Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor thesis

Privacy implications of exposing Git meta data

presented by

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

submitted on May 21, 2018

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

Abstract

Software developers use many tools during their daily work and expose lots of data, often without being aware of doing so. This data could be used to surveil, to spy on or to influence these developers.

Recent events as the Facebook scandal, in which the data of several million people has been exposed to a consulting company [1], show how data can be abused to extract valuable knowledge and can be 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 (VCS) *Git*. Simple metadata such as UNIX timestamps and email addresses might be enough to extract sensitive information about users or organizations using Git. This paper covers the whole process of gathering data from a vast amount of Git repositories, through to preprocessing, generating 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 metadata and to proof that it can be used maliciously.

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

Gotta go fast.

Senic the Herdgherd

Contents

Acronyms

API Application Programming Interface

CET Central European Time

CEST Central European Summer Time

CPU Central Processing Unit

CVCS Centralized Version Control Systems

DBSCAN Density-Based Spatial Clustering of Applications with Noise

DST daylight saving time

EU European Union

FS file system

GDPR General Data Protection Regulation

GB Gigabyte

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

I/O Input/Output

JSON JavaScript Object Notification

MIT Massachusetts Institute of Technology

ORM Object-Relational Mapping

OS Operating System

SHA-1 Secure Hash Algorithm 1

SSH Secure Shell

SQL Structured Query Language

TB Terrabytes

URL Uniform Resource Locator

UTC Coordinated Universal Time

VCS version control system

Chapter 1
Introduction

Git is a VCS used by most programmers on a daily basis these days. Its purpose is to help developers version and manage the code base of their projects. According to the Eclipse Community Survey, about 42.9% of professional software developers used Git in 2014 with an upward tendency [20]. It is deployed in many, if not most, commercial and private projects and generally valued by its users. On top of this, it allows collaborating with thousands of contributors on the same project whilst maintaining a clear version history.

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 and a total of 67 million repositories [8].

Some well-known projects and organizations use Git, for example, Linux, Microsoft, Ansible, and Facebook [8]. Each of those repositories contains the complete contribution history of every contributing user and every contribution contains all changes, a timestamp, a message from the author and their email address.

This raises the question how much information is hidden in this metadata of a Git repository and which attack vectors could be introduced by analyzing this information. Could it be used to harm or manipulate a contributor or maybe even a company?

The 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 through their open-source projects. It is also imaginable, that a private individual uses this data to monitor another person, who regularly contributes to open-source repositories.

As there have not been any papers published about this specific topic yet or at least no public paper and as Git plays such a crucial role in today's software development, I want to investigate and evaluate this potential threat. Furthermore, I want to create a foundation for future research and provide a first example of how such attacks might look like.

1.1 Motivation

Each year more and more data is collected by employers. A study conducted by Massachusetts Institute of Technology (MIT) students shows, that data-driven companies, that collect data about everything in their company, are about 5% more productive [18], but this should never justify any invasion into their employees' privacy.

Privacy violation already goes as far as tracking medical records of employees, which is a service provided by the company *Castlight* [22]. Some parties warn against surveillance of employees by the management and demand stricter handling of employee data [12]

Generally, it can be said, that we need to take better care of our personal information and that people need to know in which ways they can leak information. This thesis aims to show this on the example of the VCS Git, which is by itself a very handy tool for code versioning of a project. It was not developed with any malicious intent, but it might be used for such.

1.2 Leading Questions and Goals

The primary research objective of this thesis is to find out if data mining on Git metadata would be feasible. This question is explored by looking at various data mining techniques and applying them to aggregated data from Github.

To gather data and to perform analysis on the data for the purpose of this thesis, a program called *Gitalizer* is developed. It is capable of continuously collecting data from Github in several ways. Furthermore, three different analysis methodologies and several evaluation methods for these are implemented.

1.3 Outline

This thesis examines the process and capabilities of building a data mining software based on Git metadata. The Introduction will explain the motivation, the approach and the goals of this thesis. In Chapter 2 three different attack models will be constructed to determine possible attack goals. In this context related work will be mentioned with respect to the attack goals. Chapter 3 will explain the requirements to the data, existing solutions and the process of building an own aggregator, which collects data from Github. Chapter 4 shows the actual approach and algorithmic implementation of three different attacks. Chapter 5 evaluates the results of the previously presented algorithms by comparing it with real-world ground truth and conducting small surveys. In Chapter 6 the overall results of this research will then be discussed and an outlook will be provided.

Chapter 2
Attacks Goals and Related Work

This chapter will show the creation of several attack models, which are used to design possible attacks and outline the requirements to the data needed for these tests. In the same context, several related works will be mentioned in reference to the particular attacks. Additional related work, which is not directly related to any attacks is listed in a separate section.

2.1 Attack Models

In the following, I will present three attacker models, which were used to design the attack goals in the next section. Each attacker model represents an interest group, which might use Git metadata to surveil, spy upon, manipulate or harm a target. A number of possible valuable information about a target, which might be obtainable by analyzing Git metadata, are stated for each model.

The Employer

This attack model deals with the scenario of an employer who wants to monitor their employees. The attacker's motivation is to spot irregularities in working behavior and thereby unmotivated or unproductive employees. Information gained by this attack, such as productivity metrics of employees, compliance with working hours and sick leave, could be used to surveil employees without their consensus.

The Individual

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

A possible goal of the attacker could be to either stalk the victim, cause harm in any way or manipulate it or one of their acquaintances. The motivation of this attacker is mostly personal and on an emotional level. For this purpose, the attacker could use information about the target such as relationships to other developers, sleeping rhythm, and daily routines.

Another nonemotional attacker motive could be a robber trying to find the perfect time window to rob a house. Information about the geographic location of the target at a specific time or knowledge about when the target is at work could be used for this purpose.

A third attacker motive could be a headhunter, that tries to get information about the skills and reliability of a developer. Several metrics, such as productivity, sick leave tendencies, geographic location and daily routine could be used for this purpose.

The Industrial Spy

This attack model covers the scenario of an external person, who 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 a 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 full product or at least parts of their product as open-source software.

Valuable information for this attacker is, for instance, a list of company employees, the company employee history, the geographic location of the company's workforce and internal team structures of the company.

2.2 Attack Goals

This section attends to the establishment of several attack goals, which could be pursued by an attacker. These goals serve as a guideline for the data aggregation process, which will be covered in the next chapter. It needs to be noted, that only a few these listed attacks will be actually performed in the scope of this thesis, but this listing also serves as an exhibition of some possible attacks for anyone that wants to further investigate this topic.

Productivity of Employees

An employer wants to ensure that their employees work sufficiently. For this approach, several values could be used to create a metric of quality and quantity of work.

In [11, p. 3] simple productivity measurements such as counting the contributed lines of code and the number of function points are evaluated. It is stated, that these measurements indeed provide a metric for the quantity of code, but not about the actual quantity of work and quality of the code. In [14, p. 43] the author recommends to also consider the number of removed defects from the code. The authors of [1, p. 257] also include code style quality measurements such as *cyclomatic complexity*, *coupling* and lack of cohesion of methods.

The gained information by this attack could, for instance, be used to compare the productivity of several employees with the intent to dismiss all employees that do not perform well enough in relation to their coworkers. Another possible use case could be the revelation of developers with a specific skill set for headhunters. The data needed for this attack are the additions and deletions of all commits as well as all commit timestamps and the full patches of each commit.

Compliance of Working Hours

The aim of this attack is to allow employers to check whether an employee is productive in the given working hours. This might be especially useful to supervise employees, that work remotely and cannot be locally monitored.

In [4] a survey on repositories of *Mozilla* and *Apache* is conducted, to detect at what time their developers work. For this purpose, all commit timestamps of those repositories have been collected and analyzed. Their survey discovers that about 66% of conducted developers follow office hours.

The data needed for this attack are commit timestamps of all employees' commits.

Sleeping Rhythm and Working Behaviour

This attack aims to understand and predict the victim's sleep rhythm and working behavior. This information could also be used to detect whether the target is a person working regular shifts from Monday to Friday or rather an open-source contributor working in their leisure time. The data needed for this attack are commit timestamps.

Personal Relationships to Various Programmers

The objective of this attack is the detection of relationships between various contributors by simply analyzing Git repositories. This information could, for instance, be used by a rogue person to perform social engineering attacks based on the gained knowledge.

A similar topic has been conducted in the study of [23]. The authors try to detect a correlation between social interaction and different measures, such as written code and the Github *following* mechanic.

The data needed for this attack are commit timestamps as well as the full Git history graph of the respective repositories.

Sick Leave and Holiday

The aim of this attack is to detect anomalies in the typical work behavior. The detection of anomalies in the regular work pattern can be a valuable information for several parties. Usually, only a few of parties know about the holiday or sick leave times of a person. To know if a person tends to become sick more often or for long times is a dangerous intrusion into a person's privacy. For instance, this could be abused by headhunters 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 for detecting anomalies in the productivity of employees. In case an employee does not commit on a regular basis for several days, this behavior could be detectable 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 of the popular German weekly magazine *Der Spiegel* written by the David Kriesel [16].

The data needed for this attack are commit timestamps.

Geographic Location

This attack aims to detect the location of a target at a given time. The goal is to detect the home country of the developer or at least to narrow the location down to a timezone or to a set of countries. Another goal is to identify all significant detectable changes in the developer's location.

For instance, this information might be critical for an individual that wants to be as anonymous as possible. Revealing the home location of the target can suggest further information, such as the cultural origin and religious orientation. The results of this attack might also provide the attacker with a history of the target's travels. This could be used as an additional measure in the detection of relationships between contributors.

The data needed for this attack are the target's commit timestamps.

Company Employees

The goal of this attack is to detect employees in the repositories of a company. A motivation for this attack could be to detect company members for social engineering attacks or to headhunt these employees. This attack could also be used to detect team structures of companies and the respective role of each team and their team members.

In [13] a very similar approach on VCS data is conducted. They try a new fine-grained methodology to automatically detect communities in Git data. They do this by analyzing the VCS data from their considered projects. Their detected communities show a high accordance to the real world communities for those repositories [13, p. 10].

The data needed for this attack are the Git commit history graphs of the respective repositories.

Employment History

This attack aims to detect the timespan for which an employee worked at a given company. The knowledge about a company's employment history 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.

To perform this attack the employees of a company need to be known, it is therefore dependent on the previous attack. The subsequent proceeding is rather simple as only the first and the last commit of an employee needs to be detected. The data needed for this attack are git commit timestamps.

2.3 Further Related Work

In [21] several data mining approaches for Github are evaluated. The authors analyze the data provided by Github and present several existing solutions for aggregating data. One of their research goals is the analysis and evaluation of data mined from Github by the *GHTorrent* project. They further investigate the developer knowledge passed between different repositories by applying a novel visualization technique on the datasets.

The authors of [15] evaluate the usability of data mined from Github for scientific purposes. They warn about several possible flaws in the data, depending on the research goal. These perils include, for instance, that the major part of repositories on Github are used for personal projects and that a repository does not necessarily need to be the official repository of a project, but can rather be a fork of it [15, p. 4].

