Data Minimisation Potential for Timestamps in Git: An Empirical Analysis of User Configurations

Christian Burkert, Johanna Ansohn McDougall, and Hannes Federrath

University of Hamburg, Hamburg, Germany
{christian.burkert, johanna.ansohn.mcdougall, hannes.federrath}@uni-hamburg.de

Abstract With the increasing digitisation, more and more of our activities leave digital traces. This is especially true for our work life. Data protection regulations demand the consideration of employees' right to privacy and that the recorded data is necessary and proportionate for the intended purpose. Prior work indicates that standard software commonly used in workplace environments records user activities in excessive detail. A major part of this are timestamps, whose temporal contextualisation facilitates monitoring. Applying data minimisation on timestamps is however dependent on an understanding of their necessity. We provide large-scale real-world evidence of user demand for timestamp precision. We analysed over 20 000 Git configuration files published on GitHub with regard to date-related customisation in output and filtering, and found that a large proportion of users choose customisations with lower or adaptive precision: almost 90% of chosen output formats for subcommand aliases use reduced or adaptive precision and about 75% of date filters use day precision or less. We believe that this is evidence for the viability of timestamp minimisation. We evaluate possible privacy gains and functionality losses and present a tool to reduce Git dates.

Keywords: Privacy · Data Minimisation · Timestamps · Timestamp Precision

1 Introduction

In increasingly digital work environments, employees' digital and non-digital work steps leave traces of their activities on computer systems. Employers, supervisors and analysts see such data as a resource and opportunity to gain intelligence for business optimisation. Without strong consideration of employees' right to privacy, such legitimate interests might easily lead to excessive and invasive monitoring, even without the employees noticing. Recent reports about mass lay-offs at the game design company Xsolla show that automatic monitoring of employee performance based on software activity logs is already done in practice [7]. Such invasions of employee privacy are however restricted by data protection regulations like GDPR, which requires that the processing of personal data is necessary and proportionate for and limited to the intended purpose.

Software design can contribute to the protection of employee privacy by reducing the amount and detail of data that is stored about user interaction to such a necessary minimum. Prior work, however, indicates that software commonly used in workplaces records especially timestamps in excessive detail [2]. It shows that timestamps are not only often unused, but might otherwise also be of unnecessarily high precision. As timestamps allow an easy temporal profiling of employee activities, a reduction in precision could directly reduce the risk of profiling-related discrimination. For instance, a reduction can prevent the inference of intervals between successive work steps and thus mitigate the monitoring of speed and performance. Identifying the necessary level of precision is, of course, dependent on the domain and respective user demand. Nonetheless, similar precision demands can be expected for interactions of similar kind and frequency. In that sense, insights into which levels of timestamp precision are selected by workers if they have the choice, can inform the selection of more appropriate default precisions in software design. We argue that when users configure their tools to precisions that are lower than the default, this implies that the lower precision is still sufficient for them to fulfil their tasks. Therefore, user customisation is an indicator for users' demand for timestamp precision. With an informed understanding of users' demands, developers can then built software with demand-proportionate timestamping and privacy-friendly defaults.

To the best of our knowledge, we provide the first large-scale real-world analysis of user demand for timestamp precision. Our analysis is based on configuration files for the popular revision control system Git, that users have made publicly available on GitHub. Git is a standard tool for software development workers and its recording of worker activity in the form of commits, contributes significantly to the overall traces that developers leave during a workday. Commit dates have been used to infer privacy sensitive information like temporal work patterns [3] and coding times [17]. The analysed configurations can contain various preferences that customise the way Git presents dates, including their precision. For instance, using the date formats *iso* or *short* would indicate a high (second) or low (day) precision demand respectively. We also examined the precision of filters (e.g., *8 hours ago*) used to limit the range of outputs. In total, we analysed over 20 000 configuration files. We make the following contributions:

– We compile and provide a comprehensive large-scale dataset of date-related usage features extracted from publicly available Git configs.
– We provide empirical evidence for the demand of date precision by users, as determined by the precision of user selected date formats.
– We discuss and evaluate privacy gain and functionality loss.
– We present a utility that allows users to redact their Git timestamps.

The remainder is structured as follows: Section 2 presents related work. Section 3 provides a necessary background on Git and its date handling. We describe the acquisition and analysis of our Git config dataset in Sect. 4 and Sect. 5, and discuss findings, issues and applications in Sect. 6. Section 7 concludes the paper.

