating new repositories or starring repositories, or to simply show interest in or respect for their work. By getting all followed or following users, one might catch some friends of the user. It is also possible that a user follows the owner of a repository he contributed to. By using this feature it is thereby possible to catch some additional contributed repositories as well some friends of the user.

**Organizations**

The third feature are *organizations*. An organization is used to host projects under an account which is not necessarily led by a single natural person, but rather supports roles with different permissions and team structures. Github allows to query all repositories of an organization via their API. This enables us to link an organization to its owned repositories and as a result to perform analyses for users on a specific organization repository subset.

Generally organizations provide us with some important ground truth, even though the information might not be complete. Despite not knowing all members of an organization, we still get some useful information to estimate the tendency of precision of our knowledge extraction algorithms.

[1] 'Linux kernel source tree' Github.com, https://github.com/torvalds/linux (accessed, 24.04.2018)

## 2.3 Data Aggregation

As mentioned in Section 2.2.2, I decided to use Github as my primary data source and to utilize their *Github APIv3* for this purpose. The Aggregator and analysis program written for this thesis is named *Gitalizer*. In this section I will explain the technologies and methods used in the data aggregation process, the database structure and the interaction with Github's API. In the end some problems which occurred during the data collection will be shown as well.

### 2.3.1 Existing solutions

There are a number of existing solutions for accessing git meta data.

### 2.3.2 Database

To store the gathered Information I chose a Structured Query Language (SQL) based solution. PostgreSQL provides excellent tools to ensure a high consistency in your database, namely check constraints, as well as great support for working with times, time zones and locations. The SQL database is used in combination with the Object-Relational Mapping (ORM) library *SQLAlchemy*.

#### Database Design

The basic schema created for the purpose of this thesis consists of five ORM models. A model for Commits, Emails and Repositorys, Contributers as well as Organizations has been created. The latter is only important to validate results and is not actually used for knowledge discovery, as this is Github specific data.

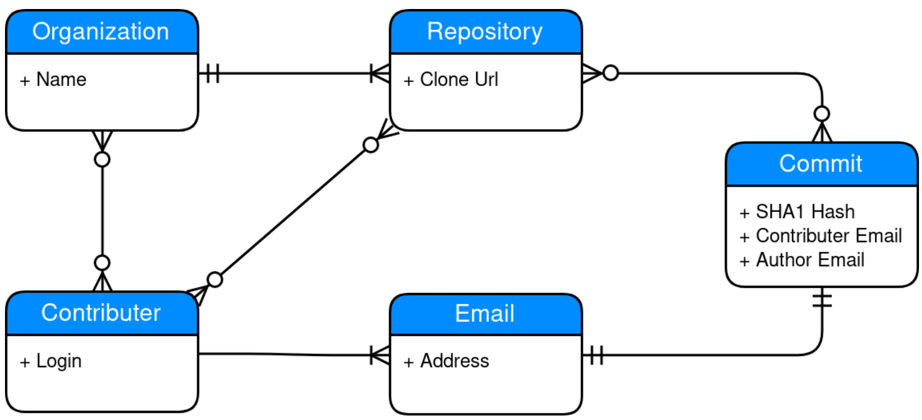


Figure 2.3: Gitalizer database relationships.

Every commit of each repository is saved in the database along with its SHA-1 hash and the two email addresses as in listing 2. It is important to note that there is a many-to-many relationship in figure 2.5 between commits and repositories. This feature prevents duplication of the same commits from forked repositories. It is for instance a common practice to create a fork of a repository to develop without intervening with the main git repository of the project. As described in Section **??** the probability of a SHA-1 collision is highly improbable. By exploiting this feature it is possible to enforce a unique constraint on the commit hash column, assuming that any duplicated commit hash actually results from a forked or copied repository. Without this mechanism it could be possible that the same commit of a contributer could be used multiple times as a result of repository forking. After collecting 43 million commits from Github and actively ignoring obvious project forks, there are still 49 million references between commits and repositories. This means that about 13% of gathered commits result from forked repositories which can not easily be identified as such.

As stated above each commit is also saved with its respective email addresses. There exists a one-to-many relationship between contributers and emails, as every contributer can obtain an unlimited amount of email addresses. To unambiguously assign a commit to a person, it is necessary to connect all email addresses of these commits to this person. This relationship does not have a `NOT NULL` constraint as it happens quite often that an email address can not be assigned to any person. Looking at the collected data it appears that roughly 20% of collected email addresses from Github are no longer connected to an active user.

As stated in Section **??** Github provides a way to organize several people in organizations and teams. As one of the goals of this thesis is to see if it is possible to detect member of an organization in open-source projects, a model for organization has been created as well. This data can then be used to check against the results of this research's results.