Chapter 3
Data Source and Aggregation

This chapter will attend to the collection of required data as stated in Section 2.2. At first the VCS *Git* will be presented and its functionalities explained. The actual source of the data *Github* will then be evaluated in terms of the amount of ground truth and availability. At last the methodology used for aggregation and exploration of Github will be explained.

3.1 Git

This section presents 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. Moreover, current cases of application and some scenarios, that might be interesting for this thesis, will be mentioned.

Introduction to Git

At its core, Git is a tool that 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 [20]. 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 lines of code, namely the *history tree*, the *branch* and the *merge*. The commit history of Git is internally represented as a directed, non-cyclic, connected graph of commits. The commits act as *nodes* and the connection to their parent commits as *edges*.

If a single node has two children, a new *branch* has been created. In Git a branch is a pointer to a specific commit, which allows you to easily jump between and to distinguish versions. Git provides the feature to name branches, whereas the main branch is per default named *master*.

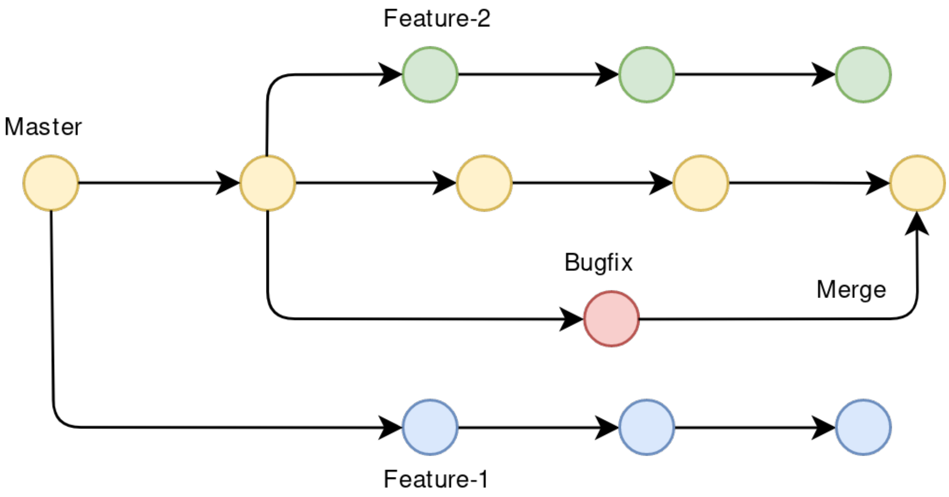


Figure 3.1: A Git commit history tree that shows a history graph with two feature branches and a bug fix that is merged into the master branch.

As shown in figure 3.1 two developers 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 often acts as a centralized repository, developers can *push* their changes to or *pull* changes from other developers. This principle is called a Centralized Version Control Systems (CVCS) and is supported by many VCSs [17]. It can, for example, be a distinct server, which is attached to some kind of network accessible by the developers. This feature allows developers to remotely contribute to a project, as long as they have access to the remote's network. Git also supports several protocols such as Hypertext Transfer Protocol (HTTP) or Secure Shell (SSH) to connect to a remote and thereby provide a simple user management layer.

Git User Roles

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

If one looks at the case of an author contributing code to a project via 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. The collected data of 49 million commits, indicate that in about 89% the author and the committer are the same person.

Internal Representation

Git's underlying storage and management solution for files is commonly described as a mini file system (FS) [2, p. 9] 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 [2, 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 a result, 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}. In 2017 Google managed to force a collision in a controlled environment, but it is really unlikely to encounter such a collision under normal circumstances [9]. This characteristic of SHA-1 hashing will become quite important in the design of the database later on.

As mentioned in the Section 3.1 Git is used to store the state of a specific directory of an actual 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 their 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: An example of a Git tree file.

As stated before the commit is utilized to provide an exact representation of a state of the repository's files and directories. For this purpose, it holds a reference to the tree object representing the root directory of the project as can be seen in line 1 in Listing 2. Just as blob objects, the tree and commit files are also stored in the `.git/objects` directory under their respective hash.

```
1   tree      cd7d001b696db430b898b75c633686067e6f0b76
2   parent    c19b969705e5eae0ccca2cde1d8a98be1a1eab4d
3   author    Arne Beer <arne@twobeer.de> 1513434723 +0100
4   committer Arne Beer <arne@twobeer.de> 1513434723 +0100
5
6   Chapter 2, acronyms
```

Listing 2: An example of a Git commit file.

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 timestamps with an Coordinated Universal Time (UTC) offset for the committer and the author.
- 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.

Relevant Features

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

checkout

Git allows jumping between different project versions with the *checkout* command. Calling `git checkout $IDENTIFIER` jumps to the commit specified by the variable `$IDENTIFIER`. The identifier, for instance, can be a partial or full SHA-1 hash of commit, a branch name or a tag.

HEAD

To mark the current checkout of a repository, Git utilizes a special marker called *HEAD*. Due to this feature, it is, for instance, possible 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 listing all files which changed between those commits as well as showing the exact changes in the files

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, that can lead to problems or irregularities in the gathered data.

force push

Git allows pushing 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 and change their branch pointer to the new commit.

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 also 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, such as 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 added to the repository. This command leads to similar problems as the rebase command since it can severely change the history of a repository.

3.2 Data source

The acquisition and clean-up of data was the biggest initial task of this thesis. 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 Application Programming Interface (API) will be given.

Requirements

The data source had to satisfy as many requirements as possible, as specified in Section 2.2.

To accomplish a meaningful analysis one needs a sufficient amount of commits. For instance, it is necessary to have a few commits per weekday over a timespan of at least 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 I had to get access to as many repositories, to which the targeted users contributed to, as possible. Thereby the data source has to provide a way to dynamically explore repositories around a single user or company.

For analysis of companioned persons as described in Section 2.1 it is crucial to find users, who 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 or to spy on company members, as described in Section 2.2, 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:

- Real world data
- Large amount of repositories
- Access to all commits of each repository
- Access complete metadata for each commit
- Email address 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

Github

I decided to use Github as a data source, as 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 [19] with 64 million repositories, 24 million users and 1,5 million organizations [8]. 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 fewer data to offer. Gitlab does not provide detailed usage statistics, but they state that they only host about 100000 organizations, which is remarkably less than Github [7].

On the other hand, one of the downsides of using Github is, that we do not have access to all metadata. For example, the full list of members for organizations is often inaccessible, as users need to actively opt-in to be publicly displayed as a member of the organization. The internal team structures of organizations are not visible at all, 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 anymore. All commits made with such an email address cannot be used any more for any analyses that require a user to commit relationship. But 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.

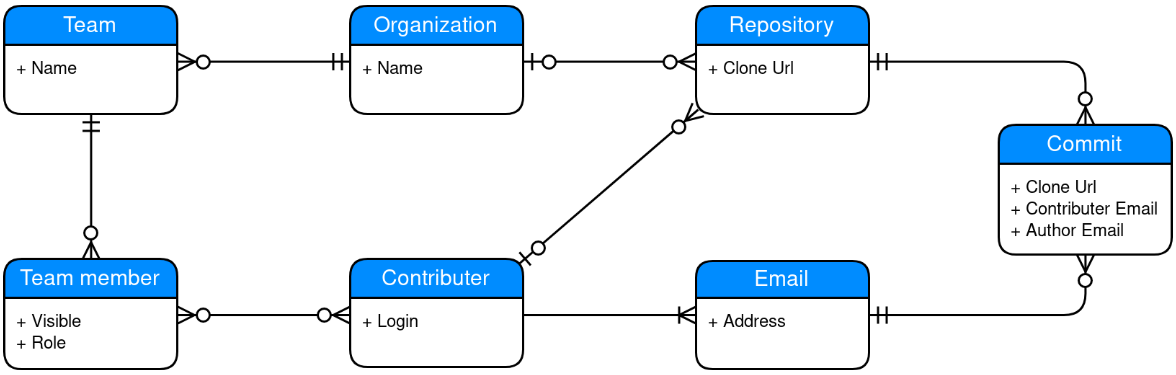


Figure 3.2: A simplified visualization of Github's internal relationships between the most important objects in Crow's foot notation.

Github's Features

In the following I will explain some of the features provided by Github, that cover the requirements listed in Section 3.2. Github offers some features, which, for example, are convenient to finding repositories a specific user contributed to or to find contributors hwo 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 comparatively large number 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 does not provide a method to get all repositories a user ever contributed to. However, Github provides an endpoint to query all starred repositories of a 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, when 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 repositories they contributed to, as well as some friends of the user.

Organizations

The last feature is *organizations*. An organization is used to host projects under an account, that is not necessarily led by a single natural person, but rather supports roles with different permissions and team structures. Github allows querying 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.

3.3 Data Aggregation

As mentioned in Section 3.2, 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

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

interaction with Github's API. In the end, some problems which occurred during the data collection will be shown as well.

Existing Solutions

There are of existing solutions for accessing and collecting Git metadata. In the following, the practicability of these solutions is evaluated based on our requirements.

GHTorrent

The *GHTorrent* project aims to provide Github's metadata to elude the limitations of Github's API rate limiting [10]. It provides representations for followers and commits, as well as organizations and organization members, but some crucial pieces of information are missing. GHTorrent only stores the main email address of a user and does thereby not support the handling of multiple emails, as commits are directly assigned to their respective Github user id. Commits miss information about additions and deletions in lines of code, which implicates that each commit would need to be scanned by a separate program again. Furthermore, GHTorrent does not have the concept of *starring*, which makes it hard to reduce the number of considered repositories to a manageable size. Their database provides about 4 Terrabytes (TB) of data according to their website, which is too much information without the possibility of specifying a precise subset of data [2]. It provides a vast amount of data, but at the same time, it cannot be ensured, that the data is as complete as possible for a specific user. Modifying the GHTorrent code base and extending their database schema has thereby been judged as impractical.

ghcrawler

Microsoft provides an open-source crawler called *ghcrawler*, which is supposed to continuously fetch data from Github [3]. Sadly their documentation is very incomplete and after diving into their source code, it appears that their crawler is for Github entities only and not for the underlying data of Git repositories.

Alitheia-Core

Alitheia-Core is a Java data collector for Git repositories. It is not actively maintained for more than three years and their documentation website is offline. Using this library seemed unpromising and unfeasible.

[2] 'The GHTorrent project' ghtorrent.com, http://ghtorrent.org/ (accessed, 05.05.2018)

[3] 'Github crawler' github.com, http://github.com/ (accessed, 05.05.2018)

RepoDriller

The *RepoDriller* project aims to support researchers by providing easy access to repository data from Github [4]. Despite providing a good solution for getting all necessary information from a repository, it provides no way to explore Github using *stars* or *following*. These features, as well as the assignment of emails to a contributor via the Github API, would need to be added. As the program is also written in Java and I am no longer familiar with the language and its ecosystem, I decided to stick to writing my own solution.

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*.

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.