2 Related Work

To the best of our knowledge, we are the first to gather empirical evidence for the potential of data minimisation in timestamps. In prior work, we inspected application source code in order to assess the programmatic use of timestamps in application data models [2]. The case study of the Mattermost application found that most user-related timestamps have *no* programmatic use and only a small fraction are displayed on the user interface. We addressed the potential to apply precision reduction to user-facing timestamps. However, the code analysis could not provide any indication of acceptable levels of reduction. More work has been done on the exploitation of Git timestamps and the potential privacy risks. Claes et al. [3] use commit dates to analyse temporal characteristics of contributors to software projects. Eyolfson et al. [6] use dates to find temporal factors for low-quality contributions. Wright and Ziegler [17] train probabilistic models on individual developers' committing habits in an effort to remove noise from coding time estimations. Traullé and Dalle [16] analyse the evolution of developers' work rhythms based on commit dates. Following a more general approach, Mavriki and Karyda [13] analyse privacy threats arising from the evaluation of big data and their impact on individuals, groups and society. Drakonakis et al. [4] evaluate privacy risks of meta data with a focus on online activity in social media and, e. g., try to infer location information from publicly available data. No work seems to exist that proposes or evaluates temporal performance metrics. Slagell et al. [15] proposes time unit annihilation, i. e., precision reduction, to make timestamps less distinct and sensitive. Looking at developer behaviour, Senarath and Arachchilage [14] found that while developers typically do not program in a way that fulfils data minimisation, being made aware of its necessity made them apply the principle across the whole data processing chain. With this paper, we also strive to raise the awareness for minimisation of temporal data.

3 Theoretical Background: Git and Date Handling

This section provides a background on Git's time and date configuration options. Experts in Git and its date and pretty formatting may jump directly to Sect. 4.

Git's command line interface exposes individual actions like creating a commit or listing the history via subcommands like `git commit` or `git log`. Their behaviour can be configured via command line arguments and—to some extent—via settings made in configuration files. Frequently used combinations of subcommands and arguments can be set as shortcuts via so-called *aliases*, similar to shell aliases. For example, the shortcut `git ly` set in Listing 1.1 configures the `log` subcommand to list all commits since yesterday.

In the following, we describe the role and creation of dates in Git and then explain the date-related features that will be empirically analysed later.

Listing 1.1: Examplary Git config

```
[alias]
  ly = log --date=human --since=yesterday
[blame]
  date = short
[pretty]
  my = %h %an (%ai)
```

Table 1: Git's built-in date formats and their precision.

Name	Suffix	Precision	Example(s)
default	-	second	Wed Sep 22 14:57:31 2021 +0200
human	-	day to second	Sep 21 2021 / 7 seconds ago
iso	i/I	second	2021-09-22 14:57:31 +0200
raw	-	second	1632315451 +0200
relative	r	year to second	7 years ago / 7 seconds ago
rfc	D	second	Wed, 22 Sep 2021 14:57:31 +0200
short	s	day	2021-10-04
unix	t	second	1632315451

3.1 Dates in Git

Git associates two types of dates with each commit: the author date and the committer date. Both are usually automatically set to the current date and time, except in case of operations that modify existing commits (e.g., rebases or cherry picks): Here, only the committer date will be updated, but the author date stays as is. Consequently, the author date reflects the time of an initial composition, while the committer date reflects the time of an insertion in the history. Both dates are recorded as seconds since the Unix epoch. Changes to the date precision are not supported by Git. For commit creation, users can provide custom dates through environment variables to use instead of the current. This interface could be used by users to manually set dates with reduced precision. This is however not supported for commands that modify commits in bulk. Here, precision reduction is only possible after the fact, by rewriting the history.

3.2 Features for Date Presentation and Filtering

Date Formatting. Date formatting is available for subcommands like `log` and `show` for commit history information, and also for commands that annotate the content of tracked files with commit metadata like `blame` or `annotate`. The formatting option customises how Git renders author and committer dates in the command outputs.

Git offers built-in date formats listed in Table 1 and the option for custom format strings which are passed to the system's `strftime` implementation. The chosen format influences the precision of the displayed date. Five of

Table 2: Git offers predefined (built-in) pretty formats that vary in which dates they show and with what date format (Table 1) those are formatted by default. Some built-ins are fixed to that default and can not be changed by date options.

Built-in	full	oneline	short	medium	reference	email	mboxrd	fuller	raw
Dates used	none			author				both	
Date Format	-	-	-	default	short	rfc	rfc	default	raw
Fixed Format	-	-	-	-	-	✓	✓	-	✓

the eight built-in formats show the full second precision but in different styles like ISO 8601. The others reduce the displayed date precision: *short* omits the time, and both *human* and *relative* use variable precisions that are exact (to the second) when the respective date is recent, and gradually less precise with growing temporal distance. Date formats can be set via the command line option `--date` or via config settings for the `log` and `blame` family of subcommands.

Pretty Formatting. Pretty formatting allows the customisation of commit metadata presentation by commands like `log` or `show`, including the names and email addresses of author and committer as well as the dates mentioned above. Like for date formatting, Git offers built-in formats as well as custom format strings with placeholders for each available piece of commit metadata.

Each built-in implies which dates are used (author, committer, or both) and a date format, that—with some exceptions—can be adapted via date options (see Table 2). In custom formats, the relevant placeholders are `%ad` for author dates and `%cd` for committer dates. The built-in date formats (cf. Sect. 3.2) are available as modifiers. For instance, `%cr` will set the committer date in the *relative* format. Hence, placeholders offer a way to adjust dates separately for each type and independently of other configurations. As shown in Listing 1.1, custom pretty formats can also be set as aliases in a config.

