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 6, 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
DBSCAN	Density-based spatial clustering of applications with noise
DST	daylight saving time
DSTs	daylight saving times
FS	file system
GB	Gigabyte
HTTP	hypertext transfer protocol
IANA	Internet Assigned Numbers Authority
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 [**article:git-popularity**]. 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 [**article:github-statistics**].

Some well known projects and organizations use Git, for example Linux, Microsoft, Ansible and Facebook [**article:github-statistics**]. 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 contributor 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. Furthermore I want to create a foundation for future research and provide a first example on how to perform such attacks.

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 contributor, 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

This chapter will attend to the collection of required data as stated in Section **??**. At first the **vcs!** (**vcs!**) *Git* will be introduced and its functionalities explained. The actual source of the data *Github* will then be evaluated in terms of amount of ground truth and availability. At last the methodology used for aggregation and exploration of Github will be explained.

2.1 Git

This section introduces the **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 at least two commits, Git is capable of showing the exact changes between any two commits, which is called a *diff* and it allows to jump between different commits 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 [**article:git-popularity**]. It enables to work with multiple developers on a single code base, as it provides several techniques to prevent, detect and resolve conflicts of changes at the same code, 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. 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*.

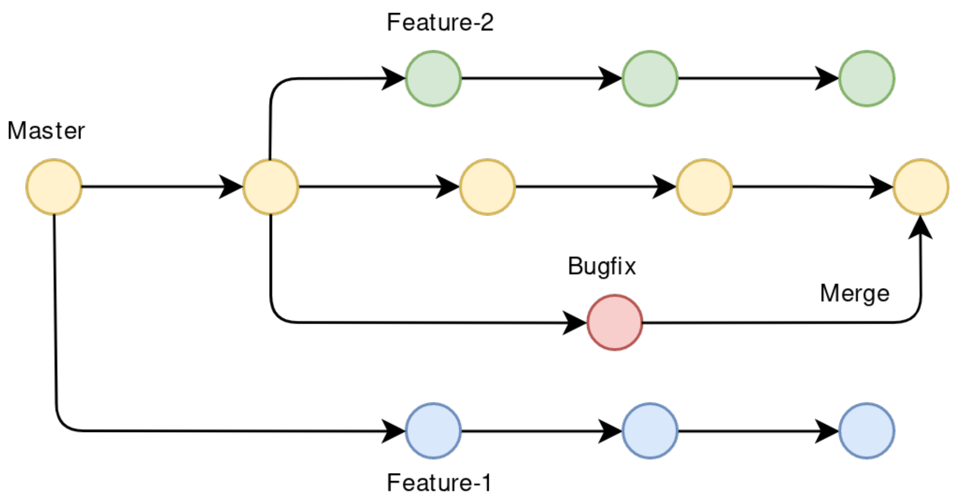

Figure 2.1: A Git commit history tree.

As shown in figure **??** two developer can for example create their own branch on which they can work unimpeded. If they finished their tasks 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. A merge commit represents the merge of changes from 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 centralized repository developers 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 contribute to a project, as long as they have access to the remote's network. Git also supports several protocols such as **http!** (**http!**) or **ssh!** (**ssh!**) to connect to a remote and thereby 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 those 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's underlying storage and management solution for files is commonly described as an mini **fs!** (**fs!**) [**book:pro-git**] In the following I will thereby refer to its file management layer as *git-fs* and explain the most important aspects of it.

The representation of a single file in Git is called a *blob* object [**book:pro-git**]. 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 **sha1!** (**sha1!**) hash of the uncompressed file. As a result, there exists a blob object for every version of every file of the project.

The **sha1!** hashing for unique file identifier might seem unsafe at first, but the probability of a **sha1!** collision is really low, roughly 10^{-45}. In 2017 Google managed to force a collision in an controlled environment, but it is really unlikely to encounter such a collision under normal circumstances [**techreport:sha-collision**]. This characteristic of **sha1!** hashing will become quite important in the design of the database later on.