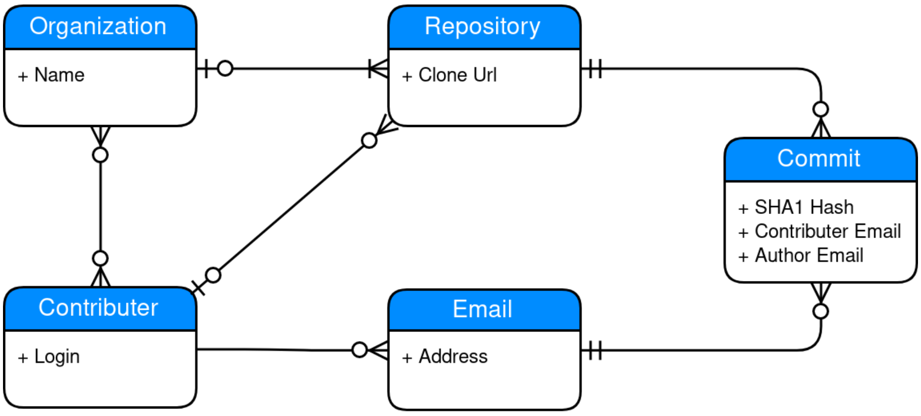


Figure 3.3: The relationships between Gitalizer's most important database objects in Crow's foot notation.

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 3.3 between commits and repositories. This feature

[4] 'A tool to support researchers on mining software repositories studies' github.com, http://github.com/ (accessed, 05.05.2018)

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 3.1 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. The formula for calculating the probability of such a collision is as follows, where p is the probability of collision, n is the number of different hashes and b is the size of the hash in bytes:

$$p \leq \frac{n(n-1)}{2} * \frac{1}{2^b} \tag{3.1}$$

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 cannot easily be identified as such. Considering Formula 3.1, the probability of a collision for 49 million hashes on the 160 bit SHA-1 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 since every contributor can obtain an unlimited amount of email addresses. It is necessary to connect all email addresses to a specific contributor, to unambiguously assign all commits to their respective contributor. This relationship does not have a `NOT NULL` constraint as it happens quite often that an email address cannot 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 3.2 Github provides a way to organize several people in organizations and teams. As one of the potential goals of this thesis is to see if it is possible to detect members of an organization in open-source projects, a model for organizations has been created as well. This data can then be used to check the results of this research's results.

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 3.3. For interaction with the Github API the *PyGithub* library is used. It 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, I will explain those approaches in detail.

Git repository

Gitalizer can scan any Git repository from a SSH or HTTP URL, as long as the user used by Gitalizer has access to it. The repository is cloned into a local directory. After the cloning is done, the actual scanning process begins. During the scan, the `HEAD` of the current default branch for this repository is checked out and every from the `HEAD` downwards reachable commit of the Git history is scanned. The program saves all available metadata for each commit in its database, which are emails, timestamps as well as additions and deletions to the project in lines of code.

After this scan, a lot of information is still missing. There exists no unique identifier of an author or committer since 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 a 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 the previous approach, I used the Github API to get some of the missing metadata. The general approach is the same as in the previous scan method. At first, the repository is cloned and locally scanned. However, a request is issued to Github every time an email is found, that is not already linked to a contributor. Github allows linking 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 their account or if the user completely deletes their account. If this happens and the email contributor relationship has not already been created in a previous scan, there is nothing that can be done and those commits need to be handled later on during the preprocessing of the data.

Github User

While trying to get 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, called the “repo-queue”, which is then processed by a multiprocessing pool of workers. Each worker process continuously pops entries from the “repo-queue” and scans every single repository as described in the previous approach.

Connected users and organizations

For detection and analysis of connections between contributors over multiple repositories, additional user repository discovery, as described in Section 3.2, 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 the previous approach. For this task, two different worker pools are utilized. The user discovery pool is initialized with a shared queue, called "user-queue", with all users we need to look at. This worker pool simply searches for relevant repositories of each user and passes the repository URL to the second shared queue. The second worker pool then processes the "repo-queue" as described in the approach for Github repositories.

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.

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 a migration to better hardware were necessary. The first considerable slowdown occurred after reaching about 12 Gigabytes (GBs) of data. At this stage, the database Input/Output (I/O) 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 SHA-1 hash as stated in Section 3.3 Ignoring forked repositories reduced the number of entries in the relation table between commits and repositories by another 26%.

Other 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 prevent data loss on a system crash. As there is no important or critical data handled, it was acceptable to pass on these mechanisms and to trade safety for performance.

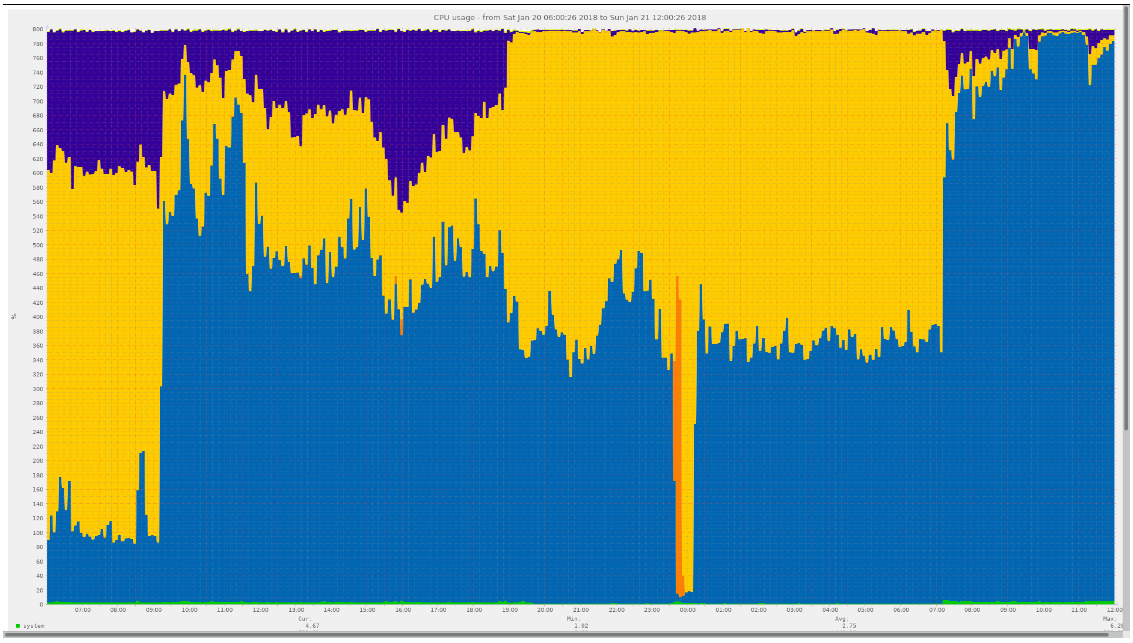


Figure 3.4: The CPU load of the aggregation server before and after query and cache optimization. The light blue area represents the CPU percentage actually used for computation. The dark blue area represents the amount of CPU idle time due to I/O load. At the beginning of the graph the server was partly unresponsive due to high I/O wait. At the right side of the graph the I/O load is significantly reduced after some optimizations.

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 3.4 the overall CPU load right before optimizing several SQL queries, by caching intermediate results and increasing the cache size of PostgreSQL, can be seen. 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.

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 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 commits can be found.

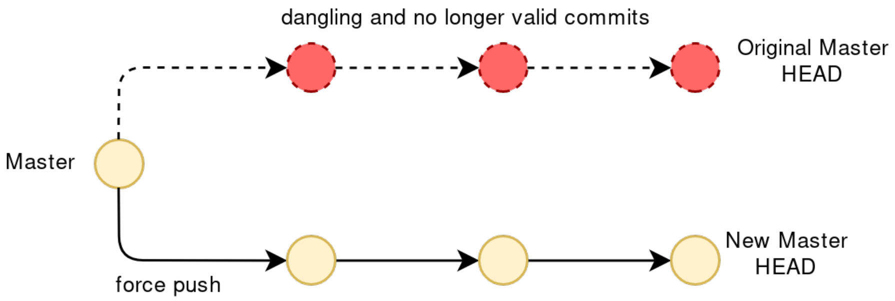


Figure 3.5: This figure shows an example of a Git history tree which master branch's history has been rewritten by a force push.

As explained in Section 3.1 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 leave dangling commits. As the complete history of a repository is stored in 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 have children, which are not in the newly scanned commits, a force push took place and the old commit history has to be flagged as such since it is now outdated and irrelevant. An example scenario can be seen in Figure 3.5, where all red commits represent the old commit history, which needs to be truncated.

Problems

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

The first version of the aggregator did not 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 had 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 of API calls issued by Gitalizer, to avoid waiting for a reset of the previously mentioned API limit. 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, that triggered a call to the Github API. Afterward, 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 *libgit2* library, used for interaction with Git, issued *stat* Linux syscalls during a diff operation for each file, which changed between the compared commits, to check if there were any local uncommitted changes. Anyhow the locally scanned repositories were cloned in *bare* mode. This means that there exists no project root directory, but rather only the git internal representations of those files, which makes the behavior stated above unnecessary and unwanted. As a result, all processes slowed severely down due to high I/O wait times, because of a huge amount of stat syscalls on non-existent files. Luckily after reporting the issue [5] it was resolved in a week and I was able to continue developing with my own compiled version of the libgit2 library.

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

Chapter 4

Attack Implementations

After collecting all necessary data as shown in Chapter 3, this chapter now attends to the analysis of this data. The approach for several attacks, as listed in Section 2.2 will be presented and their attack 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.

4.1 Holiday and Sick Leave Detection

The goal of this attack is to detect miss-out by looking for anomalies in the workday patterns of a developer.

Due to possible fluctuations or changes in the work routine, one requirement to this algorithm is the flexible detection of regular work patterns. It must have the ability to adjust to a changing work pattern, but at the same time it needs to be capable of detecting anomalies in this pattern.

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

It is really difficult to measure productivity in lines of code or in the number of commits made by a person, as they do not necessarily display the amount of work, which has been put into this code. As a result, I decided, that a day counts as a work day as long as at least a single commit is created during the day. Before performing the actual analysis, the data is preprocessed into an easier to handle format. For each week of the last year, a vector representing the weekdays is created. Afterward, each day a commit has been made on is marked as a working day. The preprocessed data is thereby equivalent to which 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:
            # Check whether there is a prototype in the next few weeks.
            prototype = find_prototype(next_six_weeks)

            # Check if this specific week is an 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: The algorithm implemented for detecting miss-out written in python.

The analysis of the data is a chronological scan of all work weeks for a specific user. The algorithm inspects every week work pattern of a given interval, which has been set to a year for this analysis. In the beginning, the algorithm tries to find a new *prototype*. A prototype is a representative week work pattern which resembles the average workday pattern of the next weeks. This is performed in the function `find_prototype` in Listing 3.

```python
def find_prototype(self, weeks, threshold):
    """Look at the first few weeks to find a new prototype."""
    # Create an entry for each pattern and count the occurrences of this entry
    occurrence_counter = {}
    for _, week in weeks.iterrows():
        pattern = week.pattern
        if pattern not in occurrence_counter:
            occurrence_counter[pattern] = {
                'prototype': week,
                'count': 1,
            }
        else:
            occurrence_counter[pattern]['count'] += 1

    # Get the prototype for the pattern with the most occurrences
    #
    # If there is no week which solely has the most occurences, return None.
    # In this case we can't predict a proper prototype.
    max_count = 0
    invalid = False
    prototype = None
    for _, item in occurrence_counter.items():
        if item['count'] > threshold and item['count'] > max_count:
            prototype = item['prototype']
            max_count = item['count']
            invalid = False
        elif item['count'] == max_count:
            invalid = True

    if invalid:
        return None

    return prototype
```

Listing 4: The code used for detecting a prototype in the work pattern on a weekly basis written in python.