Date Filtering. Some Git subcommands offer limiting their output based on temporal constraints. By passing `--since` or `--until` to `log`, it will list only commits committed since or until the given reference. References can be given in a wide range of syntaxes and formats, as absolute points in time, time distances, and combinations thereof (e. g., *April 2020*, *01/01 last year*, or *8 hours ago*). Git understands a set of common temporal reference points like *midnight* or *yesterday*. We call those *points of reference*.

4 Dataset Acquisition

The basis for our analysis of timestamp precision demand are Git configuration files (configs). To the best of our knowledge, there was no previously available dataset of Git configs or derivations thereof. For that reason, we compiled a dataset based on configs that users published on GitHub. This section describes the identification of the relevant files, their extraction, de-duplication, and the subsequent feature extraction. We also discuss ethical and privacy concerns.

4.1 File Identification and Extraction

Git supports a multi-level hierarchy of configs from the individual repository level to the user and system level [9, git-config]. We limited our data acquisition to user-level configs, in which users typically set their personal preferences and customisations. These configs are located as `.gitconfig` in the user's home directory. GitHub recognises these files in repositories hosted on their service and assigns them the *Git Config* content language tag. To perform automatic searches, we used the code search endpoint of GitHub's REST API [11]. Due to strict rate limiting and the high load that large-scale code searches might induce on GitHub's servers, we added another condition to narrow down the search to configs that include alias definitions. The resulting search is `alias language:"Git Config"`. We ran the acquisition on Sep 17th, 2021. It yielded 23 691 matching files.

The code search API returns a paginated list including URLs to access the matching files. To obtain all files, we had to overcome GitHub's limitation to return at most 10 pages per query with 100 items each. To do so, we built a crawler that finds small enough sub-queries by using file size constraints. We found that result pages are not necessarily filled to their full 100 items, probably due to pre-emptive processing. We extracted 20 757 matches (88%).

4.2 De-duplication

Users might include the same config multiple times in the same or different repositories. To avoid an overrepresentation of users through duplicates, we compared the cryptographic hashes of each config: If a hash occurred more than once within the namespace of the same user, we included only one instance in our dataset. If namespaces differed, we included both instances. We argue that the latter does not constitute a duplicated representation, but a legitimate appropriation by another user and should therefore be counted. Also note that GitHub's Search API does not index forks unless their star count is higher than their parent's [11]. Forks do therefore not introduce unwanted duplicates to our dataset.

We identified and excluded 345 duplicates, i. e., configs that occurred repeatedly in the same namespace. We noticed 554 re-uses of configs in different namespaces. After de-duplication, our dataset comprised configs from 19 468 unique users. 695 users (3.6%) contributed multiple non-identical configs, which accumulate to 1637 configs (8.4%). We argue that those configs should not be excluded, as users might use different configs in different contexts to serve different use cases. Hence, we include them to capture as many use cases as possible.

4.3 Feature Extraction

Having extracted the content of matching configs from GitHub, we subsequently tried to parse each config and extract usage information about the date-related

Table 3: Support for date-related features in Git subcommands.

	annotate	blame	diff-tree	log	rev-list	shortlog	show	whatchanged
date	✓†	✓	✓*	✓	✓		✓*	✓
pretty			✓	✓	✓		✓	✓
since/until		✓		✓	✓	✓	✓*	✓

*=undocumented, †=disfunctional

features described in Sect. 3. In the following, we first describe our process of finding and verifying Git subcommands that support the features in question. After that, we briefly describe the extraction result.

To ensure that all relevant subcommands were regarded during extraction, we first searched and inspected the manual pages of each subcommand for the respective command line options. To compensate for any incompleteness or incorrectness in the manuals, we performed automatic tests to check whether any of the date-related options are accepted *and* make a difference to the resulting output. As a result, we identified discrepancies in terms of undocumented feature support as well as non-functioning documented support (in Git version 2.29.2), which we extracted nonetheless. The feature support is shown in Table 3.

We performed the feature extraction on all downloaded configs. 41 configs could not be parsed due to invalid syntax. The extraction result comprises usage counts for all options described above as well as derived precision information. We have made the dataset available on GitHub [1].

4.4 Potential Ethical and Privacy Concerns

Compiling a dataset of users' Git configurations might raise concerns about the ethics of data extraction or user privacy. We carefully designed our process to address potential concerns.

Code search queries cause a higher load for GitHub than other requests. However, using it enabled us to significantly reduce the total number of queries compared to alternative approaches like searching for repositories named `dotfiles` (about 150 000 matches). Conducting a pure filename based search is also only possible via code search and would have yielded more than three times as many results without the constraint of the `alias` keyword. We are therefore convinced that our approach minimised the load on GitHub compared to other approaches. In general, we followed GitHub's best practices [10].

Regarding user privacy, our dataset only includes configs that users made public on GitHub. The common practice of publishing *dotfile* repositories follows the spirit of sharing knowledge with the community and providing others with a resource of proven configurations. However, we cannot rule out that an unknown proportion of users uploaded their configuration accidentally or not knowing that it will be public. Our extraction is therefore designed to only extract feature usage counts and no free-form data. This ensures that *no* identifying information,

Table 4: Alias definitions and date-related feature usage in the dataset.