As mentioned in the introduction **??** Git is used to store the state of a specific directory of an actual **fs!** running under an **os!** (**os!**). 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 **sha1!** hash reference to all underlying blob and tree objects as well as their names and **fs!** permissions. To represent a subdirectory, a tree simply holds a reference to another tree object. With these tools git manages to build its own basic representation of a **fs!**.

```
    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 **??**, 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 **utc!** (**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 Features

Git provides a collection of high level abstraction tools to work with its underlying **fs!**. In the following the most important features for data aggregation will be listed and briefly explained.

HEAD
: To mark the current checkout of an repository, Git utilizes a special marker called *HEAD*. For instance, it is possible, due to this feature, to jump to the previous commit in history with `git checkout HEAD 1`.

diff
: The diff is a tool to compare the state of two different commits in a repository. It allows to list all files which changed between those commits as well as showing the exact changes in the files

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

2.1.5 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.

filter-branch
: In case somebody pushes a huge file, which significantly increases the size of the repository, or adds a file with secret information, for example a password file, git provides the *filter-branch* command. The filter-branch command removes a specified file from every commit in the history and thereby rewrites the history to the point, where this file was introduced to the repository. This command leads to similar problems as the rebase command, since it can severely change the history of a repository.

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 the old git repository version before the force push, they would now be incapable of simply pulling the newest version. Instead they would need to manually checkout the newest commit of the rewritten remote branch.

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 all 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 **api!** (**api!**) will be given.

2.2.1 Requirements

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

To accomplish a meaningful analysis of commit conduct 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 **??** 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 users as their friends or colleagues.

To attack a company, as described in Section **??**, or to spy on company members, as described in Section **??**, the best case scenario would be to have full access to all repositories owned by the company. The data source thereby needs to provide some kind of representation for a company. Ideally there should also be a list of all company members for evaluation purposes of data mining findings.

A summary 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 contributor
- 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 **url!**s (**url!**s) for cloning repositories, but also provides solutions for most of the other requirements. It hosts one of the biggest collections of open source projects [**techreport:how-github-conquered**] with 64 million repositories, 24 million users and 1,5 million organizations [**article:github-statistics**]. 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 [**article:gitlab-help**]. 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.

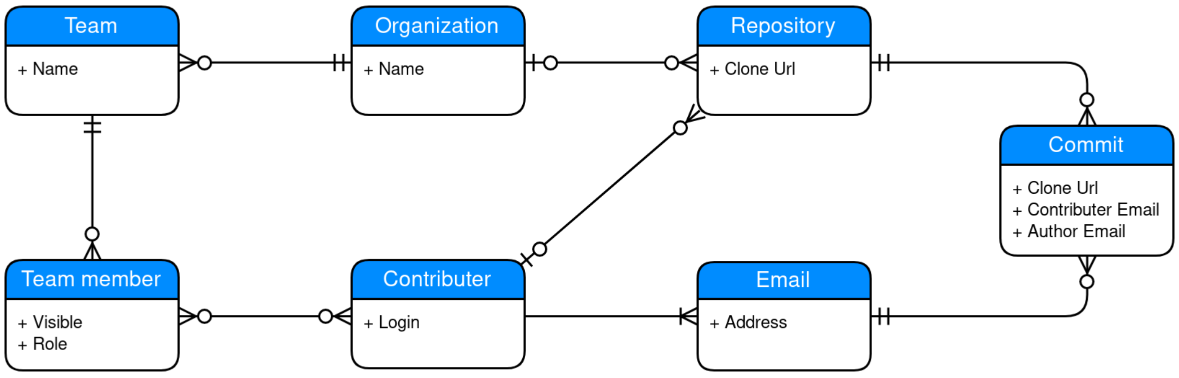

Figure 2.2: Simplified Github relationships in Crow's foot notation.

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 **??**. Github offers some features, which are convenient to for example find repositories a specific user contributed to or to find contributors 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

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

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 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 get 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.

2.3 Data Aggregation

As mentioned in Section **??**, 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 metadata.

Add some references to existing solutions.

2.3.2 Database

To store the gathered Information I chose a **sql!** (**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 **orm!** (**orm!**) library *SQLAlchemy*.

The basic schema created for the purpose of this thesis consists of five **orm!** models. A model for commits, emails, repositories, contributors and 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.

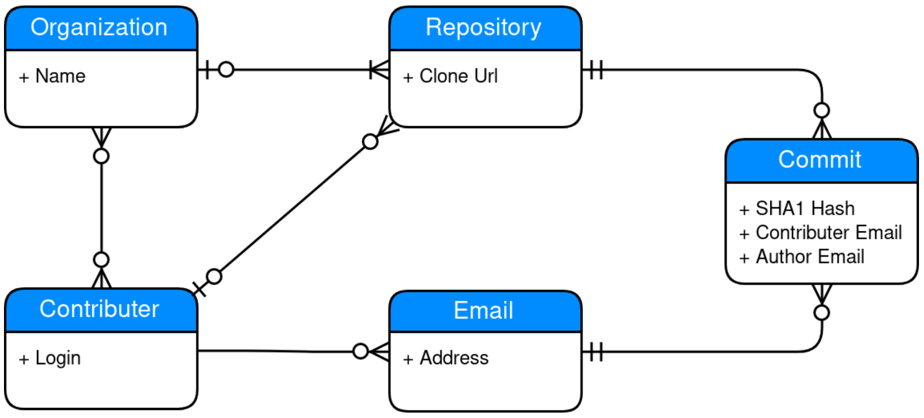

Figure 2.3: Gitalizer database relationships.

Every commit of each repository is saved in the database along with its **sha1!** hash and the two email addresses as in Listing **??**. It is important to note that there is a many-to-many relationship in figure **??** 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 **sha1!** 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. The formula for calculating the probability of such a collision is:

$$p \leq \frac{n(n-1)}{2} * \frac{1}{2^b} \tag{2.1}$$

Without this mechanism it could be possible that the same commit of a contributor 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. Considering Formula **??**, the probability of a collision for 49 million hashes on the 160 bit **sha1!** hash would be about $8.21 * 10^{-34}$.