This function, which can be seen in Listing 4, performs a simple iteration over a fixed amount of future weeks to find a workday pattern occurring 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 then firstly checked if this week is an anomaly in regards to the current prototype. Anomalies are simply detected by comparing the number of working days of the prototype and the considered week. The difference in the working patterns is not suitable for this analysis, as it produces too many false positives for employees with flexible work time.

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

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 workday, will then be marked as such.

4.2 Sleep Rhythm and Working Hours

The next attack aims to extract information about the working hour behavior of a target. This should be achieved by displaying the pattern of the target in form of a weighted scatter plot and by comparing those patterns between several targets. Additionally, a detection of anomalies, such as automated programs that contribute to a project on a regular basis, will be conducted.

A clustering will be performed to find common patterns, anomalies 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 number of considered contributors from 175.000 to about 10.300.

The data used 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 commits per hour per weekday over the last year. This results in a simple vector with length 168 which corresponds to seven days with 24 hours each. I will refer to this representation hereafter as a *punch card*.

```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 5: The preprocessing code used to simplify all commits into a usable format of a vector with length 168.

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 Listing ??. The resulting punch card vector is then stored in the database for faster and easier analysis in the next steps.

Figure 4.1 shows the punch card of the author. The y-axis represents the days of a week, the x-axis defines the hour of a day. The weight on the dots of the scatter plot corresponds to the number of commits contributed at this specific hour in respect to the number of all commits in the considered time span.

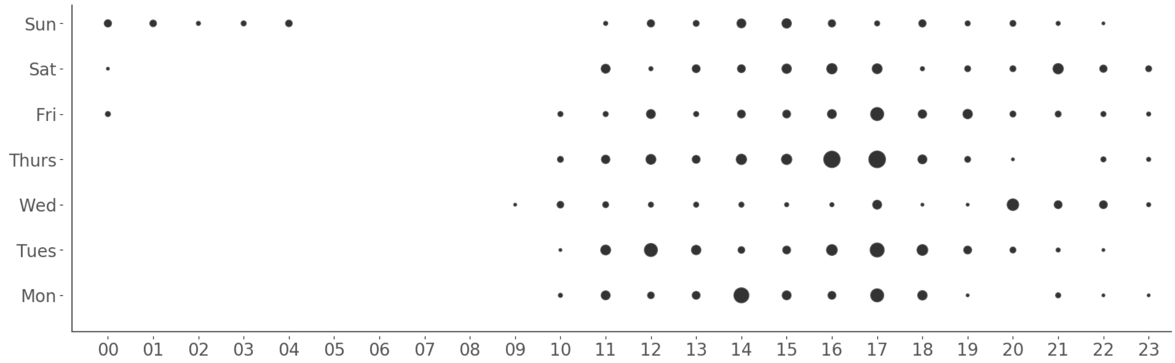


Figure 4.1: Punch card of the author.

Punch card Clustering

To find common work patterns, several cluster algorithms have been applied to 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 punch card patterns in the data, a clustering algorithm that can operate on a high-dimensional dataset with an unknown amount of clusters is required. Scikit provides three different clustering algorithms, which can handle an unknown amount of clusters.

Mean Shift

Mean shift is a clustering method which performs an operation similar to a gradient descent, during which all adjacent data points are shifted towards their cluster center [3]. The goal of this algorithm is to find a representative centroid for each cluster and to assign each data point to a cluster. Unfortunately, this methodology proves to be too aggressive for the current data.

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 supercluster, which contains more than 89% of all data points. Such a supercluster can be seen in Figure 4.2. 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 of such an extreme outlier can be seen in Figure 4.3.

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

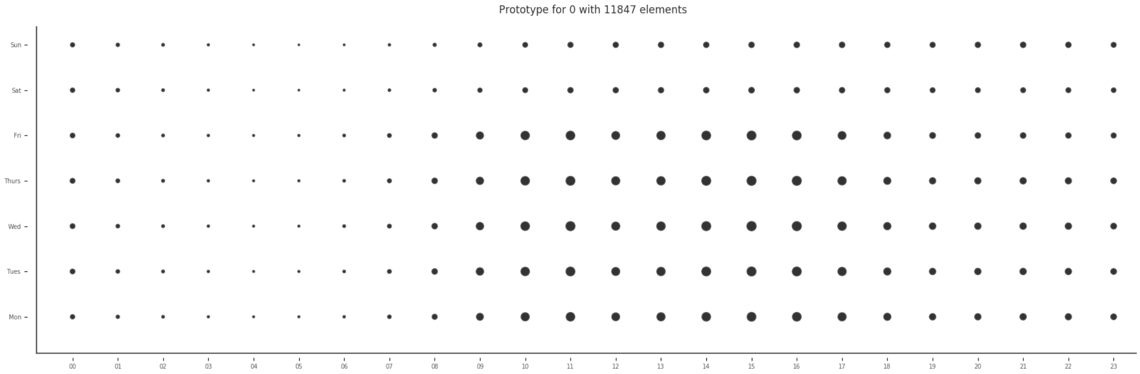


Figure 4.2: Punch card of the super cluster centroid found by mean-shift clustering.

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

DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm operates by creating clusters of transitively connectable data points within a specific maximal acceptable distance between adjacent points [5]. It is highly scalable and performant, even for large data sets, which made it my first choice. Unfortunately, it produces very similar results to the mean shift clustering algorithm 4.2 since it finds a supercluster very similar to Figure 4.2.

I assume that this algorithm suffers from similar problems as the mean shift approach, which are high density and equal distribution of data points without clear borders. Thereby the algorithm manages to reach the most part of all data points transitively from a single starting point. When supplied with smaller values for the maximal acceptable distance between data points it creates huge amounts of mini clusters, just as expected.

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 supercluster.

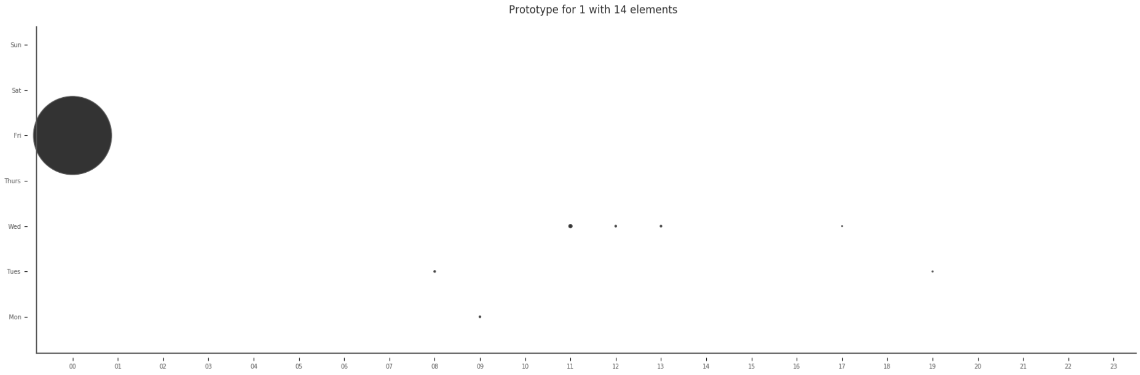


Figure 4.3: Punch card of an extreme outlier centroid found by mean-shift clustering.

Affinity Propagation

The Affinity Propagation algorithm considers similarities between all data points to find clusters [6]. This clustering algorithm features a promising approach since it utilizes a method similar to message passing for finding an *exemplar*. An exemplar 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 supercluster. 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 since it split relatively similar patterns into two or more different clusters.

Additionally, the memory requirements for this algorithm scale quadratically for non-sparse sets with the number of the data points [6, p. ii]. About 12 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 suitable for the validation of this thesis.

4.3 Geographic Location

This attack aims to extract information about the geographic location of a target. The algorithm tries to determine the exact timezone of the target and to exclude any countries or states that do not match the observed timestamps.

The data used for this analysis are the 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 different time zones and to narrow down the home location of a contributor. Several external data sources have been used to accomplish this:

IANA Database
The Internet Assigned Numbers Authority (IANA) provides a free-to-use database, with all timezones and the respective daylight saving times (DSTs) switches for each year. It also provides the exact UTC offset and offset switches for all time zones.

Natural Earth
The Natural Earth organization provides detailed free-to-use and up-to-date geological data with timezones, countries and even states on a resolution down to 1:10m. This data is used for visualization in combination with the Python *cartopy* library.

Pycountry
Since the country codes and names used by the IANA database and Natural Earth do not always match, another layer with more information about country names 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
OpenStreetMap is a collaborative project that provides a free map of the world. Since there are some time zones that cannot be assigned to a country with the help of the *pycountry* library, another solution for getting the relation between timezone and state or country was necessary. To match non-assignable timezone strings to their respective state or country, 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 a custom adapter that extracts 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 for each project setup, but it needs to be updated in case 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 the current_location set