Aliases	Mean	Std.	Q_1	Q_2	Q_3
subcommand	15.5	22.4	4	9	19
- shell	3.5	8.6	0	1	4
- log-like	2.3	3.1	1	1	3
- blame-like	0.0	0.2	0	0	0
- filter capable	2.3	3.3	1	1	3
pretty format	0.0	0.3	0	0	0

(a) Descriptive statistics about the absolute frequencies of subcommand aliases with sub-types as well as pretty format aliases per config. (Q_i: i-th quartile)

in %	total	date	pretty	since
annotate	28	0.0	-	-
blame	396	2.0	-	0.0
diff-tree	169	0.0	0.0	-
log	41 467	23.7	76.4	2.5
rev-list	194	0.0	5.7	0.5
shortlog	2202	-	-	4.8
show	3440	9.0	19.0	0.1
whatchanged	553	3.6	14.7	-

(b) Relative frequency of feature usage in aliases for subcommands (if supported).

sensitive data or unwanted disclosures like cryptographic secrets are included. We thus deemed it unnecessary to seek approval from the university ethics board.

5 Data Analysis

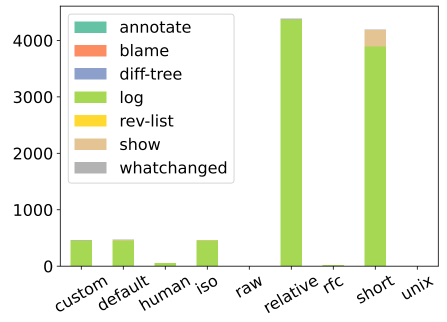
The extracted config features are the basis for our analysis described in this section. To put the findings into perspective, we first provide some basic statistics about the composition of our dataset, before we then analyse users' choices for formatting and filtering, and derive date precisions from them.

We extracted features from 20 369 files. Table 4a provides descriptive statistics about the per-config frequencies of subcommand and pretty format aliases. Overall, we extracted 315 520 definitions of subcommand aliases. On average, each config provided 15.5 such aliases. We identified and excluded in total 71 727 (23%) aliases with shell expressions. Table 4b provides relative occurrences of features accumulated per subcommand. We found that pretty formatting is very commonly used for `log` (76%), but less for other subcommands. Date formatting is used fairly often for `log` as well (24%), but far less for others.

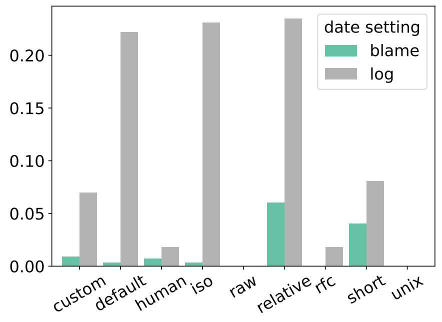
5.1 Date Formatting

We analysed the usage of date formats in subcommand aliases and the two config settings for `log` and `blame`. The usage in subcommand aliases is dominated by aliases for `log` (see Fig. 1a), since almost a quarter of the more than 41 000 `log` aliases use date formatting. The prevalent formats are *relative* and *short*. Aliases for the `show` subcommand predominantly use the *short* option.

We saw a low use of the settings `log.date` and `blame.date`. Only 491 log and 68 blame date formats were set by users, less than 10% of which are custom format strings (see Fig. 1b). The *relative* format is again the most popular, but in contrast to command aliases, *short* is only forth after *iso* and *default*. Note



(a) used with `--date` in command aliases



(b) used for config settings

Figure 1: Distribution of date formatting options used as arguments in subcommand aliases or as config settings. The most frequent options vary noticeably between those contexts: *short* is much less common in settings than in aliases.

Table 5: Date usage in pretty formats across all configuration options.

		built-in [%]				custom [%]			
	total	none	author	committer	both	none	author	committer	both
command aliases	32 430	31.1	0.5	0.0	1.3	9.9	28.1	27.3	1.5
pretty aliases	594	0.0	0.0	0.0	0.3	15.2	48.0	30.0	6.6
format.pretty	603	10.0	2.0	0.0	6.6	1.3	48.6	29.4	2.2

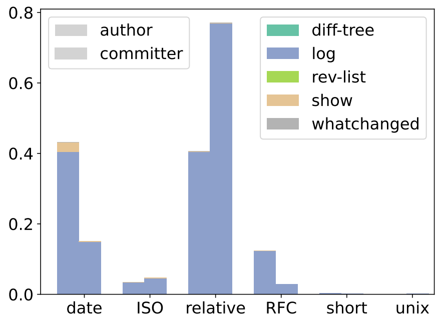
that the high number for the *default* format is due to users having selected the *default-local* option. As localisation does not factor into date precision, we have counted all localised options for their non-localised correspondent.

Due to the overall low adoption of custom date format strings in conjunction with their comparatively complex and system-dependent interpretation, we omitted them from further analysis.