As stated above each commit is also saved with its respective email addresses. There exists a one-to-many relationship between contributors and emails, as every contributor can obtain an unlimited amount of email addresses. It is necessary to connect all email addresses commit to a specific contributor, to unambiguously assign all commit to their respective contributor. 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 all 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 **??**. For interaction with the Github **api!** the *PyGithub* library is used, which provides a convenient abstraction layer for requests and automatically maps **json!** (**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. After 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 metadata for each commit in its database, namely the emails, timestamps as well as additions and deletions to the project in lines of code.

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

Github Repository

To tackle the problem of missing metadata in **??**, I used the Github **api!** to get some of the missing metadata. 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 an email is found, which we do not already have linked to a contributor. 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 metadata 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 completely 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 has been added, which highly utilizes the Github **api!**. 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 pops another entry from the “repo-queue” and scans a single repository as described in **??**.

Connected users and organizations

For detection and analysis of connections between contributors over multiple repositories additional user repository discovery as described in **??**, another feature has been added to Gitalizer. Gitalizer is able to achieve this by not just scanning a single user, but also scanning 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 worker pool simply searches for relevant repositories of each user and passes the repository **url!** to a second shared queue. The second pool then processes this “repo-queue” as described in **??**.

For organizations it is 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 **gb!**s (**gb!**s) 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 countermeasure was the reduction of commits using deduplication by using the features of the **sha1!** hash as stated in Section **??** 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 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.

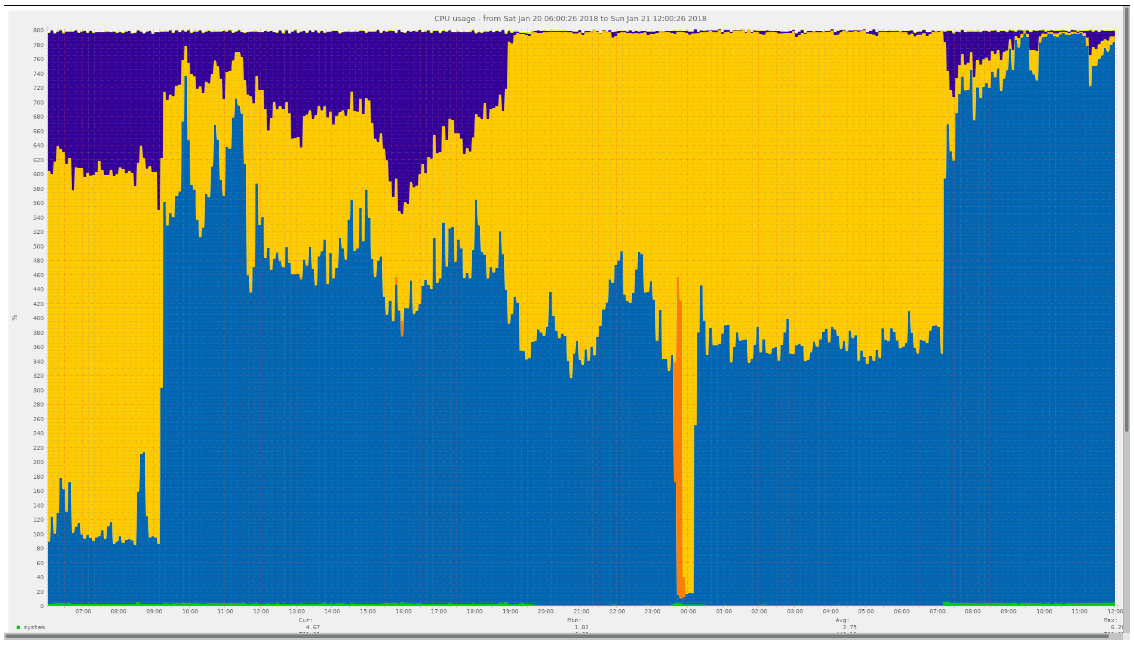

Figure 2.4: The CPU load of the aggregation server during optimization.

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 **cpu!** (**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 will never 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.

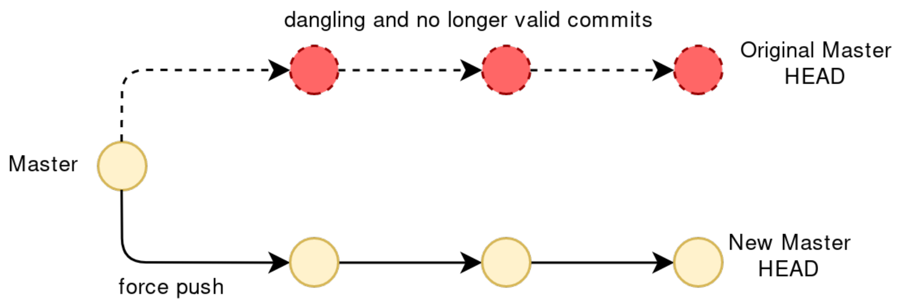

Figure 2.5: Gitalizer database relationships.

As explained in Section **??** 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. An example scenario can be seen in Figure **??**, where all red commits represent the old commit history, which needs to be truncated.

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. 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. The solution for this problem resulted in various hacks, which include 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 to the **api!** with multiple 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 finishing with incomplete 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 metadata.

A big throwback became apparent as I discovered that PostgreSQL automatically normalizes **utc!** timestamps with any offset to the UTC +0 timezone. As a result of this normalization, the exact time of the commit admittedly stays the same, but the crucial metadata about the offset is lost. As a consequence the commit model needed to be adjusted, as the **utc!** offset had to be stored explicitly, and the whole commit aggregation process was started from scratch.

Another problem occurred during the local scanning of the repositories. The library used for interaction with Git *libgit2* issued *stat* Linux syscalls during a diff operation for each file, which changed between those commits, to check if there were any local uncommitted 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
Implementation

After collecting all necessary data as shown in Chapter **??**, I will now begin to analyse this data. In this chapter the approach for several attacks, as listed in Section **??** will be introduced and the attack's goals recapitulated. The possible applications for the gained knowledge will be stated and the implementation of and requirements to the respective algorithm will be explained for each attack.

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 know about the holiday or sick leave times of a person. To know if a persons tends to become sick more 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 employers this might be 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 persons. This attack could even be performed by an outsider on a commercial open-source project, if the employees of the targeted company are known. The information gained by this attack could be quite delicate, as it could reveal relationships between persons. 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 [**article:spiegel-mining**].

3.1.1 Implementation

One requirement 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 needs 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.

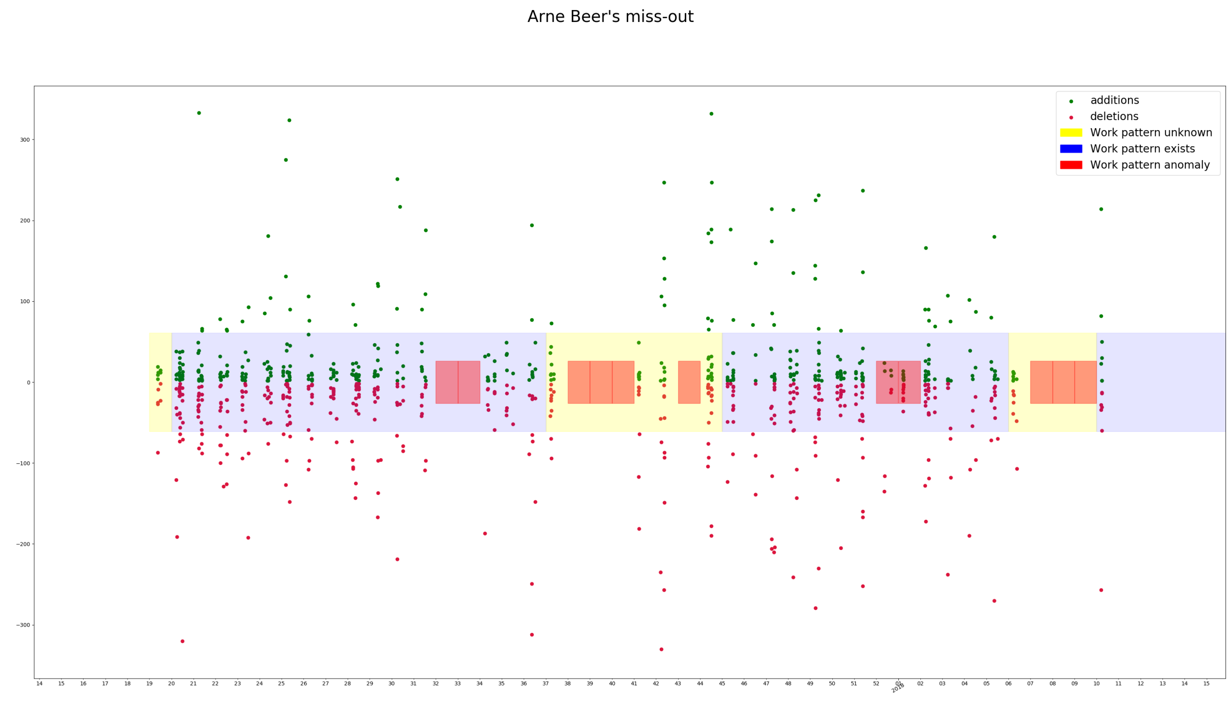

Figure 3.1: The work time analysis of the author.

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 an easier to handle format. 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. The preprocessed data is thereby equivalent to the days a contributor worked on, ordered by the week of the year.