### 2.3.3 Gitalizer

The Program written for this thesis features data aggregation, preprocessing, knowledge extraction and visualization. Gitalizer uses a PostgreSQL database for data storage and data consistency checks as described in 2.3.2. For interaction with the Github API the *PyGithub* library is used, which provides a convenient abstraction layer for requests and automatically maps JavaScript Object Notification (JSON) responses to *Python* objects.

The data aggregation module of Gitalizer is capable of several approaches for gathering data. In the following we will look at those approaches in detail.

**Git repository**

Gitalizer can scan any git repository from a SSH or HTTP URL as long as the current user has access to it. The repository is cloned into a local directory and when the cloning finished the actual scanning process begins. During the scan, we git checkout the *HEAD* of the current default branch for this repository and walk down every reachable commit of the Git history. The program saves all available meta data for each commit in its database, namely the emails, timestamps and names of the committer as well as additions and deletions to the project in lines of code.

After this scan we are still missing a lot of information. The unique identifier of an author or committer is their email address, as names may change or can be ambiguous. The problem with the simplicity of Git is that there exists no concept of an user. Thereby we cannot easily link email addresses to a specific contributer without additional metadata.

**Github Repository**

To tackle the problem of missing meta data in 2.3.3, I used the Github API to get some of the missing meta data. The general approach is the same as in the previous scan method. The repository is cloned and locally scanned. However, a request to Github is issued every time a email is found, which we do not already have linked to a contributer. Github allows to link multiple email addresses with a single user account and automatically references the respective user in their own API commit representation. With this additional meta data we gain ground truth about the identity of an author or committer.

Anyway this approach does not work, if the user of a commit removes the email, which has been used for the commit, from his account or if the user deletes their account. If this happens and the email contributor relationship has not already been created, there is nothing that can be done and these commits need to be handled later on in the preprocessing of the data.

**Github User**

To try getting all repositories of a specific user a new functionality, which highly utilizes the Github API, has been added. At first several requests are issued to get all repositories of the specified user, as well as all starred repositories of this user. During the repository exploration, every relevant repository is added to a shared queue, lets call it "repo-queue", which is then processed by a multiprocessing pool of workers. Each worker process then scans a single repository as described in 2.3.3 and gathers missing meta data as described in 2.3.3.

**Connected users**

For detection and analysis of connections between contributers over multiple repositories additional user repository discovery as described in 2.2.1, another feature has been added to Gitalizer. Gitalizer is able to achieve this by not just scanning a single user, but rather scanning the repositories of the specific user, as well as the repositories of all following and followed users as described in **??**. For this task two different worker pools are utilized. The user discovery pool is initialized with a shared queue, lets call it "user-queue", of all users we need to look at. This pool simply searches for relevant repositories of each user and passes the repository

URL to a second shared queue. The second pool then processes the “repo-queue” as described in 2.3.3.

For organizations it’s nearly the same approach. Initially all repositories, which are owned by the organization, are added to the “repo-queue”. All publicly visible organization members are then added to the “user-queue” and processed as described above.

### 2.3.4 Database optimization

As the database kept growing, it became the bottleneck in the aggregation process several times. As a result adjustments in the database schema, PostgreSQL parameter tweaking and migration to better hardware were necessary. The first considerable slowdown occurred after reaching about 12 gigabytes (GBs) of data. At this stage the database write and read operations took longer than the actual aggregation process and the whole machine started to become unresponsive because of high I/O load.

The performance of the database then needed to be continuously tweaked in several steps. The first counter measures was the reduction of commits using deduplication through utilizing the SHA-1 hash as stated in Section 2.3.2 The ignoring of forked repositories reduced the amount of entries in the relation table between commits and repositories by another 26%.

The next performance boosts were achieved by disabling or loosening of several fail-safe mechanisms of PostgreSQL, namely ‘synchronous commit’ and ‘write ahead’ parameter, which are designed to save data on a system crash. As there is no important or critical data handled it was acceptable to pass on these mechanisms, and trade safety for performance.