        location = set(zones)
        intersection = location & current_location["set"]

        # Check if the possible timezones of this commit
        # matches any timezone of the current set.
        if len(intersection) > 0:
            # By reassigning the intersected set we gain additional precision
            # by considering possible specific DST changes
            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 be marked.
            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 current_location set
            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 6: Algorithm used to detect changes in the target's location by analyzing the UTC offsets of Git commit timestamps.

The algorithm in Listing 6 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 an intersection between the possible timezones of the last commits and the current commit. This is usually the case if the contributor did not travel to another time zone. 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 also something that can be used to improve the precision of the location.

For instance, Germany 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. In case only commits are considered that were created by a German contributor during a small time interval in the winter, it cannot be determined whether the contributor lives in South-Africa or in Western Europe. But if one considers the commits of a whole year, it can be concluded that the contributor has to be in a country that enforces DST and switches between the offsets +1 and +2.

If no intersection between the timezones can be found, it needs to be determined, whether the contributor actually committed or if the change happened 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 continue for longer than a day or which happen on the same day as a commit from the previous location are marked as insignificant and are ignored.

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

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

    # Try to find the current home location and
    # to 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 7: Methodology used to determine the main location of a target, based on the information gained from the function in Listing 6

To detect the home location of a contributor, the algorithm in Figure 7 is used. The parameter provided to this function are the results of function `get_travel_path` in Figure 6. The algorithm simply tries to determine the best possible set of timezones that subsists for the longest duration.

During this process, intersections of matching timezone sets are performed 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 5
Evaluation and Interpretation

In the previous chapter, the implementation of several possible attacks which could be performed on the gathered data has been shown. 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 the real-world ground truth. This information will be then be explained and audited in terms of precision and reliability.

5.1 Holiday and Sick Leave Detection

Figure 5.1 shows the analysis of an employee for their employer's repositories. The y-axis represents the additions and deletions of commits, the x-axis represents the week of the respective year. For better verification and evaluation of the results, a scatter plot with the additions and deletions of each commit has been added on top of individual miss-out graphs.

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 was allowed to use anonymous statistics of a company with a test group of five developing employees.

After scanning and analyzing the company's repositories, a survey has been conducted, for which each employee had to look at their visualized miss-out graph and check whether there are any wrong or missing detections.

The algorithm successfully managed to find all sick leave and holiday-related miss-outs in all cases. However, there were three false positive cases. Firstly a developer did work related research and did thereby not commit as regularly as usual, which was then detected as a miss-out. Secondly, a developer contributed to repositories, which are not owned by the employee. As a result, two weeks were marked as miss-out. Thirdly there were two developers, that contributed to a branch, but their work has not been merged into the master branch yet, hence the commits were not being scanned yet. This particular problem can be solved by not only scanning the master branch but also all other branches, however, this adds a lot of complexity to the continuous mining process and has thereby not been implemented yet. Anyway, this problem only seems to occur in the latest weeks.

A not intended side effect of detecting prototypes is that the algorithm not only detected anomalies, such as sick leave or holiday, but also found inconsistencies in the work routine. For instance between week 37 to 45 in Figure 5.1, a developer was forced to

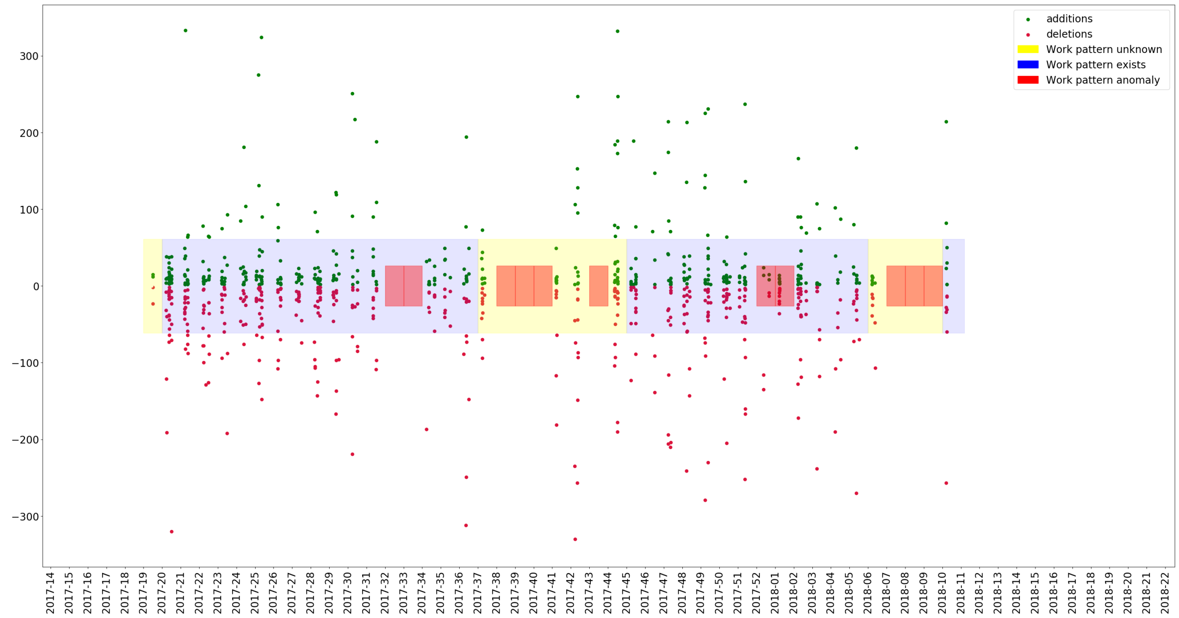


Figure 5.1: The work time analysis of an employee. Blue sections are time spans in which the target has a consistent reoccurring week work pattern. Yellow sections show irregularities in their week work pattern. Red blocks are miss-out anomalies detected by the algorithm, which indicate holiday or sick leave.

reduce their working hours due to legal questions and continuously shifted 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.

The conducted survey also included a question about the precision of these inconsistencies. In the case of three developers, the inconsistencies perfectly matched real-world occurrences in their work behavior. In the two remaining cases, there were a few inconsistencies the developers could not explain, even though there definitely were inconsistencies in their commit pattern. But those inconsistencies could be explained by the fact that those developers were working on flexible work time.

In Figure 5.2 the comparison between multiple employees can be seen. *Contributor0* and *Contributor2* are working on flexible work time, which reflects in the inconsistencies in the work patterns (yellow sections in Figure 5.2) of those contributors. The other two contributors have very consistent working hours patterns, as can be seen by the long-lasting blue sections.

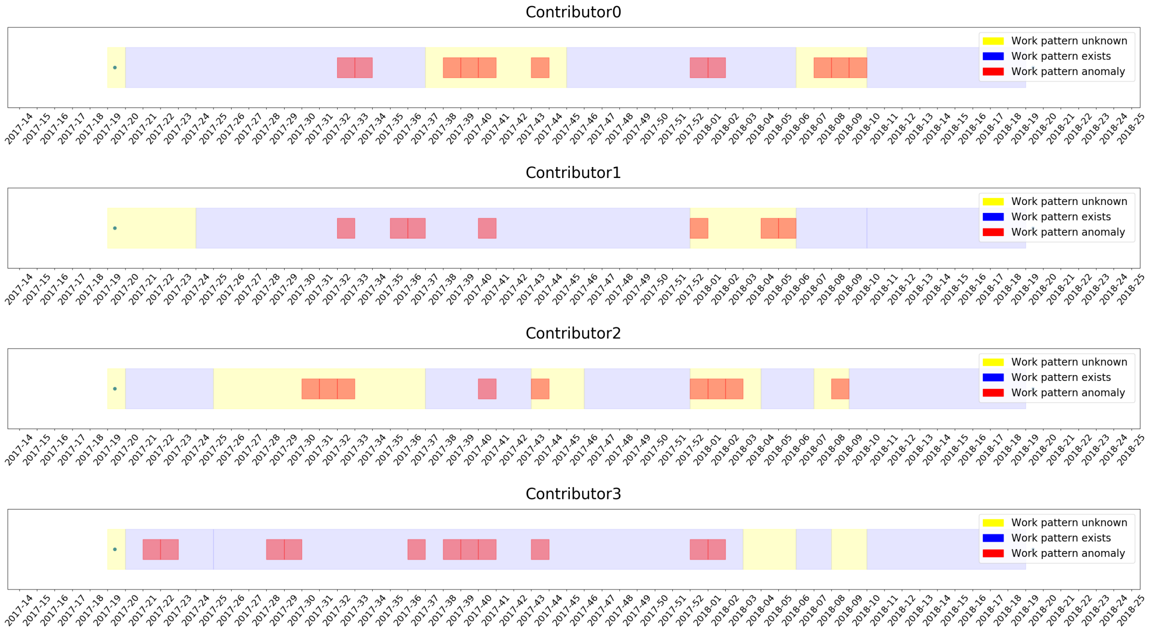


Figure 5.2: The miss-out analysis graphs of the four employees from the survey.

5.2 Sleep Rhythm and Working Hours

Sleep Rythm