```python
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 algorithm.

The algorithm inspects every week work pattern of a given interval. At the beginning a new *prototype* is tried to be found, which is performed in the function `find_prototype` in Listing??. A prototype is a representative week work pattern, which resembles the average work day pattern of the next weeks.

The function 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 Working hours

This attack aims to extract information about the working hour behaviour of a target. This should be achieved by displaying the pattern of the target in form of a weighted scatter plot sorted by hour per weekday and by comparing those patterns between several targets. There are several possible vectors for this attack:

- Information about the sleep rhythm of the target.
- Detect whether the target is a person working regular shifts from from Monday to Friday or rather an open-source contributor working in their leisure time.
- Detect anomalies such as automated programs, which contribute to a project on a regular basis.
- Compare the working hour patterns of several people in the same project or organization. This could, for instance, be used to infer relationships between colleagues, based on an equal working shifts.

Additionally a clustering will be performed to find anomalies, common patterns and to evaluate the results of this analysis. As we are only interested in contributors with a representative amount of commits, all contributors with less than one hundred commits in the last year have been excluded. This reduced the amount of considered contributors from 175.000 to about 10.300.

3.2.1 Implementation

The initial data for this analysis are the commit timestamps of the target, as well as the Github employee information for verification. These commit timestamps are then converted into a different format, which represents the occurrences of commit per hour per weekday over the last year. The result is a simple vector with length 168, which corresponds seven days with 24 hours each. I will refer to this representation hereafter as a *punchcard*.

```python
def preprocess(commits):
    punchcard = [0] * 168
    for commit in commits:
        hour = commit.commit_time.hour # returns 0-23
        weekday = = commit.commit_time.weekday() # returns 0-6

        index = weekday*24 + hour
        punchcard[index] += 1
```

Listing 4: Data preprocessing

The data transformation is achieved by incrementing the field of the respective weekday and hour by one for each commit, as can be seen in **??**. The resulting punchcard vector is then stored in the database for faster and easier analysis in the next steps.

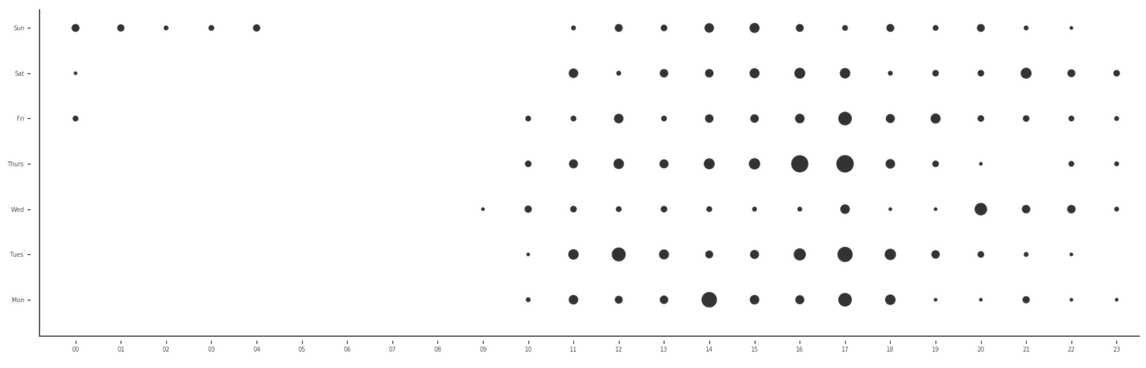

Figure 3.2: Punchcard of the author.

3.2.2 Punchcard Clustering

To find common work patterns, several cluster cluster algorithms have been performed on the aggregated data. The Python *scikit* framework has been used for this purpose, as it features nine different clustering methods and provides good documentation and abstraction from the underlying clustering logic [1]. For the task of finding similar punchcard patterns in the data, a clustering algorithm is required, which can operate on a high-dimensional dataset with an unknown amount of clusters Scikit provides three different clustering algorithms, which can handle an unknown amount of clusters.

Mean Shift

Mean shift is a clustering methods, which performs an operation similar to a gradient descent, which shifts all adjacent data points to their center [**article:mean-shift**]. The

[1]'Clustering' scikit-learn.org, http://scikit-learn.org/stable/modules/clustering.html (accessed, 24.04.2018)

goal of this algorithm is to find a representative centroid for each cluster and assign each data point to a cluster. This methodology proved to be too aggressive for the current data.

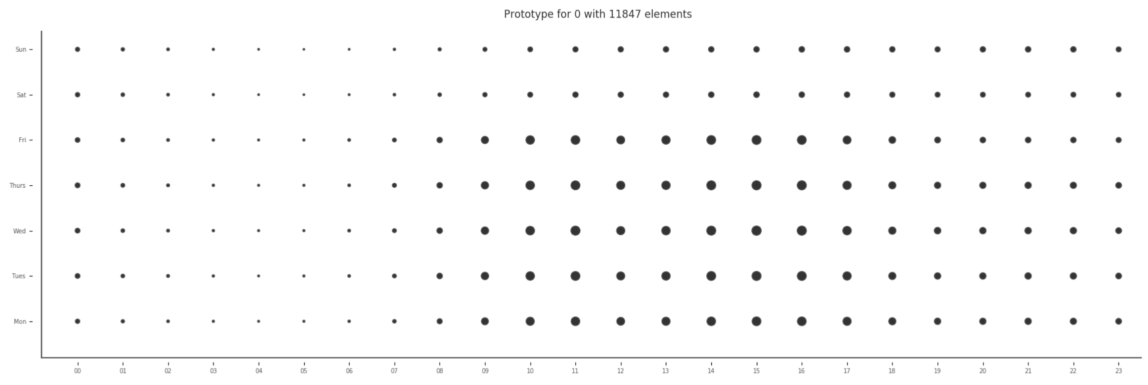

Figure 3.3: Super cluster centroid found by mean-shift clustering.