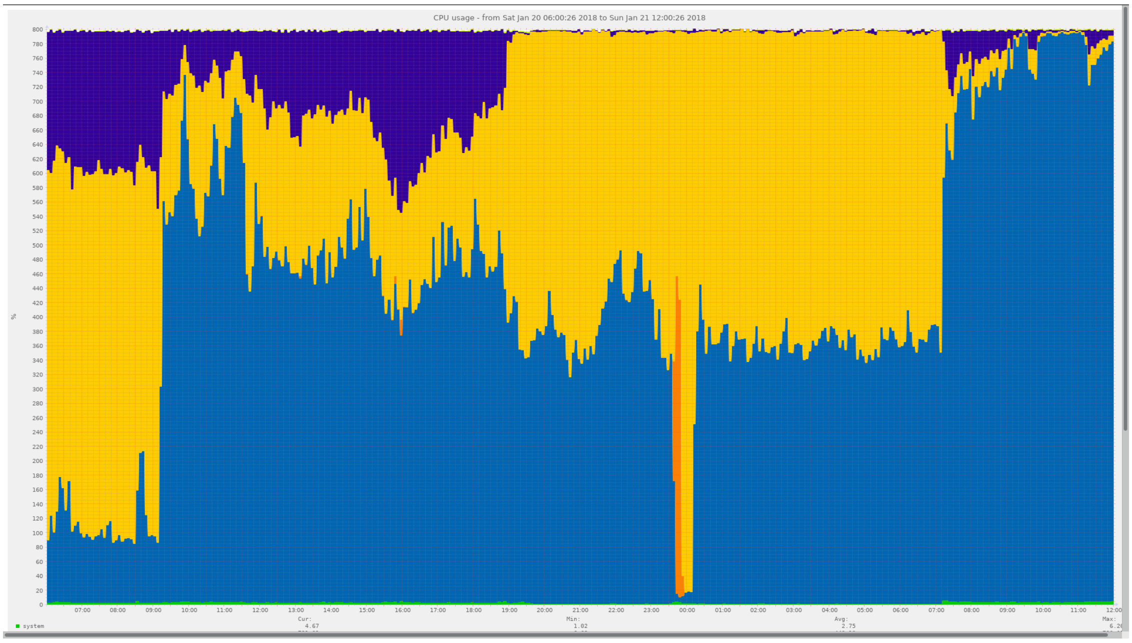


Figure 2.4: The CPU load of the aggregation server during optimization.

After renting a root server and deploying gitalizer onto it the aggregation process worked really well until the database size hit about 25 GB. In Figure you see the overall central processing unit (CPU) load right before optimizing several SQL queries by caching intermediate results and increasing the cache size of PostgreSQL. The dark blue represents the I/O wait time while the light blue represents the actually used processor capacity by the aggregator. Due to complex and numerous SQL queries the server became partly unresponsive and the aggregation process began to stall.

After many improvements the server can now run with 16 worker threads and roughly 38 GB of data without any signs of slowdown whilst using the rate limit to capacity.

### 2.3.5 Incremental aggregation

To ensure a constantly up to date database Gitalizer needed to be capable of fast rescans of repositories. The initial scan of a repository always includes cloning, scanning the whole repository and writing it into the database. After a repository is scanned completely at least once it is marked as as such and won't ever by completely scanned again. All following runs then only clone the repository and scan the newest unknown commits. These are discovered by performing a breadth first search until no new nodes are found.

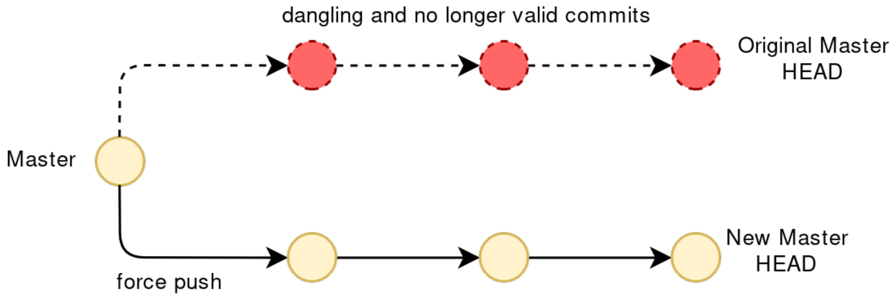


Figure 2.5: Gitalizer database relationships.

As explained in Section 2.1.4 it is possible to rewrite commits and force push them. This scenario needs to be explicitly handled since force pushes can completely alter the history of a git repository, which can subsequently lead to a split in the Git history and leaves dangling commits. As the complete history of a repository is stored inside the database, Gitalizer needs to detect a force push by walking down the git history tree until it finds known commits. If any of these commits has children, which are not in the newly scanned commits, a force push took place and the old commit history has to be truncated, since it is now outdated and irrelevant.

### 2.3.6 Problems

During the development of the data aggregator I experienced a few problems and edge cases which needed to be handled. The earliest and most delaying problem was the rate limit of the Github API, which limits to 5000 requests per hour. But beside this rate limiting there also is an abuse detection mechanism, which triggers, if too many requests are fired in a short amount of time, which resulted in various hacks with random wait times to detain those mechanisms from triggering.