To evaluate the significance of the punch card in terms of sleep rhythm analysis, a small survey in a closed community has been made. A group of ten people, who know each other well, has been selected for this purpose. Furthermore, a subset of four people has been chosen who were going to be evaluated. The data gathered for the test group only contained their leisure time contributions, as their work repositories were not open-source. All ten people then needed to assign the punch card of those four persons to a specific person. The specific task given to test group was to assign each punch card to a specific person based on their knowledge about their sleep rhythm and leisure time behavior.

To this end, three quite similar patterns, from which one contributor has a very regular sleep rhythm (Subfigure 2 in Figure 5.3) and one pattern of a practically inexistent sleep rhythm (Subfigure 1 in Figure 5.3) have been selected.

For the three similar graphs, no significant results could be assessed, as the assignments were more or less random. The contributor with the irregular sleep rhythm, on the other hand, got correctly assigned in all cases.

Sadly, as such a survey needs very specific targeting and a long lead time for data collection, it could only be executed on a small number of subjects. Anyhow the result of this survey indicates that there exists a correlation between the structure of the punch card and the sleep and leisure time commit behavior of a contributor, even if it can only be accurately assigned in extreme cases.

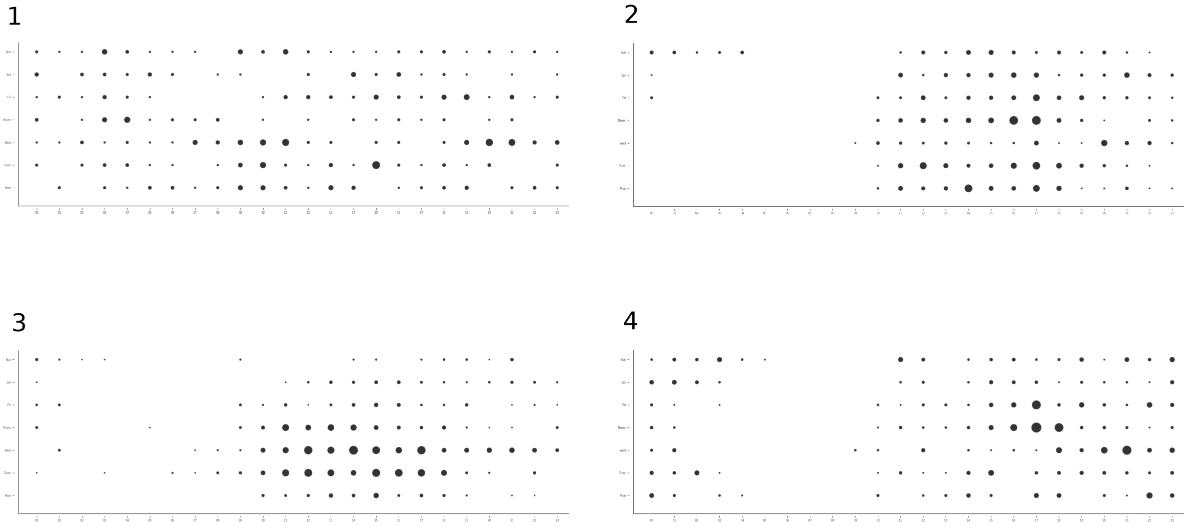


Figure 5.3: The punch cards created and used for the test group in the survey.

Employee or Open-Source Contributor

To determine whether a punch card could be used to distinguish between an employee or an open-source volunteer, the results of the clustering described in Section 4.2 have been utilized. For this approach, two assumptions have been made. A usual employee works between Monday and Friday during the day and only as an exception at the weekend, an example cluster can be seen in Figure 5.4. An open-source volunteer works outside of the usual work shifts, which means early and late during weekdays and at the weekend, an example cluster can be seen in Figure 5.6.

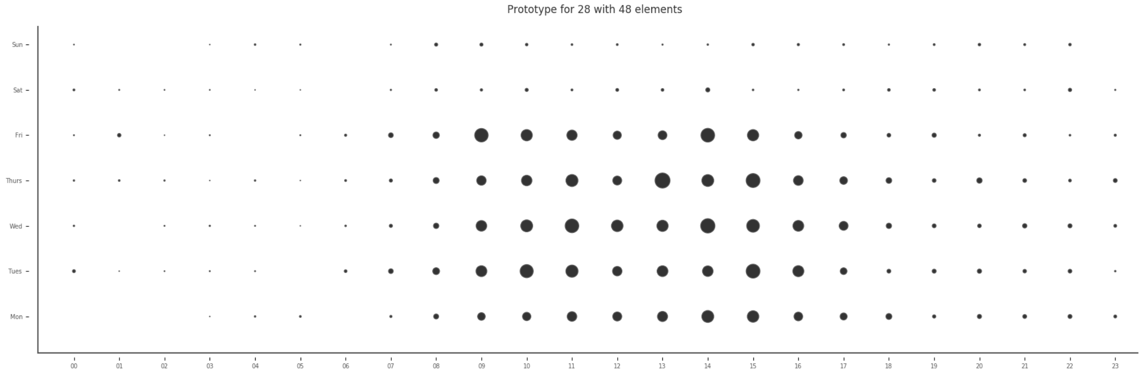
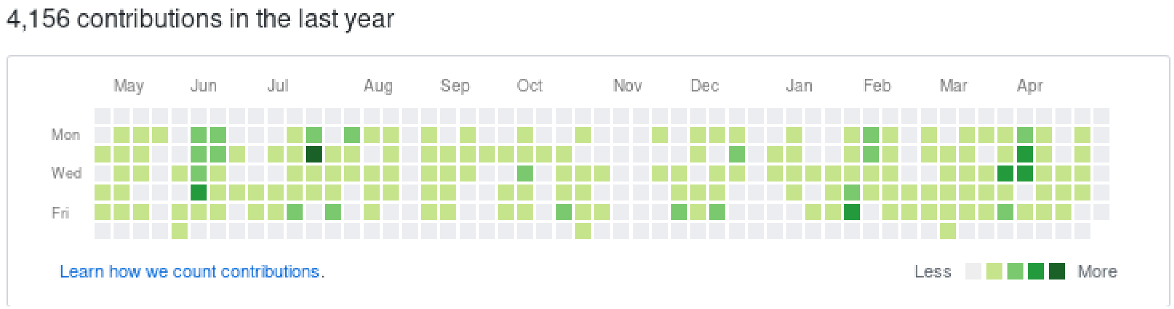


Figure 5.4: Punch card of an example from an affinity propagation cluster of contributors with normal work shifts. A clear tendency to regular office working hours can be seen (Monday to Friday between 7:00 and 18:00 o'clock)

For each assumption, two representative clusters have been chosen and ten random persons have been selected from each cluster. The manual verification is conducted by checking if the contributor mainly contributes to repositories which belong to the registered employee. If no employee is registered, but other sources such as a homepage are provided, the information of these sources is checked for possible employee details as well. In case no employee exists, it is examined whether the contributor pushes to their own and to open-source projects or rather to the repositories of a specific company.


4,156 contributions in the last year

Learn how we count contributions.

Figure 5.5: Github contribution overview of a Google developer with the nickname alxhub.

For this purpose, the Github contribution overview on the contributors' profile page has been used. An example of such an overview can be seen in Figure 5.5. 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 month.

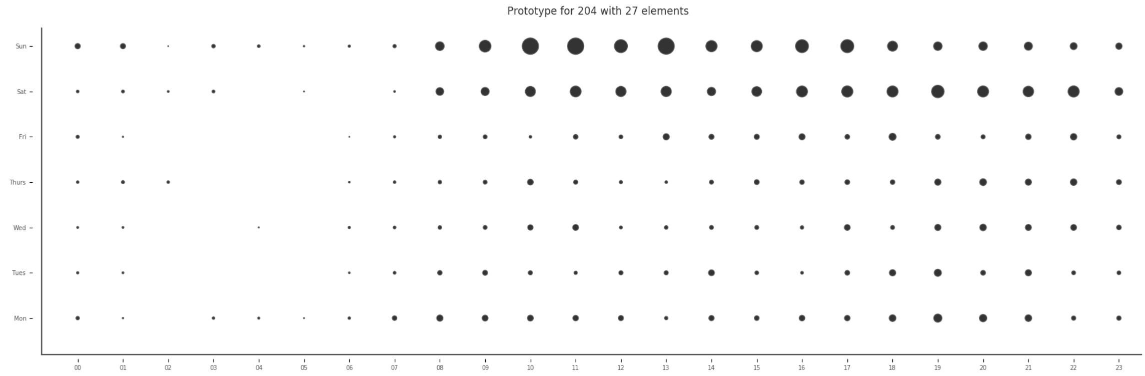


Figure 5.6: Punch card of an example from an affinity propagation cluster of leisure time contributors. A clear tendency to contributions out of regular working hours and at the weekend.

The representatives for the usual five-day week commit behavior were surprisingly accurate. 19 out of 20 considered contributors were mainly working on projects of their companies, with occasional commits to other open-source projects. For the remaining contributor, it could not be determined if they work for a company.

The representatives for the leisure time commit behavior are mostly correct as well. 15 out of 20 considered contributors were irregularly contributing to either work unrelated open-source projects or to their own projects. See Appendix A.1 for a table with the results of the leisure time cluster evaluation. The remaining five contributors were either contributors working and committing to their employee's projects, but also to their own and open-source projects or employees with an untypical commit behavior.

This analysis shows quite well, that there is a correlation between the assumed patterns and the Github commit behavior or the usage of their Github accounts. Unfortunately, the evaluation process for these results is very time consuming and thereby only a relatively small sample (n=40) has been chosen. As it is not trivial to link the employee of a contributor to all their funded projects, all verification needed to be conducted manually.

Table 5.1: Employee cluster evaluation.

Nickname	Employer	Main projects	Work related	Over 90% work related
brettfo	Microsoft	visualfsharp	yes	no
aputinski	Salesforce	Salesforce	yes	yes
alxhub	Google	Angular	yes	yes
MatrixFrog	Google	Google Projects	yes	yes
ryanemerson	Red Hat	Infinispan	yes	no
garagatyi	Red Hat	Eclipse and related projects	yes	yes
eternoendless	PrestaShop	PrestaShop	yes	yes
initvector	vanilla	vanilla	yes	yes
kyhavlov	HashiCorp	HashiCorp	yes	no
XenoPhex	CloudFoundry	CloudFoundry	yes	yes
gjoranv	unknown	vespa-engine/vespa	probably	yes
doolse	Equella	Equella	yes	yes
leplatrem	Mozilla	Mozilla/Kinto	yes	yes
StrongMonkey	Rancher Labs	Rancher Labs	yes	no
lukaseder	jOOQ	jOOQ	yes	yes
DaazKu	vanilla	vanilla	yes	yes
brettcannon	Microsoft	Microsoft/Python	yes	yes
isidorn	Microsoft	Microsoft Projects	yes	yes
jackhorton	Microsoft	Microsoft Projects	yes	yes
glasserc	Mozilla	Mozilla/Kinto	yes	yes
nickwei84	CloudFoundry	CloudFoundry	yes	yes

Bot Detection

Another possible attack that opened up during the creation of the clusters was the detection of automatically committing programs, so-called *bots*. Several clusters showed a very consistent commit behavior around a specific hour, such a punch card can be seen in Figure 5.7.

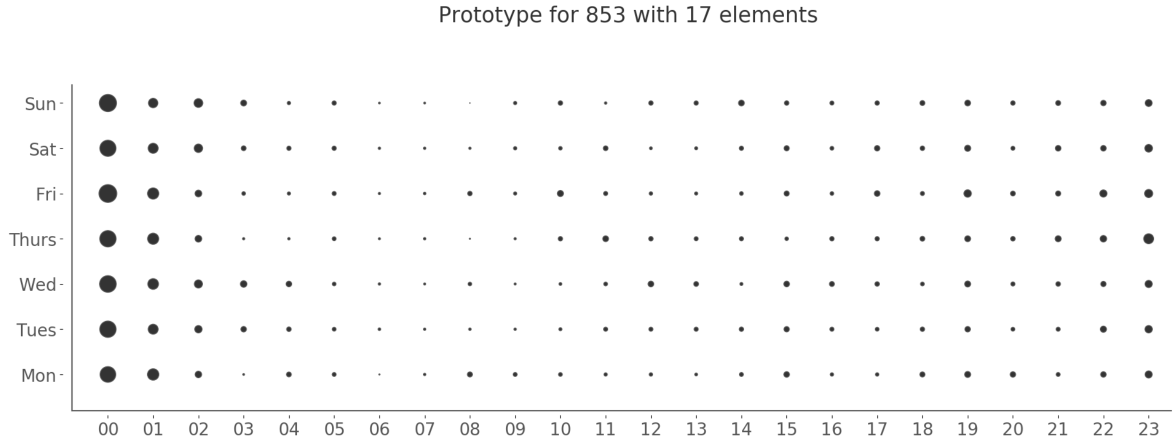


Figure 5.7: A punch card with an extraordinary commit pattern around midnight. This might indicate a regularly automatically committing program, a so-called *bot*

In the following, I implemented an algorithm which simply detected centroids with an extremely equally distributed pattern or patterns with a spike at a specific hour. After a manual revision of the clusters detected by this methodology, it became apparent that only a very small subset of those clusters actually contained bots. Even if the cluster contained bots there usually were only one or two of 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 100 possible candidates led to not a single bot. After reviewing these results, I decided that there is currently no viable approach to this problem.

Fingerprinting

Another possible attack was to fingerprint a contributor and create a unique identifier by analyzing their commit behavior.

This attack soon proved to be unfeasible, as the pattern of a contributor can significantly differ from month to month, as can be seen in Figure 5.8 and Figure 5.9.