Despite trying a wide range of values for the bandwidth, which is the measure of distance used for detecting adjacent points, this algorithm always created a super cluster, which contained more than 89% of all data points. Such a super cluster can be seen in Figure **??**. The other 11% were invariably small clusters representing extreme outlier or strange patterns, which do not resemble any of the expected patterns for normal work shift or leisure time developers. An example for an extreme outlier can be seen in Figure **??**.

My assumption is that the density for the major part of the provided data is too high and equally distributed around the centeroid of the super cluster. Thereby all those data points are slowly shifted to this single centroid. As it is difficult to debug 168-dimensional space, I decided that an profound analysis would be too time consuming and to try the next solution.

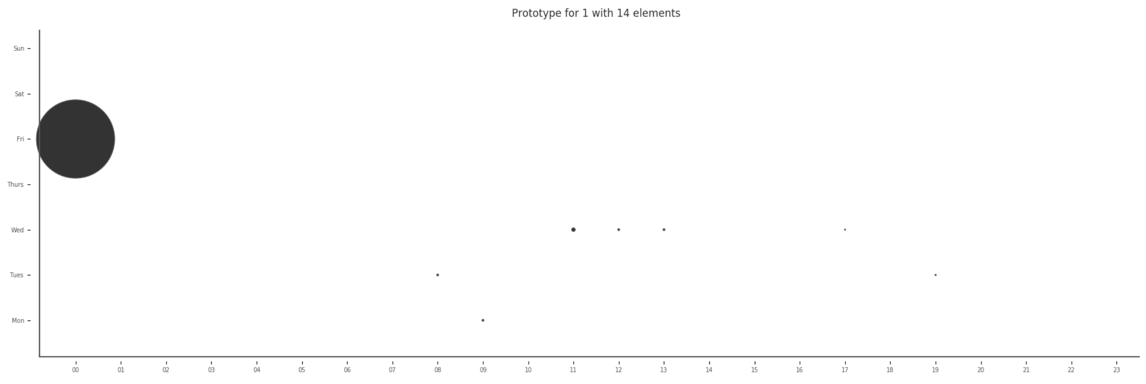

Figure 3.4: Extreme outlier centroid found by mean-shift clustering.

DBSCAN

The **dbscan!** (**dbscan!**) clustering algorithm operates by creating clusters of transitively connectable data points with a maximal distance between each point. **??**. It is highly scalable and performant, even for large data sets, because of this scalability it was my first choice. Unfortunately it produces very similar results to the mean shift clustering algorithm **??** as it finds a super cluster very similar to Figure **??**.

I assume that this algorithm suffers from similar problems as the mean shift approach, which are high density of data points without clear borders. Thereby this methodology manages to most part of all data points transitively from a single starting point, but creates a huge amount of mini clusters, when supplied with smaller values for the maximum distance between data points.

This algorithm also manages to find extreme outlier clusters, but it is not suitable for the purpose of this thesis, due to the extremely low granularity on the data inside the super cluster.

Affinity Propagation

The Affinity Propagation algorithm considers similarities between all data points to find clusters [**article:affinity-propagation**]. This clustering algorithm features a promising approach, as it utilizes a method similar to message passing, to find an *exemplar*, which resembles the representative of a cluster, and its surrounding cluster member. Affinity Propagation was the only available clustering method that was detailed enough to find interesting patterns without creating a super cluster. About 200 different patterns have been discovered using this methodology. However it has to be noted, that this clustering method is sometimes a little too detailed, as it, for instance, split very similar patterns into two or more different clusters. Additionally the memory requirements for this method scale quadratically, for non-sparse sets, with the number of the data points [**article:affinity-propagation**]. About 13 **gb!** memory have already been used with a sample of roughly 10.000 data points. This algorithm becomes thereby impractical for analyses on the whole dataset, but it works for smaller analyses and is thereby suitable for the validation of this thesis.

3.3 Geolocation

This attack aims to extract information about the actual location on the globe of a target. The algorithm tries to firstly determine the exact timezone of the target and then to exclude any countries or states, which do not match the observed timestamps. There are several possible vectors for this attack:

- Detect the main country or timezone of a target.
- Analyse where and how often a target travels to a specific location.
- Determine the location of a workforce of a company.

3.3.1 Implementation

The data used for this analysis are all commit timestamps of the target, as well as the target's Github location for verification. In the following I will explain the algorithm used to detect the different timezones and determine the main location of the contributor. Several external data sources have been used to accomplish this:

IANA Database
The **iana!** (**iana!**) provides a free to use database, with all timezones and the respective **dst!**s (**dst!**s) switches for each year. It also provides the exact **utc!** offset and offset switches for all timezones and thereby a relation to all countries.

Natural Earth
The Natural Earth organization provides extremely detailed free to use and up-to-date geological data with timezones, countries and even states on a resolution down to 1:10m.

Pycountry

As the codes and names used by the **iana!** database and Natural Earth don't always are identical, another layer with more information about countries and country codes was necessary. To match non-assignable timezones to their respective country on the map, the library *pycountry* was added.

OpenStreetMap and geocoders

As the codes and names used for states and provinces by the **iana!** database are not necessarily identical to the Natural Earth database, another solution for getting the relation between timezone and state was necessary. To match non-assignable timezone strings to their respective state, Gitalizer occasionally issues requests to the *OpenStreetMap* **api!** with help of the python *geocoder* library.

To assign **utc!** offsets to their possible timezones a special reverse mapping of the existing **iana!** database was necessary. The python library *tzinfo* provides interaction with the **iana!** database, but this adapter is only capable of resolving timezones to their respective **utc!** offset.