The first version of the aggregator didn't clone and scan the repository locally, but rather gathered all information from the Github API endpoints. This approach worked well until the aggregator hit the official repository of Nmap, which has about 11.000 commits and took over three hours to scan. Soon I realized that this would severely slow down my research and I then started to continuously minimize the amount API calls issued by Gitalizer. A connected user scan of my own Github account led to about 600.000 commits and took about one and a half day on the final working version of Gitalizer, to provide you with a reference of scale.

After implementing multiprocessing, I managed to hit the rate limit again, as I was now issuing requests with many threads. To fix this issue I implemented a wait and retry wrapper around every single function call or object access, which triggered a call to the Github API. Afterwards the aggregator was capable of running multiple days without worker processes silently dying or incomplete collected data.

Fine tuning the edge cases and the handling of the API took about 3 months, since there were many problems such as unpredictable error responses from github, missing data in queries or simply unknown or broken encodings in Github's meta data.

Another problem occurred during the local scanning of the repositories. The library used for interaction with Git *libgit2* issued a *stat* Linux syscalls during a diff operation for each file which changed between those commits in case there were any local not-commited changes. Anyhow the repositories, which were locally scanned, were cloned with *bare* mode. This means that there exists no project root directory, but rather only the git internal representations of those files, which makes the behaviour stated above unnecessary and unwanted. As a result all processes slowed severely down due to high I/O wait times, because of stat syscalls on non existent files. Luckily after reporting the issue [2] it was resolved in a week and I was able to continue developing with my own compiled version of the libgit2 library.

[2]'Unnecessary syscalls on bare repository' github.com, https://github.com/libgit2/libgit2/issues/4480 (accessed, 25.04.2018)

# Chapter 3
# Analysis

In this chapter several attacks as listed in Section 1.1 will be shown. The methods and

## 3.1 Holiday and Sick Leave Detection

The information about anomalies in the regular work pattern can be a valuable information for several parties. Usually only few parties people know about the holiday or sick leave times of a person. To know if a persons tends to become sick often or for long times is a dangerous intrusion into a persons privacy. For instance this could be abused by head hunters or personnel managers to cull possible employees with too high sick leave rates and thereby reduce the job prospects of the target.

For employees this might convenient to detect anomalies in the productivity of an employee. In case an employee doesn't commit on a regular basis for several days, this behaviour would be instantly visible with this method.

Another attack vector could be to look at the correlation of miss-out between several employees. This attack could even be performed by an outsider, if the employees of a company are known. The information gained by this attack could be quite delicate, as they could reveal relationships between employees. This attack is heavily inspired by an article about data mining articles from the popular German weekly magazine *Der Spiegel* written by the David Kriesel [5].

### 3.1.1 Implementation

The requirements for this algorithm is the detection of a regular work pattern for a given interval. It must have the ability to adjust to a changing work pattern, but at the same time has to be capable of detecting anomalies in this pattern. If the attacker wants to look at multiple people, some kind of measure for similarity in the missing time patterns has to exist.

The input for this analysis is the intersection between all commits from the considered repositories and all commits from the considered contributors. The commits' meta data used for this analysis are time stamps as well as additions and deletions in lines of code.