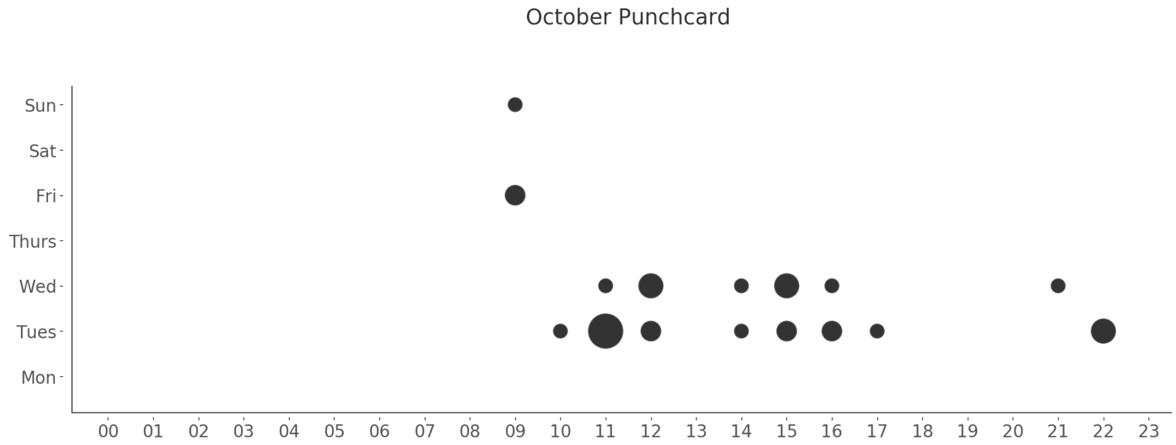


Figure 5.8: The author's punch card from October 2017.

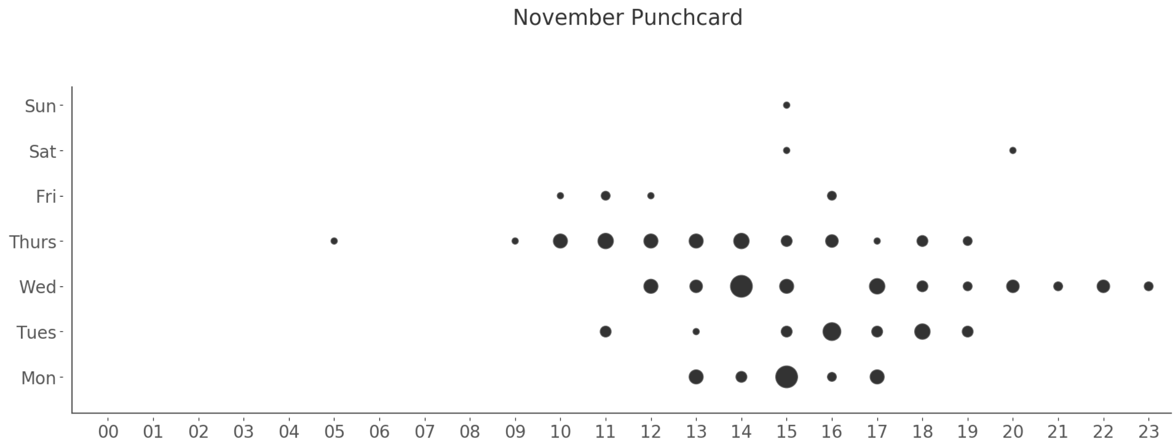


Figure 5.9: The author's punch card from November 2017.

If the interval of a year is considered and the compared interval is shifted by a single month, the occurring changes are not that drastic, but still too different to see a consistent pattern over a longer time. The commit behavior of people seems to be too inconsistent to create a unique fingerprint.

5.3 Geographic Location

First of all, it needs to be clarified that parts of this attack only works under specific circumstances. Git commit timestamps are created by taking the current local time of the underlying Operating System (OS). If one wants to show the travel path of a target, the target's OS needs to automatically adjust the UTC offset accordingly to the current geographic location of the device.

This feature is available for newer versions of popular OSs, such as *Windows* [1] and *Mac Os*, but they are not enabled by default. It is also available for Linux, for instance, with the *tzupdate* package [2], but it needs to be manually installed and activated.

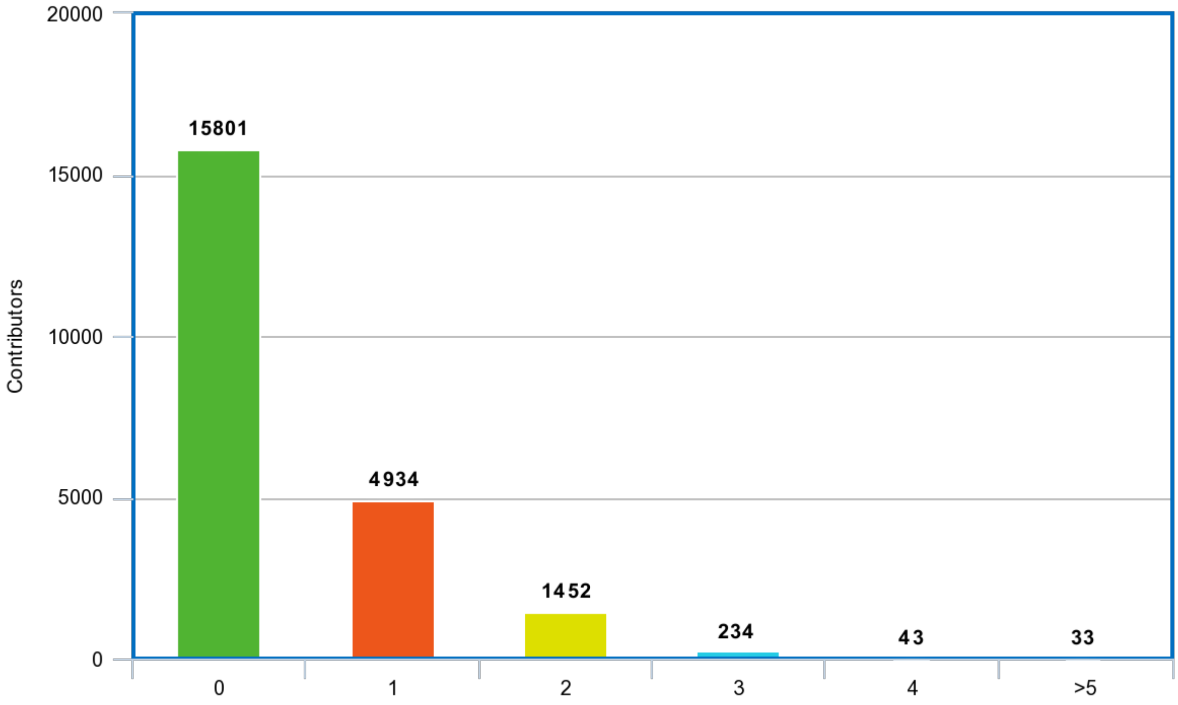


Figure 5.10: Distribution of users according to the amount of timezone switches detected by the algorithm. The major part of all contributors does not have any detectable timezone changes.

Figure 5.10 shows the number of contributors in relation to the number of detected timezone switches. On about 70% of considered contributors, only a single timezone has been detected, looking at the last year. These 70% do either not commit when they travel, their OSs do not synchronize the timezone accordingly to their location or they simply did not travel in the last two years.

In Figure 5.12 the visualized home location analysis of the author can be seen. Regions marked in dark green are regions in which the contributor is likely to live. The light green region represents the timezone of the home location. As you can see in Figure 5.12 the country of French Guiana is also marked as a possible home location. This problem occurs due to the several conversions between country names and codes, which were necessary as stated in Section 4.3. This misassignment only happens during

[1] Ivan Jenic, 'Your Time Zone Can Now Switch Automatically in Windows 10', windowsreport.com, https://windowsreport.com/time-zone-automatic-switch-windows-10 (accessed, 24.04.2018)

[2] 'Set the system timezone based on IP geolocation', github.com, https://github.com/cdown/tzupdate (accessed, 24.04.2018)

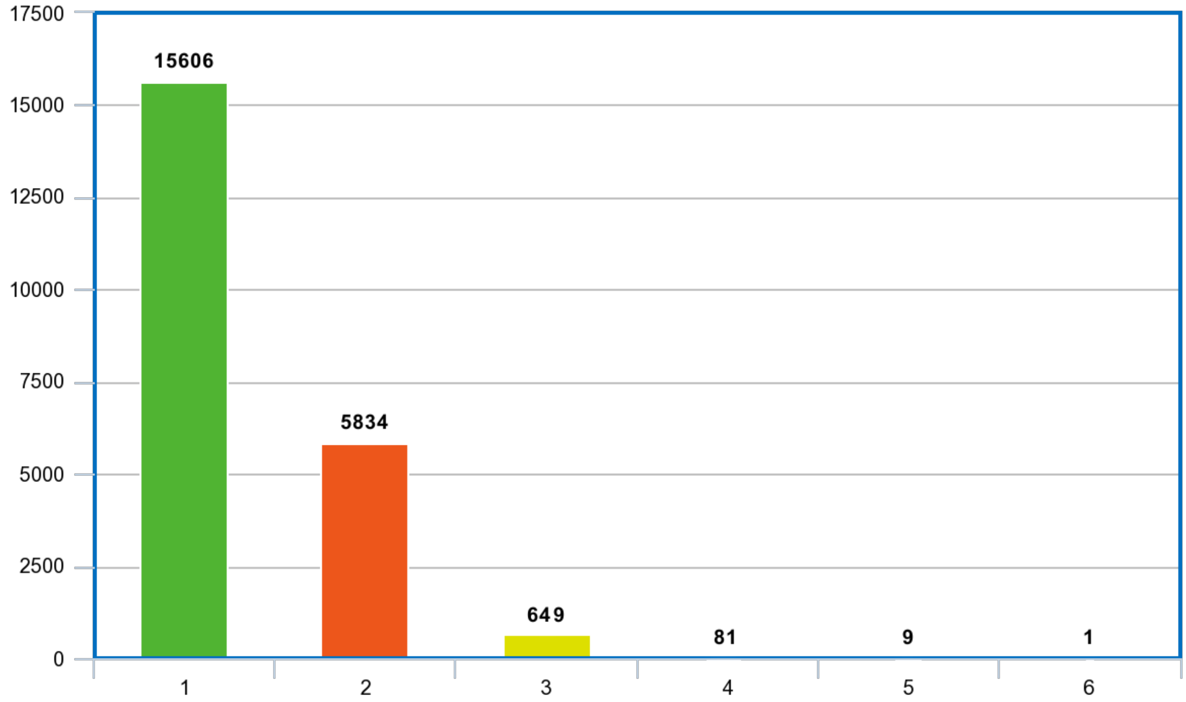


Figure 5.11: Distribution of users according to the amount of the different timezones detected by the algorithm.

the visualization process of the results and thereby does not affect the results of the analysis.

To evaluate the overall precision of the geographic location results, the correctness of the determined home locations are checked. Github allows users to specify a string for their current home location, which is also collected during the data aggregation process. Unfortunately, there are no conventions on how this string has to look like. Initially, I tried to pass these strings to the OpenStreetMap API, but this resulted in too many wrongly assigned locations. The data provided by the users was obviously too arbitrary and full of mistakes for OpenStreetMap to handle.

As a result, I decided to manually choose a subset of locations by looking for distinct identifiers in the location strings. For instance, every home location of a contributor, that contained *Germany* or *Deutschland* in their location string, should be in the timezone `Europe/Berlin`, which switches between Central European Time (CET) and Central European Summer Time (CEST). I created 14 such rules and was thus able to validate the home location of about 4200 contributors. The assignment of the contributors home location was correct in about 82% of the considered contributors.

It needs to be noted, that the accuracy of this result is quite certainly deteriorated by location strings which contain ambiguous information. On manual review of the location strings there were strings such as `I love NYC` which belongs to a developer living in

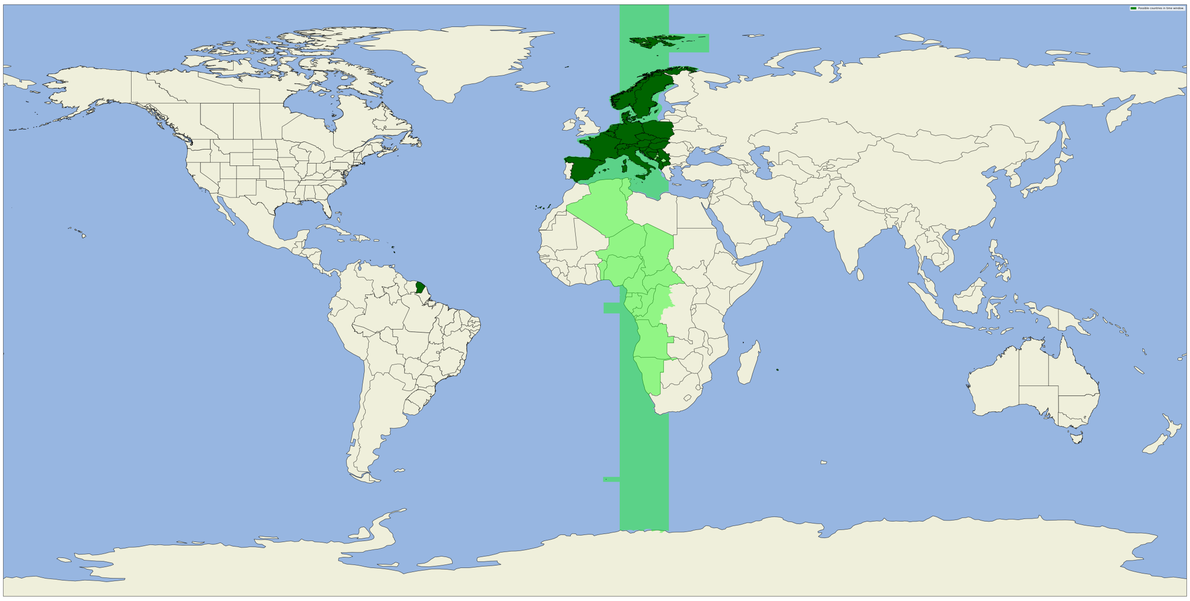


Figure 5.12: The visualized home location analysis of the author. The light green indicates the timezone the target is probably in. The dark green color shows the remaining countries after considering DST switches.

Germany. The result might also be deteriorated by contributors who moved to another country in the last year, as we cannot detect those for sure.

It also needs to be noted, that the IANA database does not always have exact mappings for countries to their timezone. For many countries or states, the current or an old capital city is used. Some countries do not have an own timezone, such as Japan, which is included in `Pacific/Palau`. The IANA database is the currently best viable approach, but for better results and a more fine-grained resolution, a specific mapping between countries, states and time zones would be necessary.

Nevertheless, an accuracy of 76% clearly shows, that it is possible to narrow down the location of a contributor to a timezone and even to a subset of countries, see Table 5.2 for reference, by simply looking at their git commit timestamps.

Query string	Expected timezone string	Considered	Correct	Timezone strings after DST	Before DST
San Francisco	US/Pacific	772	639	12.07	18
NYC, NY, New York	America/New_York	485	366	23.60	52
India	Asia/Colombo	148	127	4	4
UK, United Kingdom	Europe/London	667	510	13.14	22
France	Europe/Paris	490	421	31.59	40
New Zealand	Pacific/Auckland	59	52	4.76	9
Germany, Deutschland	Europe/Berlin	976	846	31.48	38
Poland	Europe/Warsaw	180	154	31.56	40
Italy	Europe/Rome	130	112	31.39	41
Tokyo	Pacific/Palau	61	48	8.51	12
Spain	Europe/Madrid	129	116	32.83	40
Los Angeles	America/Los_Angeles	86	76	12.24	18
Adelaide	Australia/Adelaide	2	2	4.00	4
Mexico	Mexico/General	13	7	11.62	39

Table 5.2: Results of the home location evaluation. The first column shows the comma separated strings by which contributors are selected depending on their location. The second column is the timezone that is expected to be in the remaining home location timezone string set.

Chapter 6
Conclusion and Outlook

The study set out to determine how feasible and precise data mining attacks on simple Git metadata could be. All three performed attacks, lead to promising results and showed potential for malicious usage.

The miss-out analysis showed that it is possible to automatically detect holiday and sick-leave anomalies. Additionally, it is capable of detecting other anomalies in the developer's work pattern.

Analyzing the Git commit timestamps to narrow down the geographic location of a user-led to a significant reduction of possible locations on the globe. With a proper test group, it is also likely to prove, that the other detected timezones represent the travel history of the target.