As a result I wrote my own adapter, which extractes the data from **iana!**, with help of tzinfo, and saves it into the Gitalizer database. The database model is named *TimezoneInterval* and contains the timezone identifier, the **utc!** offset and the exact start and end of this specific timezone interval. This table only needs to be populated once, but it needs to be updated as soon as a new version of the **iana!** database is released.

```python
def get_travel_path(commits):
    travel_path = []
    current_location = None
    last_valid_location = None
    change_at_day = None
    location_candidate = None

    for commit in commits:
        commit_time = commit.commit_time
        zones = find_timezones(commit_time, commit.commit_time_offset)

        # Create the initial timezone
        if current_location is None:
            current_location = {
                "set": set(zones),
                "start": commit_time.date(),
                "end": commit_time.date(),
            }
            last_valid_location = commit_time.date()

            continue

        # Get possible timezone candidates for this commit and intersect them with
        location = set(zones)
        intersection = location & current_location["set"]

        # Check if the possible timezones of this commit matches any timezone of th
        if len(intersection) > 0:
            # By reassigning the intersected set we gain additional precision by co
            current_location["set"] = intersection
            current_location["end"] = commit_time.date()
            last_valid_location = commit_time.date()

        # There is no match between the possible timezones and the current set.
        # In this case we need to check if this is a single occurrence (anomaly) or
        # if this is an actual change.
        else:
            # No change_at_day exists, but we detected a change
            # Remember the change. If this change lasts for at least a day it will
            if change_at_day is None:
                change_at_day = commit.commit_time.date()
                location_candidate = {
                    "set": set(zones),
                    "start": commit_time.date(),
                    "end": commit_time.date(),
                }

        # No change detected
        if change_at_day is None:
            continue

        # There was an anomaly, but not for a whole day.
        # This could for instance be a developer committing from a remote server.
        if change_at_day <= last_valid_location:
            change_at_day = None
            location_candidate = None

            continue

        # The change is not older than a day
        # ignore it until the change lasts for longer than a day
        if change_at_day <= last_valid_location:
            continue

        # There exists a change from the last day.
        duration = current_location["end"] - current_location["start"]

        # The current_location set only existed for a single day.
        # This is most likely an outlier. Thereby drop it and restore the previous
        if duration < timedelta(days=1) and len(travel_path) > 0:
            last_location = travel_path.pop()
            last_location["end"] = current_location["end"]
            current_location = last_location

            # Check if the old location and the current candidate actually match
            # If that is the case drop the candidate and completely replace the cur
            intersection = location_candidate["set"] & current_location["set"]
            if len(intersection) > 0:
                # Update current_timezone
                current_location["set"] = intersection
                current_location["end"] = commit_time.date()

                # Reset candidate and last_valid_location occurrence
                last_valid_location = commit_time.date()
                change_at_day = None
                location_candidate = None

                continue

        # We detected a change and it seems to be valid.
        # Save the current timezone and set the candidate as the current timezone.
        travel_path.append(current_location)
        current_location = location_candidate
        change_at_day = None
        location_candidate = None
        last_valid_location = commit_time.date()

    current_location["end"] = datetime.now().date()
    travel_path.append(current_location)
    return travel_path
```

Listing 5: Algorithm used to detect changes in the target's location by analysing the **utc!** offsets of git commit timestamps

The algorithm in Listing **??** iterates through every commit and determines in which timezone the contributor could have been at commit time. For each following commit it is checked, if there is a intersection between the possible timezones of the last commits and the current commit. This is usually the case, if the contributer did not travel to another timezone. But it is possible that a change in the timezone happens, even though the contributor did not travel. This is due to **dst!**, which is something that can be used to improve the precision of the location.

Germany for instance enforces **dst!** and switches between the **utc!** offsets +1 and +2. Angola on the other hand does not have **dst!** and thereby has a continuous offset of +1. For instance, if commits during a small time interval in the winter from a German contributor are considered, it cannot be determined whether he lives in South-Africa or in Western Europe. But if the commits of a whole year are considered, it can be concluded, due to the intersection of the possible timezones of previous commits, that the contributer has to be in a country, which enforce **dst!** and switches between the offsets +1 and +2.

In case no intersection between the timezones can be found, it needs to be determined, whether the contributer actually committed or the change happend through some other event, such as the commit from a remote server in a different timezone. For this purpose, all timezone switches, which do not stay longer than a day or happen at the same day as a commit from the previous location, are marked as insignificant and ignored.

The algorithm returns a chronological list of all detected and as significantly ranked locations with their respective time interval.

```python
def find_home_location(travel_path):
    home_location_candidates = []
    home_location = None
    found = False

    # Try to find the current home location and narrow it down as good as possible:
    for location in travel_path:
        duration = location["end"] - location["start"]

        # Try to find a set which intersects with the current set
        for candidate in home_location_candidates:
            intersection = location["set"] & candidate["set"]

            # Found an intersection, set the new intersection and increment days
            if len(intersection) > 0:
                candidate["set"] = intersection
                candidate["days"] += duration.days
                found = True
                if candidate["days"] > home_location["days"]:
                    home_location = candidate

                break

        # Found no matching location, create a new candidate
        if not found:
            location["days"] = duration.days
            home_location_candidates.append(location)
        else:
            found = False

        if not home_location:
            home_location = location

    return home_location
```

Listing 6: Methology used to determine the main location of a target, based on the information gained from the function in Listing **??**

To detect the home location of a contributer, the algorithm in Figure **??** is used. The parameter provided to this function are the results of function get$_{t}ravel_{p}ath$ in Figure **??**. The algorithm simply tries to determine the best possible set of timezones, which exists for the longest duration.

During this process further intersections of matching timezone sets are made to further increase the precision of the home location. The result of this function is a non empty intersection of timezone sets, which persisted for the longest time period, compared to all other possible non empty intersection sets.

Chapter 4
Evaluation and Interpretation

In the last chapter I showed the implementation of several possible attacks, which could be performed on the gathered data. This Chapter will now attend to the evaluation of all results gained from these attacks. I will present the extracted information from each algorithm and compare it to real world ground truth. This information will be then be explained and audited in terms of precision and reliability.

4.1 Holiday and Sick Leave Detection

Figure **??** 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 verification 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 **??** I was forced to reduce my working hours due to legal questions and continuously shift hours and working days for several weeks. It is hard to interpret those inconsistencies without more contextual information, but nevertheless it provides the fact that something happened during this time.