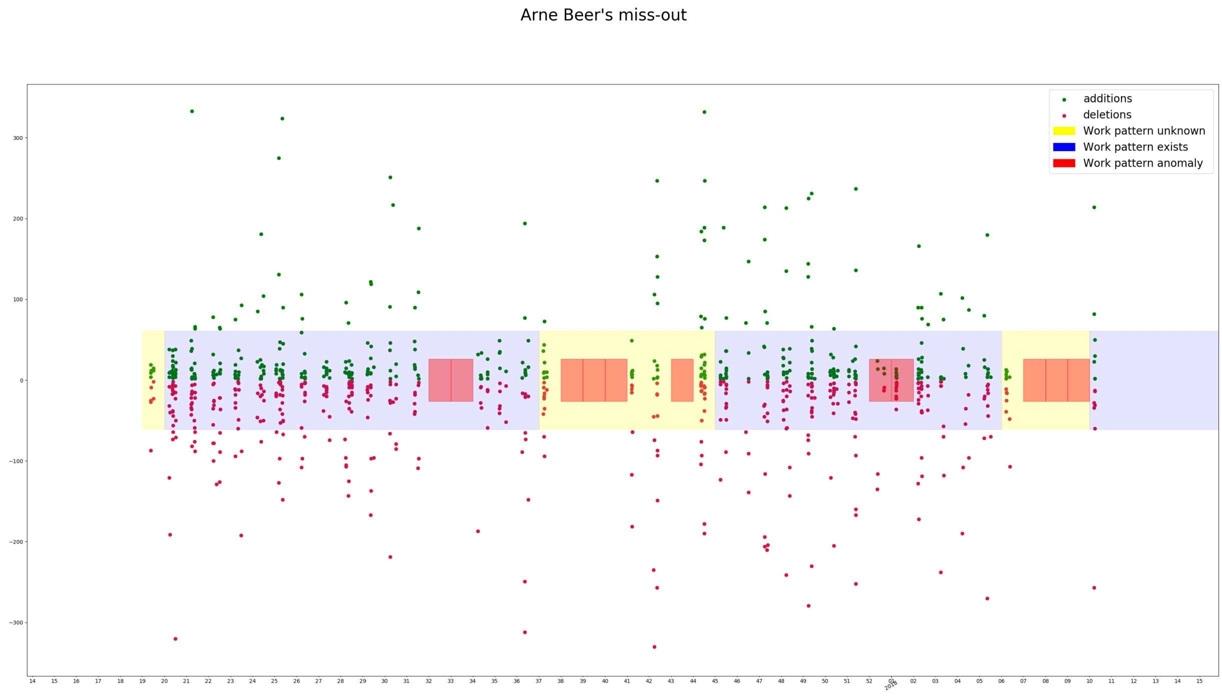


Figure 3.1: The work time analysis of the author.

The analysis of the data is a chronological scan of all commits for specific user. Before performing the actual analysis, the data is converted into a usable format representing the week days. The converted data format represents the amount of commits for each day in the last year. It is really difficult to measure productivity in lines of code committed or in the amount of commits made by a person, as they don not necessarily display the amount of work that have been put into those commits. As a result I decided, that a day counts as a work day as long as at least single commit has been made during the day.

```
def analyse(weeks):
    prototype = None
    for index, week in weeks.items():
        next_six_weeks = weeks[index:index+future_lookup]
        if not prototype:
            # See if there is a prototype in the next few weeks.
            prototype = find_prototype(next_six_weeks)

            # Check if this specific week is a anomaly
            check_anomaly(prototype, week)

            continue
```

```
        prototype_exists = prototype_exists_in_next_weeks(next_six_weeks)
        if not prototype_exists:
            # We couldn't find the prototype in the next few rows
            # Try to find a new prototype
            prototype = find_prototype(next_six_weeks)

        check_anomaly(prototype, week)


def check_anomaly(prototype, week):
    if week.working_days == 0:
        save_anomaly(week)

    if prototype is not None:
        different_days = week.working_days - prototype.working_days
        // A single day variance is acceptable
        if different_days >= 1:
            save_anomaly(week)
```

Listing 3: Miss-out analysis .

The algorithm inspects every week work pattern of a given interval. At the beginning a new *prototype* is tried to be found. A prototype is a representative week work pattern, which resembles the average work day pattern of the next weeks. This happens in the function `find_prototype`. It performs a simple iteration over a given interval to find a work day pattern, which occurs more often than a given threshold. If a prototype is found, we are capable of identifying anomalies that deviate from this pattern.

For each following week it is firstly checked if this week is a anomaly for this prototype. Anomalies are simply detected by comparing the amount of working days of the prototype and the currently looked at week. The real difference in the working pattern is not suitable for this analysis, as it produces too many false positives for employees with flexible work time.

Secondly it is checked if there exists a week in the near future, which is identical to the prototype. If there is no week identical to the prototype in the near future, the current prototype is reset and a new prototype needs to be found.

In case no prototype can be found, anomalies cannot be easily identified, as there exists no pattern to check against. Only obvious anomalies, namely weeks without a single work day, will then be marked as such.

### 3.1.2 Interpretation and evaluation

Figure 3.3 shows the analysis of the author for his work repositories. The y-axis shows the additions or deletions per commit, the x-axis shows the week of a year. For better analysis and evaluation of the results, a scatter plot with the additions and deletions per commit has been added on top of the miss-out graph.

The evaluation of this algorithm turned out to be quite difficult, as there is no publicly available information about sick leave or holiday. For the purpose of this thesis I had to use anonymous statistics of several friends and colleagues to evaluate the algorithm. The algorithm successfully manages to find all anomalies, which occurred in the last year, for all seven regarded users.

An unexpected side effect of detecting prototypes is that the algorithm also also finds inconsistencies in the work routine. For instance between week 37 to 45 in Figure 3.3 I was forced to reduce my working hours due to legal questions and shift hours and working days. It is hard to interpret those inconsistencies without more contextual information, but nevertheless it provides the fact that something happened during this time.