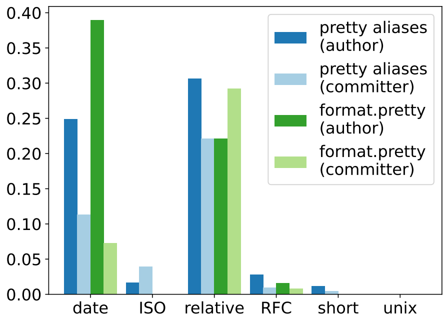
5.2 Pretty Formatting

Date Usage. We analysed which (if any) date types are used in pretty formats for command aliases, `pretty.*` aliases, and the `format.pretty` setting. For built-in formats, we classified the date use according to documentation [9, git-log], which we verified experimentally. For custom formats, we considered a date type as present if the corresponding placeholder (e.g., `%ad` for author) is contained and not escaped. If we encountered the use of a user-defined format alias, we resolved the alias and proceeded as if the resolved format was directly used. The results are shown in Table 5 and described in the following.

Pretty usage in **command aliases** is also dominated by `log`. Over three quarters of all `log` aliases use pretty formatting. The largest proportion of these



(a) custom formats in subcommand aliases



(b) format aliases and `format.pretty`

Figure 2: Relative distribution of date modifier usage in pretty formatting. The *relative* and *date* modifiers are most commonly used across all config options.

formats use no dates at all. 31% are built-ins with no date and about 10% are date-less custom format strings. We found that `oneline` with over 90% is the only built-in with frequent use in aliases. Custom formats with either author or committer date are used in around a quarter of formats each. Formats with both dates make up less than 3% combined. Regarding **pretty format aliases**, we extracted 594 uses, which are almost entirely user-defined formats. Their date usage varies significantly from subcommand aliases: Almost half the formats exclusively use author dates and 30% exclusively use committer dates. 15% contain no dates. Regarding **format.pretty settings**, we found 603 configs that use this feature. Here, most occurrences of built-in formats have no (10%) or both dates (7%). The usage of exclusive author and committer dates in custom formats closely corresponds to our observation for pretty aliases, with about 80% combined. However, only about 1% are custom formats with no date. It appears that demand for date-less formatting is satisfied by built-in formats.

Date Modifiers. As described in Sect. 3.2, custom pretty formats in addition to choosing the desired date type, also allow a rudimentary date formatting.

For **command aliases**, *date* and *relative* are the most common modifiers (see Fig. 2a). More than 75% of committer and 40% of author dates use the *relative* modifier. The *short* date format receives almost no usage. The *date* modifier, which makes the output dependent on `--date` and related settings, is used for about 40% of author dates and about 15% of committer dates. The modifier usage in **pretty format aliases** is illustrated in Fig. 2b (blue bars). Most used is the *relative* format with combined over 50%, followed by the adaptive *date* modifier with about 35%. The usage in **format.pretty settings** is depicted by the green bars in Fig. 2b. Similar to format aliases, *date* and *relative* are by far the most used modifiers with about 40 and 55% each.

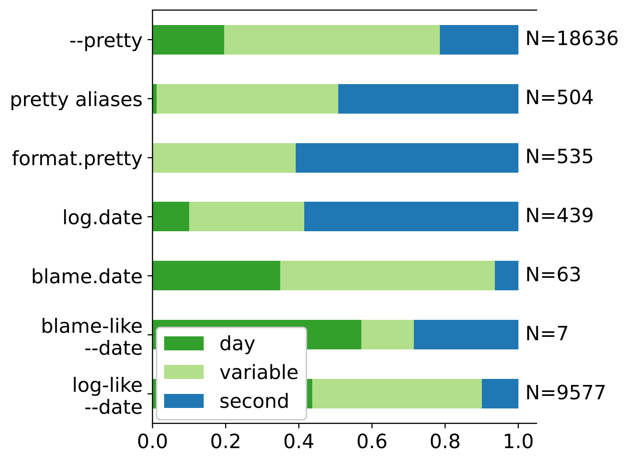


Figure 3: Distributions of date output precision across the formatting options. Second precision is least common in alias definitions for log-like subcommands (regarding `--pretty` and `--date`) which also have the most frequent use.

5.3 Resulting Date Output Precisions

Based on the previous analysis, we could determine the precisions of dates displayed as a result of using the extracted configs. The precision directly follows from the used date format (see Table 1) and can be either *second*, *day*, or *variable*. The effective precision of variable formats depends on the recency of the event and ranges between day and second. As we cannot resolve this variability, we will leave it as a third precision in between. Formats with *no* dates are not considered in this section.