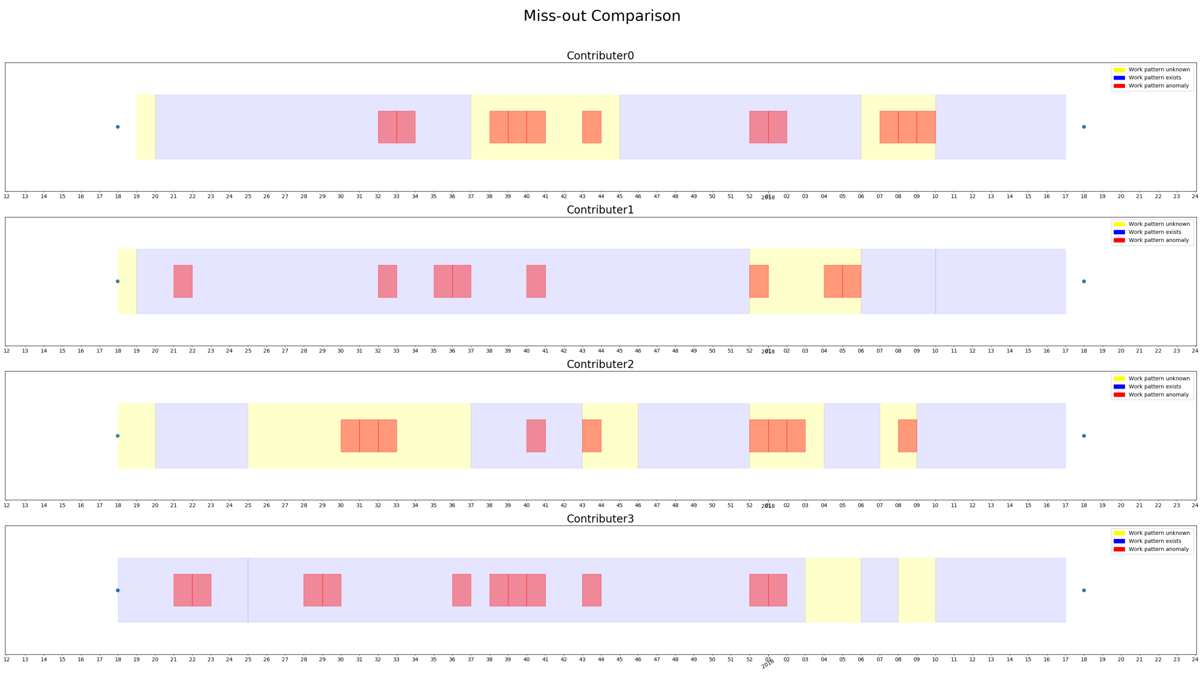

Figure 4.1: The miss-out analysis of several employees.

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

4.2 Working hours

4.2.1 Sleep rythm

To evaluate the significance of the punchcard in terms of sleep rhythm analysis, a small survey in a closed community has been made. A selected group of ten people, who know each other well has been selected, for this purpose. Furthermore a subset of four people has been chosen, which were going to be evaluated. All ten people then needed to assign the punchcard of the chosen subset to a specific person.

To this end, three quite similar patterns, where one contributor has a very regular sleep rhythm, Graph 2 in Figure **??**, and one pattern of a practically inexistent sleep rhythm, Graph 1 in Figure **??**, have been selected.

The results of this survey where just as expected. For the three similar graphs, no significant results could be assessed, as the assignments where more or less random. The contributor with the irregular sleep rhythm on the other hand got correctly assigned in all in every survey.

Sadly, as such an survey needs very specific targeting and a long lead time for data collection, it could only be executed on a small set on subjects. Anyhow the result of this survey proofs, that there exists a correlation between the structure of the punchcard and the actual commit and sleep behaviour of an contributer, even if it can only be accurately assigned in extreme cases.

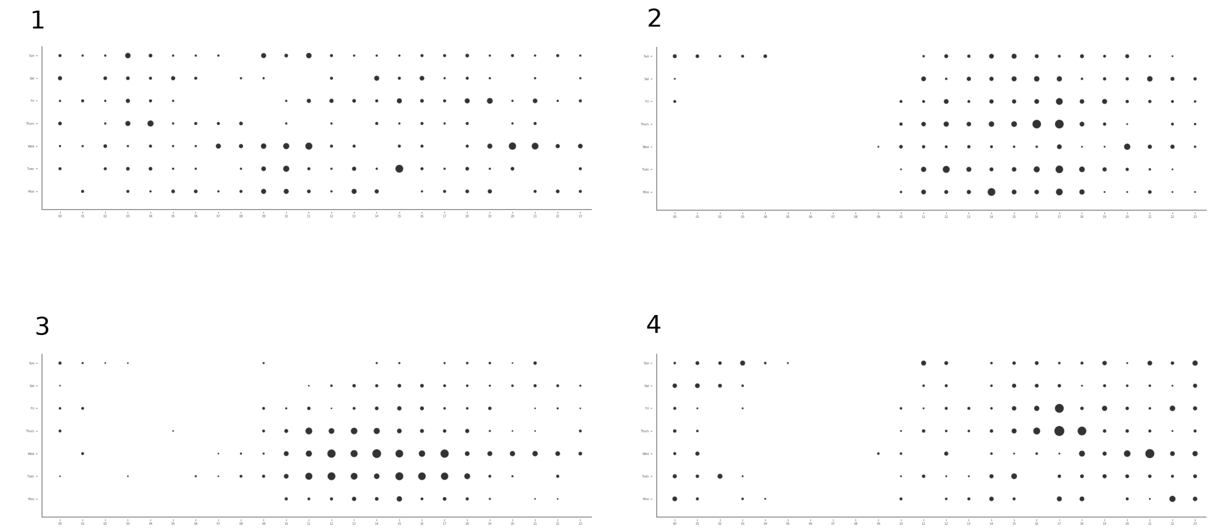

Figure 4.2: Punchcard of an user without an irregular sleep rhythm.

4.2.2 Employee or Open-Source Contributor

To determine whether a punchcard could be used to distinguish between an employee or an open-source contributor, the results of the clustering described in Section ?? has been utilized. For this approach two assumptions have been made. An usual employee works between Monday and Friday during the day and only as an exception at the weekend. An open-source developer works outside of the usual work shifts, which means early and late during weekdays and at the weekend.

For each assumption two representative clusters have been chosen and ten random persons have been selected for each cluster. The manual verification is conducted by checking if the contributor mainly contributes to repositories which belong to the registered employee. In case no employee exists, it is examined whether the contributor pushes to their own or open-source projects or rather to the repositories of a specific company.