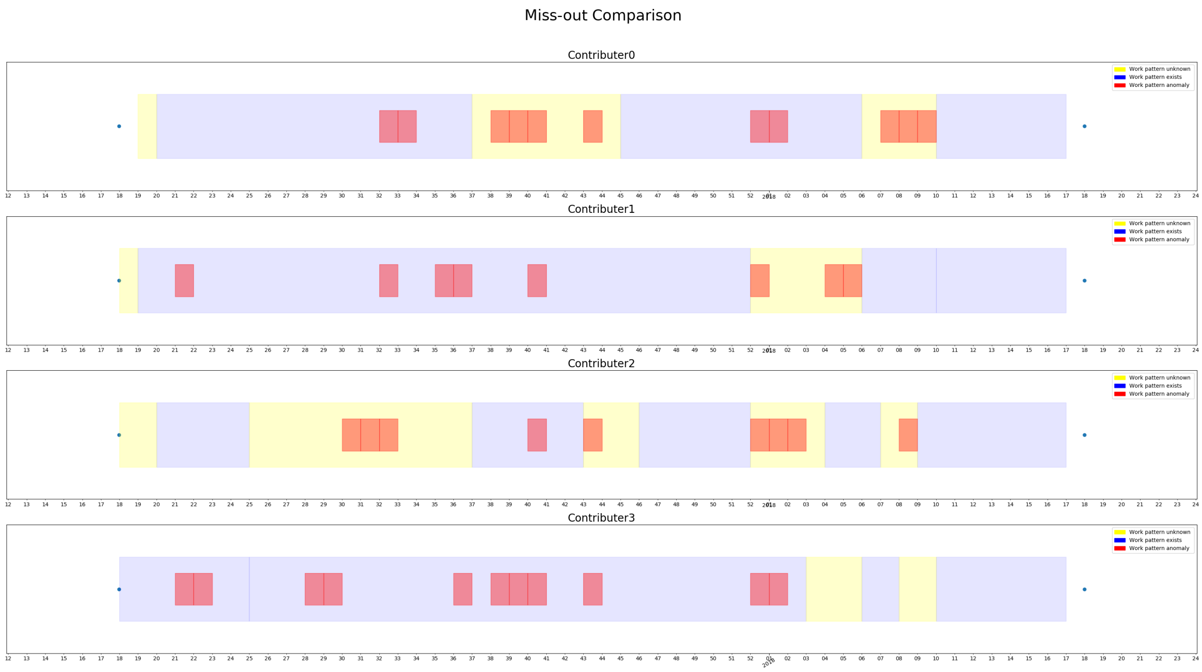


Figure 3.2: The miss-out analysis of several employees.

In Figure 3.4 the comparison between multiple employees can be seen. Contributor0 and Contributor2 are working on flexible work time, while the other two contributors have regular working hours, which reflects in the inconsistencies of those contributors.

## 3.2 Working hours

This attack aims to gather as much information about the working hour behaviour as possible. Knowing the

Usually only few parties people know about the holiday or sick leave times of a person. To know if a persons tends to become sick often or for long times is a dangerous intrusion into a persons privacy. For instance this could be abused by head hunters or personnel managers to cull possible employees with too high sick leave rates and thereby reduce the job prospects of the target.

For employees this might convenient to detect anomalies in the productivity of an employee. In case an employee doesn't commit on a regular basis for several days, this behaviour would be instantly visible with this method.

Another attack vector could be to look at the correlation of miss-out between several employees. This attack could even be performed by an outsider, if the employees of a company are known. The information gained by this attack could be quite delicate, as they could reveal relationships between employees. This attack is heavily inspired by an article about data mining articles from the popular German weekly magazine *Der Spiegel* written by the David Kriesel [5].

### 3.2.1 Implementation

The requirements for this algorithm is the detection of a regular work pattern for a given interval. It must have the ability to adjust to a changing work pattern, but at the same time has to be capable of detecting anomalies in this pattern. If the attacker wants to look at multiple people, some kind of measure for similarity in the missing time patterns has to exist.

The input for this analysis is the intersection between all commits from the considered repositories and all commits from the considered contributors. The commits' meta data used for this analysis are time stamps as well as additions and deletions in lines of code.