For **subcommand aliases** supplying explicit `--pretty` formats, we proceeded differently for built-in and custom formats: For built-ins a with fixed date format, the precision directly follows from the fixed format. For instance, `email` hard-codes the *rfc* format which has a precision of seconds. The precision of all other built-ins is determined by the `--date` option, or—if none is given—by the `log.date` setting or its default, the *default* date format. The same date format resolution applies, if custom pretty formats use the *date* modifier. Otherwise, the resulting precision directly follows from the used modifier. Note that the about 1.5% of custom pretty formats that use both date types could therefore use a different precision per type. In that case, we considered the higher precision of the two for our further analysis. For that purpose, we used the following sort sequence of precisions: day < variable < second. Fig. 3 illustrates, e. g., that 60% of subcommands' pretty formatting that contains date information effectively display it with a variable precision. And over 90% of pretty-capable (log-like) aliases that supply `--date` options display with variable or day precision.

We applied the same evaluation to pretty formats set as **format aliases** and the **format.pretty** option. Since both settings are taken outside the context of a command invocation, considering possible `--date` options is not applicable. Otherwise, we followed the process described above to determine the applicable date format, including considering potential `log.date` options. In contrast to subcommand aliases, day precision outputs are negligible and second precision output is much more common with about 50 to 60% (see Fig. 3).

5.4 Date Filters

We found that date filter usage is again dominated by `log`. In general, it appears to be an infrequently used feature, with only 2.5% among `log` aliases. In relative terms, it is most commonly used in `shortlog` aliases. We also found that among the date filtering options, `--since` makes up for almost the entire feature usage, whereas `--until` is almost exclusively used in combination with `since`. All figures include the alternative names `--after` and `--before`.

Extraction Methodology. In contrast to date formatting, the precision of filters does not follow directly from a set of predefined options. Moreover, the leniency of the filter parser makes it difficult to cover all allowed inputs during precision classification. For that reason, we decided to directly use Git's parser code for our analysis. We sliced the responsible functionality from the official Git source code [8, v2.32.0-rc2] and linked the functions with our analysis tool. We instrumented Git's date parser at 25 locations to keep track of the smallest unit of time addressed by a filter. This is illustrated by the following two examples:

$$\underbrace{\text{1 hour}}_{\text{hour}}\ \underbrace{\text{30 minutes}}_{\text{minute}}\ \text{ago} \qquad \underbrace{\text{yesterday}}_{\text{day}}\ \underbrace{\text{5pm}}_{\text{hour}}$$

In the first example, the smallest unit is given in minutes, so we consider the filter to have minute precision. In the second example, the smallest unit is given by the full hour, thus we consider the filter to have hour precision. We excluded date filters containing shell command substitutions, of which we identified 26 (2.2%). Another 7 were rejected by Git as invalid, leaving 1156 valid filters.

Precision Classification. When classifying date filter precision, the question arises whether the hour 0 should be treated as hour precision like every other hour value, or as an indicator for the lower day precision. In order to not underestimate the demand for precision, we assumed the hour precision. This is also in concordance with the *midnight* point of reference (POR). The available precision levels are the date unit based precisions year to second (including week), supplemented by the *undefined* precision which is assigned if date filters use the PORs *now* or *never* which allow no classification. Fig. 4 illustrates the resulting precisions. Most date filters are in the *day* precision (46%), followed by *hour* (23%) and *week* (18%). Precisions higher than *hour* make up less than 0.5%.

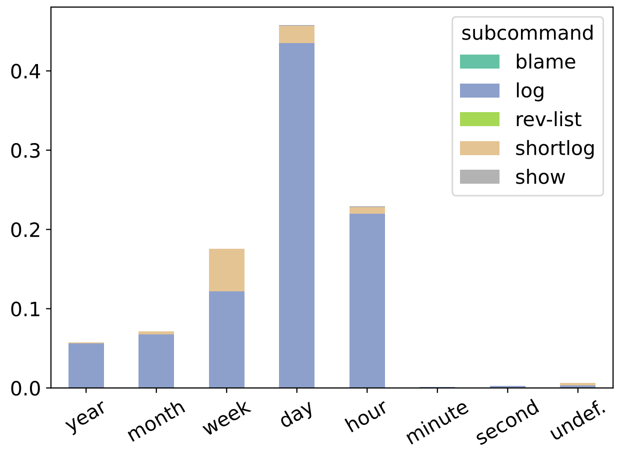


Figure 4: Overall distribution of precisions that are implied by the date filters used for the `--since` or `--until` options in subcommand aliases. Day precision is by far the most common. Filters with precisions higher than hour have almost no use at all.

6 Discussion

In the following, we discuss the privacy gain and functionality loss related to precision reduction, as well as possible objections to the representativeness of our dataset. Additionally, we present options to reduce date precision in Git.

6.1 Privacy Gain and Functionality Loss

In principle, the GDPR mandates data minimisation regardless of achievable privacy gains. Legally, the necessity of data needs to be argued and not its harmfulness to privacy. Nonetheless, to evaluate technical minimisation approaches, some notion of privacy gain might be of interest. Benchmarking the effectiveness of timestamp precision reduction based on known inference techniques is however highly context specific and ignorant to future technical developments. Moreover, timestamp-specific inference techniques are scarce (cf. Sect. 2). Instead, we argue that a data-oriented evaluation of statistical properties like changes in distribution are more conclusive of discriminatory power and minimisation effects. For instance, the number of distinguishable activity points over a given period expresses an attacker's ability to observe intervals between actions, which might be used to monitor users' throughput. Following that method, we evaluated the effect of different precision reductions on real-world Git data. This provides additional empirical evidence on the question whether a reduction within the precision range of our previous demand analysis, i. e., no less than day precision, could meaningfully improve user privacy.