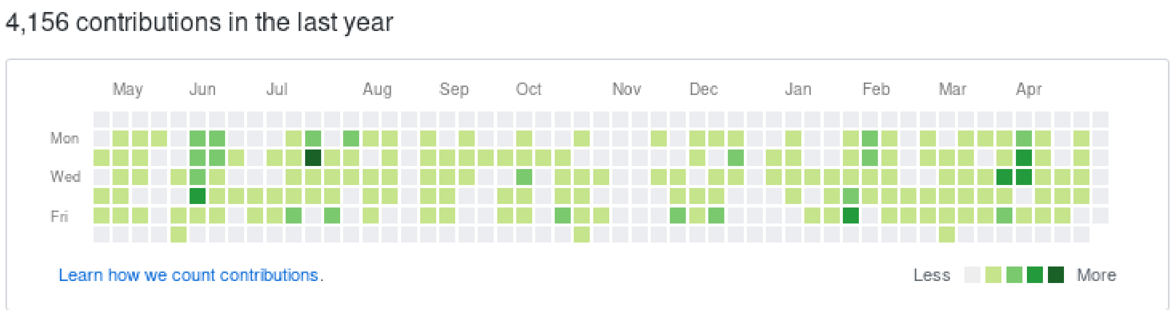
4,156 contributions in the last year
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
Mon
Wed
Fri
Learn how we count contributions.
Less
More

Figure 4.3: Github contribution overview a Google developer with the Github nickname alxhub.

It provides a good overview of the usual weekday work pattern over the last year and allows to quickly inspect the repositories a contributor committed to at a specific date.

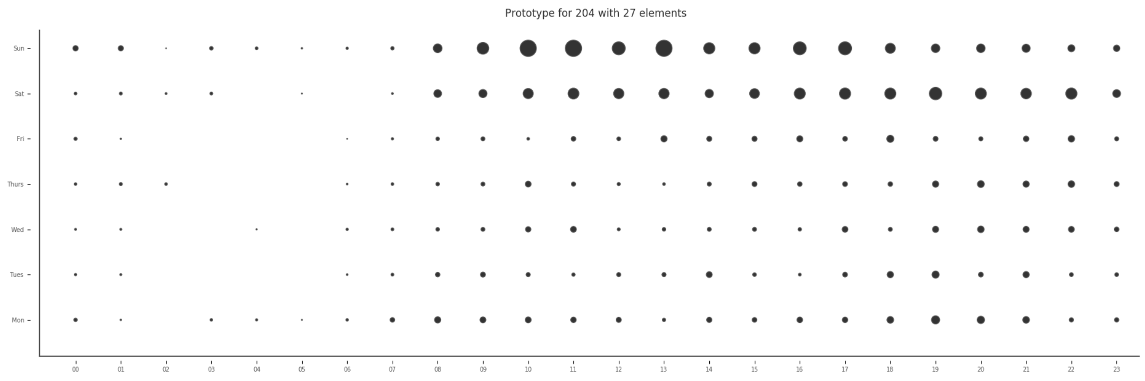

Figure 4.4: Punchcard of an example from an affinity propagation cluster with a weekend tendency.

The representatives for the usual five-day week commit behaviour were surprisingly accurate. About 93% of considered contributors were mainly working on projects of their companies, with occasional commits to their open source projects. The remaining 7% were either open-source developer with a very consistent commit behaviour, or it could not be determined if they work for a company.

The representatives for the leisure time commit behaviour are mostly correct as well. About 88% of considered contributors were irregularly contributing to either work unrelated open-source projects or their own projects. Approximately the half of those contributors specified their employee and didn't work on their employees' projects. The other half worked either on their own projects or on open-source projects on an irregular basis. The remaining 12% were either contributors working and committing to their employee's projects, but also to their own and open source projects, and employees with an untypical commit behaviour.

This analysis shows quite well, that there is a correlation between the assumed patterns and the github commit behaviour or the employment status of contributor. Unfortunately, the evaluation process for these results is very time consuming and thereby only a relatively small sample group has been chosen. As it is not trivial to link the employee of an contributor to all their funded projects, all verification needed to be conducted manually.

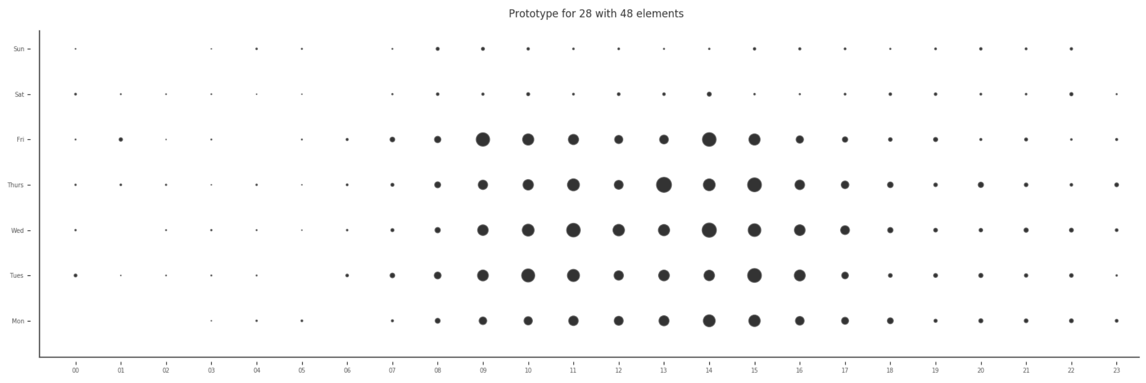

Figure 4.5: Punchcard of an example from an affinity propagation cluster with normal work shifts.

4.2.3 Bot detection

Another possible attack was the detection of automatically committing programs, so called *bots*. This algorithm simply detected centroids with an extremely equally distributed pattern or patterns with a spike at a specific hour. After a manual revision of the by the algorithm detected clusters, it became apparent, that only a small subset of those clusters actually contained bots. And even if the cluster contained bots, there were usually only one or two of them from a much larger pool of cluster members.

Detection of bots in the outliers, which were not assigned to any cluster, did not seem to be promising as well. Manual revision of over 50 possible candidates lead to not a single bot. After reviewing these results, I decided, that there is currently no viable approach for my data to this problem.

4.2.4 Fingerprinting

Another possible attack was to fingerprint a contributor, by analysing their commit behaviour and thereby creating a unique identifier.

This attack soon prooved to be unfeasible, as the pattern of a contributor significantly differs from month to month. There even are significant changes, if the interval of a year is considered and the shift is only by a month. The commit behaviour of people seem to be too inconsistent for a fingerprint.

List of Figures

List of Listings

List of Tables

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