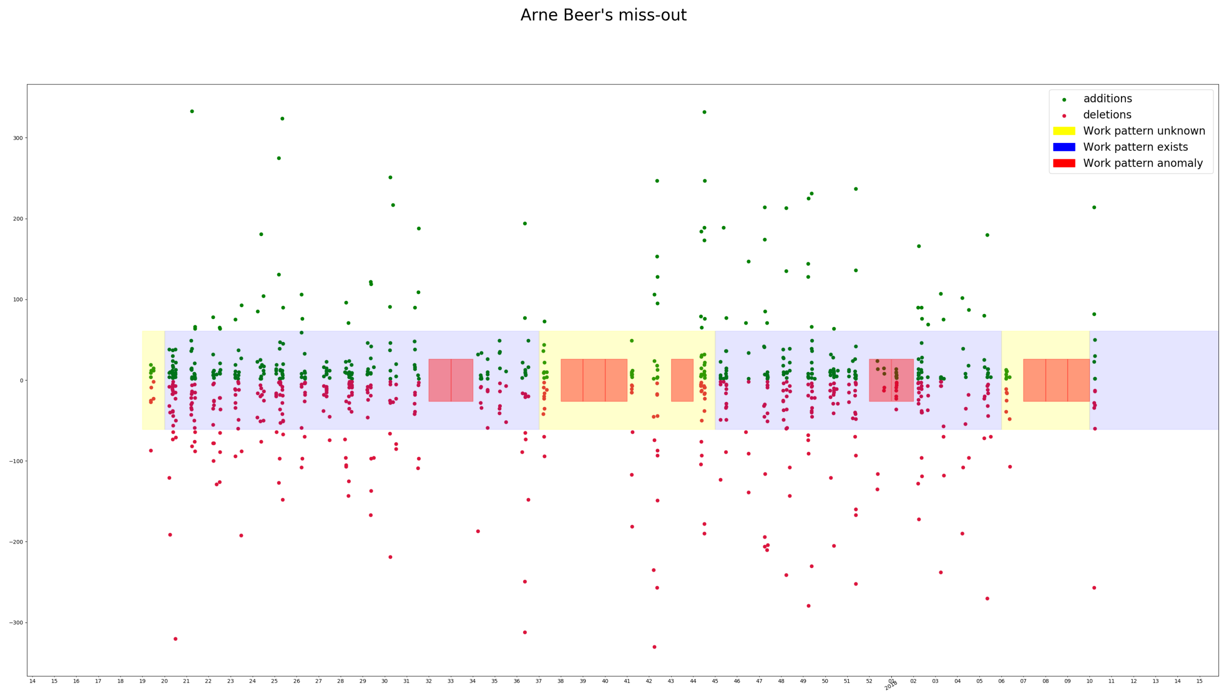


Figure 3.3: The work time analysis of the author.

The analysis of the data is a chronological scan of all commits for specific user. Before performing the actual analysis, the data is converted into a usable format representing the week days. The converted data format represents the amount of commits for each day in the last year. It is really difficult to measure productivity in lines of code committed or in the amount of commits made by a person, as they don not necessarily display the amount of work that have been put into those commits. As a result I decided, that a day counts as a work day as long as at least single commit has been made during the day.

```
def analyse(weeks):
    prototype = None
    for index, week in weeks.items():
        next_six_weeks = weeks[index:index+future_lookup]
        if not prototype:
            # See if there is a prototype in the next few weeks.
            prototype = find_prototype(next_six_weeks)

            # Check if this specific week is a anomaly
            check_anomaly(prototype, week)

            continue
```

```
        prototype_exists = prototype_exists_in_next_weeks(next_six_weeks)
        if not prototype_exists:
            # We couldn't find the prototype in the next few rows
            # Try to find a new prototype
            prototype = find_prototype(next_six_weeks)

        check_anomaly(prototype, week)


def check_anomaly(prototype, week):
    if week.working_days == 0:
        save_anomaly(week)

    if prototype is not None:
        different_days = week.working_days - prototype.working_days
        // A single day variance is acceptable
        if different_days >= 1:
            save_anomaly(week)
```

Listing 4: Miss-out analysis .

The algorithm inspects every week work pattern of a given interval. At the beginning a new *prototype* is tried to be found. A prototype is a representative week work pattern, which resembles the average work day pattern of the next weeks. This happens in the function `find_prototype`. It performs a simple iteration over a given interval to find a work day pattern, which occurs more often than a given threshold. If a prototype is found, we are capable of identifying anomalies that deviate from this pattern.

For each following week it is firstly checked if this week is a anomaly for this prototype. Anomalies are simply detected by comparing the amount of working days of the prototype and the currently looked at week. The real difference in the working pattern is not suitable for this analysis, as it produces too many false positives for employees with flexible work time.

Secondly it is checked if there exists a week in the near future, which is identical to the prototype. If there is no week identical to the prototype in the near future, the current prototype is reset and a new prototype needs to be found.

In case no prototype can be found, anomalies cannot be easily identified, as there exists no pattern to check against. Only obvious anomalies, namely weeks without a single work day, will then be marked as such.

### 3.2.2 Interpretation and evaluation

Figure 3.3 shows the analysis of the author for his work repositories. The y-axis shows the additions or deletions per commit, the x-axis shows the week of a year. For better analysis and evaluation of the results, a scatter plot with the additions and deletions per commit has been added on top of the miss-out graph.

The evaluation of this algorithm turned out to be quite difficult, as there is no publicly available information about sick leave or holiday. For the purpose of this thesis I had to use anonymous statistics of several friends and colleagues to evaluate the algorithm. The algorithm successfully manages to find all anomalies, which occurred in the last year, for all seven regarded users.