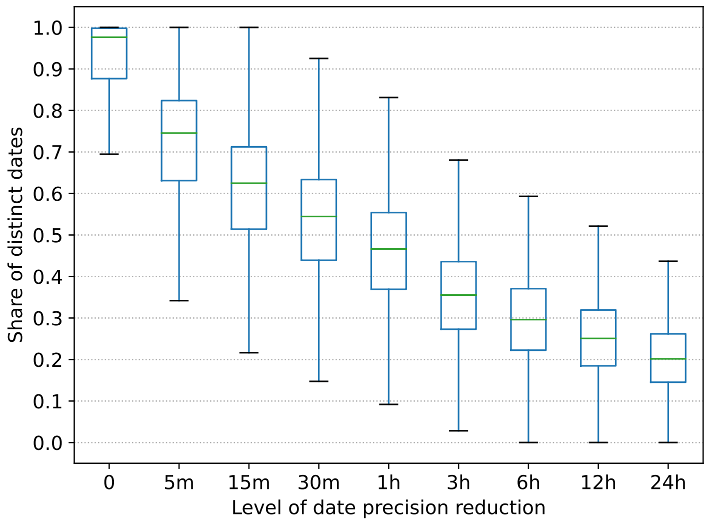


Figure 5: The share of users' distinct Git dates decreases fast with increased precision reduction, as evaluated on a GitHub snapshot of 360 million commits. At a 1 hour precision level, more than half of the median user's timestamps are indistinguishable from their chronological predecessors, thus preventing the inference of the temporal intervals between the respective activities.

We obtained commits from a GitHub mirror curated by the GHTorrent project [12], which contains all public GitHub activity since 2012. Based on a snapshot from April 1st, 2019, we extracted all commits from users with a total of one to ten thousand commits each, calculated over the full lifespan of the dataset. We argue that this sample of users adequately represents frequently active users without introducing much bias by bot-driven accounts that is expected to increase on more active accounts, given that ten thousand commits equates to almost four commits per day, for *every* day in the scope of the dataset. To nonetheless compare the findings, we also performed the analysis on the sample of users with between 10 and 100 thousand total commits. The expectation being, that the more commits a user has, the higher their activity density, and hence the higher the observed precision reduction effects. In total, we analysed 360 million commits by 160 thousand users in the range of one to ten thousand commits, and 100 million commits by 5000 users in the range of 10 to 100 thousand commits.

For each user, we counted the number of distinct activity points in time at various precision reduction levels from five minutes to one day. Activity points in time are given by the commit dates and are regarded as distinct, if—after applying the precision reduction—the remaining significant date information differs. The precision reduction is applied by rounding towards the next smaller integral multiple of the precision. As Fig. 5 shows, a 5 minute reduction level already results in only 75% distinct dates (median), and less than 50% at 1 hour. With the sample of very active users, we saw 54% and 28% for 5 minutes and 1 hour respectively. This indicates that a moderate precision reduction already prevents monitoring of intervals for a significant share of activities.

Functionality loss on the other hand is relevant to evaluate the cost of minimisation techniques. Such loss could be caused to the minimised application itself or to attached processes and workflows. As Git itself does not programmatically use commit dates but only passes them on, there is no direct loss of functionality or integrity. Usability should only be effected in the sense that users get unexpected results, e. g., for filtering, if they were unaware of the precision reduction. For instance, if a commit occurred within the last minute but was reduced to hour precision, a filter for *until 30 minutes ago* would list this command, provided no further precautions where taken. Such precautions could be to show a notice if filters conflict with timestamp precision or to reject them. The extent to which precision reduction affects Git workflows is of course very subjective. Our empirical data on chosen display and filter precisions is one indicator for reduction impact. Any reduction within the range of those commonly chosen precisions would have limited loss for workflows based on our analysed features. In qualitative interviews with four DevOps workers of different seniority, their stated interest in timestamp precision varied from no interest to precise oversight of team activities. This underlines our assumption that workflow-related interest in precise timestamps might be more driven by individual mannerism than procedural necessities. As user privacy should not be left to individual discretion, tools like Git should support to enforce the precision levels agreed upon on a per-team basis.

6.2 Representativeness and Limitations

Configurations on GitHub might not be representative for the overall user base of Git. Only users that desire a behaviour different from the default even make certain settings like aliases. However, a motivation to define aliases in general, is to make frequently used commands and arguments more easily accessible. Such settings are therefore not necessarily motivated by a wish to change default behaviour. We argue that the subset of users that define aliases is therefore not necessarily biased towards a date-related behaviour that differs from the default. The analysed settings might require a more experienced Git user to discover and use them. In that sense our analyses might be biased towards such users. To assess whether experience influences precision demand, future research could correlate our precision analysis with, e. g., commit counts. We argue that experience certainly factors into the discoverability of options, but presumably less into their configuration. Whether or not users need second-precision dates in a log output is likely unrelated to their experience.