The analysis of punch cards showed, that it is possible to detect developers working at regular five day office hours and to distinguish between working employees and leisure time developers.

However, it must be noted that in all attacks only a small amount of the available data was used. Simply using the Git commit timestamps allowed us to perform analyses such as narrowing down the location of a contributor. The possible applications for the remaining data, like actual changes in code, references of contributors between repositories or commit messages, are extensive.

If one would add additional data from Github, such as followers, stars or information from their issues system, the results could become even more accurate. Developers around the world provide metadata about themselves on a daily basis, probably without knowing how much they are actually exposing. To prevent the unauthorized usage and abuse of this data, we need to create countermeasures or prevent exposing this data in the first place.

Luckily the European Union (EU) set an example by enforcing the General Data Protection Regulation (GDPR), which is a regulation that strictly rules the handling of any user data. But there are still many countries left in the world, that do not have such strict rules and that might need ways to protect their privacy from being invaded.

While *Gitalizer* is a foundation for data aggregation and the conduct of rather simple analyses, there is a necessity for more detailed research with better sources for ground truth. Additionally, more statistics about the mining process would be convenient for evaluating the research results, such as the ratio between starred and contributed repositories. *Gitalizer* is a quite complex program, but it is well documented and should allow other people to easily jump into using it.

Many of the attacks mentioned in Section 2.2 are not implemented, as they did not fit in the scope of this thesis. Implementing those could be the topic of another bachelor thesis or for a subsequent master thesis.

Furthermore, it would be interesting to explore the possibilities of countermeasures such as obfuscating Git commit timestamps.

Appendices

Appendix A
Leisure time cluster evaluation

Table A.1: Leisure time developer cluster evaluation.

Nickname	Employer	Main projects exists	Work related	Over 90% work related	Commits rarely
craigbarratt	unknown	backuppc	unknown	unknown	yes
hackebrot	Mozilla	Own repos, open-source	no	no	no
itamarnet	unknown	Open-source	no	no	yes
JamesNK	unknown	Own repos	no	no	no
kpreid	unknown	Own repos	no	no	yes
vrasneur	unknown	Own repos	no	no	yes
chrissimpkins	SourceFoundry	Own repos, SourceFoundry	yes	no	no
klauscfhq	Cookfood HQ	Own repos	no	no	no
knowthelist	unknown	Own repos	no	no	yes
larskanis	Comcard	Open-source	no	no	no
NickeManarin	unknown	Own repos	no	no	yes
obilodeau	GoSecure	GoSecure	yes	no	no
piotrmurach	unknown	Own repos	no	no	no
hauke	unknown	Open-source	unknown	unknown	yes
rspeele	unknown	Own repos	no	no	yes
DylanC	PhotoFlare	PhotoFlare	yes	yes	yes
Gitmaninc	unknown	SecWiki	no	no	yes
lgreski	unknown	Own repos	no	no	yes
vitaut	Facebook	Open-source	no	no	no
wikimatze	MyHammer AG	Own repos, open-source	no	no	yes
schaal	unknown	Own repos	no	no	yes

List of Figures

List of Listings

List of Tables

Bibliography

[1] Gábor Antal, Ádám Zoltán Végh, and Vilmos Bilicki. “A methodology for measuring software development productivity using Eclipse IDE”. In: (Jan. 2015), pp. 255–262.

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

[3] Yizong Cheng. “Mean shift, mode seeking, and clustering”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17.8 (Aug. 1995), pp. 790–799. ISSN: 0162-8828. DOI: 10.1109/34.400568.

[4] Maëlick Claes et al. “Do Programmers Work at Night or During the Weekend?” In: *CoRR* abs/1802.05084 (2018). arXiv: 1802.05084. URL: http://arxiv.org/abs/1802.05084.

[5] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise”. In: AAAI Press, 1996, pp. 226–231.

[6] Brendan J. Frey and Delbert Dueck. “Clustering by Passing Messages Between Data Points”. In: *Science* 315.5814 (2007), pp. 972–976. ISSN: 0036-8075. DOI: 10.1126/science.1136800. eprint: http://science.sciencemag.org/content/315/5814/972.full.pdf. URL: http://science.sciencemag.org/content/315/5814/972.

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

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

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

[10] Georgios Gousios et al. “Lean GHTorrent: GitHub Data on Demand”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 384–387. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597126. URL: http://doi.acm.org/10.1145/2597073.2597126.

[11] AdriÁn HernÁndez-lÓpez, Ricardo Colomo-Palacios, and Angel Garcia Crespo. “Software engineering job productivity-a systematic review”. In: 23 (July 2013).

[12] Heinz-Peter Höller and Peter Wedde. *Die Vermessung Der Belegschaft*. 2018. URL: https://www.boeckler.de/112779_112796.htm (visited on Apr. 28, 2018).

[13] Mitchell Joblin et al. “From Developer Networks to Verified Communities: A Fine-grained Approach”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 563–573. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818824.

[14] T. Capers Jones. “Measuring Programming Quality and Productivity”. In: *IBM Systems Journal* 17 (1978), pp. 39–63.

[15] Eirini Kalliamvakou et al. “The Promises and Perils of Mining GitHub”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 92–101. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597074. URL: http://doi.acm.org/10.1145/2597073.2597074.

[16] 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).

[17] P. Louridas. "Version Control". In: *IEEE Software* 23 (Jan. 2006), pp. 104–107. ISSN: 0740-7459. DOI: 10.1109/MS.2006.32. URL: http://doi.ieeecomputersociety.org/10.1109/MS.2006.32.

[18] Andrew McAfee and Erik Brynjolfsson. "Big Data: The Management Revolution". In: (2012). URL: https://hbr.org/2012/10/big-data-the-management-revolution (visited on May 5, 2018).

[19] 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).

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

[21] Gianluca Roveda and Dott Tullio Facchinetti. "Mining Git based Software Repositories". In: ().

[22] Rachel Emma Silverman. "Bosses Tap Outside Firms to Predict Which Workers Might Get Sick". In: (2016). URL: https://www.wsj.com/articles/bosses-harness-big-data-to-predict-which-workers-might-get-sick-1455664940?mod=e2tw (visited on May 5, 2018).

[23] Yu Wu et al. "Exploring the Ecosystem of Software Developers on GitHub and Other Platforms". In: *Proceedings of the Companion Publication of the 17th ACM Conference on Computer Supported Cooperative Work &#38; Social Computing.* CSCW Companion '14. Baltimore, Maryland, USA: ACM, 2014, pp. 265–268. ISBN: 978-1-4503-2541-7. DOI: 10.1145/2556420.2556483. URL: http://doi.acm.org/10.1145/2556420.2556483.

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