An unexpected side effect of detecting prototypes is that the algorithm also also finds inconsistencies in the work routine. For instance between week 37 to 45 in Figure 3.3 I was forced to reduce my working hours due to legal questions and shift hours and working days. It is hard to interpret those inconsistencies without more contextual information, but nevertheless it provides the fact that something happened during this time.

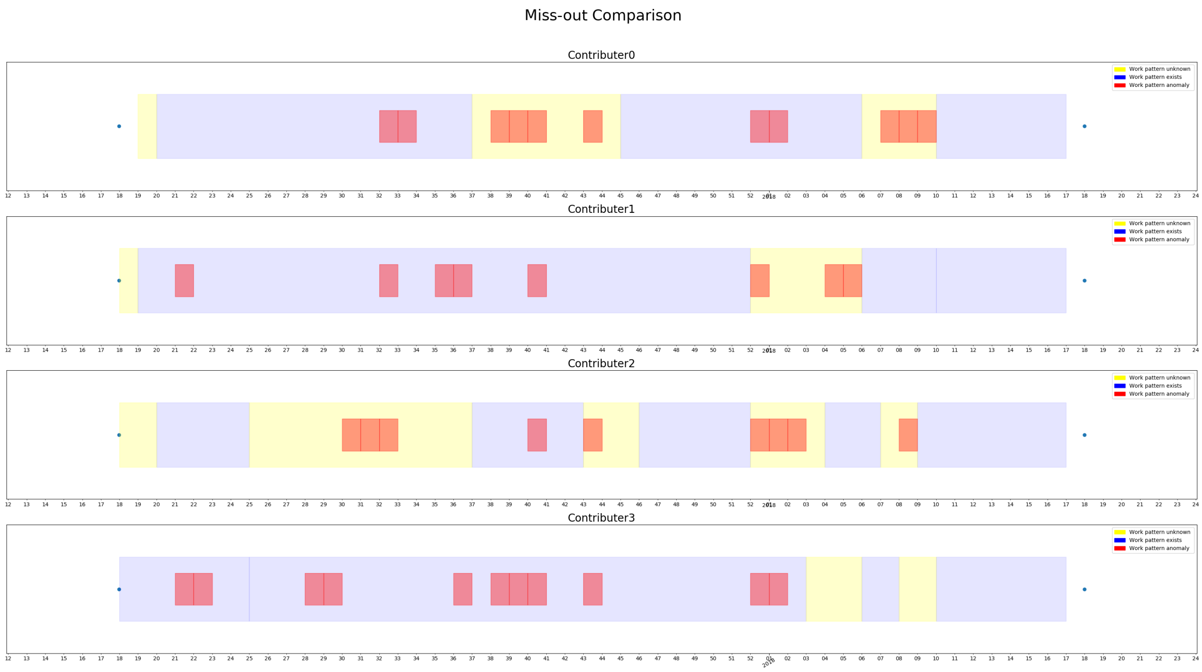


Figure 3.4: The miss-out analysis of several employees.

In Figure 3.4 the comparison between multiple employees can be seen. Contributor0 and Contributor2 are working on flexible work time, while the other two contributors have regular working hours, which reflects in the inconsistencies of those contributors.

# List of Figures



# List of Listings



# List of Tables

# Bibliography


[1] Scott Chacon and Ben Straub. *Pro Git, 2nd Edition.* Apress, 2014. ISBN: 978-1484200773.

[2] GitLab. *GitLab Help.* 2017. URL: `https://gitlab.com/help` (visited on Jan. 22, 2018).

[3] Google. *Version Control Systems Popularity in 2016.* 2017. URL: `https://octoverse.github.com` (visited on Jan. 22, 2018).

[4] Google. *Version Control Systems Popularity in 2016.* 2017. URL: `https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html` (visited on Dec. 16, 2016).

[5] David Kriesel. *SpiegelMining: Wer, wann, was, mit wem? Das soziale Netz der SpiegelOnline-Redakteure.* 2016. URL: `http://www.dkriesel.com/blog/2016/0814_spiegelmining_soziales_netz_redakteure` (visited on Apr. 28, 2018).

[6] Cade Metz. *Version Control Systems Popularity in 2016.* 2015. URL: `https://www.wired.com/2015/03/github-conquered-google-microsoft-everyone-else/` (visited on Apr. 20, 2018).

[7] rhodecggode.com. *Version Control Systems Popularity in 2016.* 2016. URL: `https://rhodecode.com/insights/version-control-systems-2016` (visited on Jan. 22, 2018).

# Eidesstattliche Erklärung

„Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.“

| | |
|---|---|
| Ort, Datum | Unterschrift |