6.3 Timestamp Reduction Approaches and Tools

Timestamp reduction could be applied on the presentation level, but to hinder performance monitoring and not be easily circumventable, it should be applied during recording. Wherever precision demand is highly user-specific, the recorded precision should be customisable. Nonetheless, a privacy-friendly default should be chosen that reflects most needs. If Git users wish to reduce the precision with which their actions are timestamped, they find no support to do so in Git today. And as dates are included in the input to the hash function that determines the commit hash, retroactive reductions interfere with hash chaining and history keeping. As such, modification to the dates might cause diverging Git histories. To nonetheless provide users with the option to reduce their timestamp precision, we built `git-privacy` [5], a tool that uses Git hooks to reduce timestamps while avoiding conflict with previously distributed states. It uses a unit annihilation approach similar to the rounding down described in Sect. 6.1, where users can choose the most significant time unit that should remain precise. In systems like Git with integrity-protected timestamps, at least excluding higher-precision timestamp parts from the integrity protection would allow post-recording reduction policies to take effect without compromising the history.

7 Summary and Conclusion

Using Git config files that users published on GitHub, we have compiled and analysed a large-scale dataset of features related to users' demand for timestamp precision. Our analysis of the usage of date and pretty formatting as well as date filters indicates that Git's current behaviour of recording dates to the precise second might not be justified by user demand. In fact, we found that when users customise output of subcommand aliases, over 40% of formats omit dates entirely. And of the remaining formats, 80% display dates with a reduced variable or static day precision. As a result, a static full second precision is not utilised by nearly 90% of all subcommand pretty formats. Similarly, over 90% of date formatting in subcommand aliases uses variable or day precision. We saw a higher ratio of second precision output in pretty aliases as well as format and date settings, which could be due to users picking default date formats in pretty format stings, and due to a preference for ISO-style output. Our analysis of date filters found that only 0.5% of filters would require a precision of minute or second. In fact, 74% require a precision of day or less. All in all, we believe that our analysis provides strong empirical evidence, that user demand for precision can be met with less than second-precise timestamps. Our evaluation of possible privacy gains suggests that small precision reduction levels of a few minutes already have significant effects. As Git itself does not require any date precision, making it configurable would not only allow teams to define appropriate levels for their use case, but also facilitate a more GDPR-compliant use in companies. We encourage software engineers to employ reduced and adaptive precision timestamping for more proportionate solutions.

Acknowledgements. We would like to thank the anonymous reviewers and Vaclav Matyas for their constructive and very helpful suggestions to improve this paper. The work is supported by the German Federal Ministry of Education and Research (BMBF) as part of the project Employee Privacy in Development and Operations (EMPRI-DEVOPS) under grant 16KIS0922K.

References

1. Burkert, C.: .gitconfig Date Study Dataset, (2022). `https://github.com/EMPRI-DEVOPS/gitconfig-study-dataset`
2. Burkert, C., and Federrath, H.: Towards Minimising Timestamp Usage In Application Software - A Case Study of the Mattermost Application. In: DPM (2019)
3. Claes, M., Mäntylä, M.V., Kuutila, M., and Adams, B.: Do Programmers Work at Night or During the Weekend? In. ICSE. ACM (2018)
4. Drakonakis, K., Ilia, P., Ioannidis, S., and Polakis, J.: Please Forget Where I Was Last Summer: The Privacy Risks of Public Location (Meta)Data. In: NDSS (2019)
5. EMPRI-DEVOPS: git-privacy, `https://github.com/EMPRI-DEVOPS/git-privacy`
6. Eyolfson, J., Tan, L., and Lam, P.: Do Time of Day and Developer Experience Affect Commit Bugginess? In: MSR. ACM (2011)
7. Game World Observer: Xsolla cites growth rate slowdown as reason for layoffs, CEO's tweet causes further controversy, (2021). `https://gameworldobserver.com/?p=10949`
8. Git: Git Source Code, (2021). `https://github.com/git/git`
9. Git: Reference, (2022). `https://git-scm.com/docs` (visited on 28th Mar. 2022)
10. GitHub Docs: Best practices for integrators, (2021). `https://docs.github.com/en/rest/guides/best-practices-for-integrators` (visited on 24th Sept. 2021)
11. GitHub Docs: Search API, (2021). `https://docs.github.com/en/rest/reference/search` (visited on 24th Sept. 2021)
12. Gousios, G.: The GHTorrent dataset and tool suite. In. MSR '13 (2013)
13. Mavriki, P., and Karyda, M.: Profiling with big data: Identifying privacy implications for idividuals, groups and society. In: MCIS (2018)
14. Senarath, A., and Arachchilage, N.A.G.: Understanding Software Developers' Approach towards Implementing Data Minimization, (2018). `https://arxiv.org/abs/1808.01479`
15. Slagell, A.J., Lakkaraju, K., and Luo, K.: FLAIM: A Multi-level Anonymization Framework for Computer and Network Logs. In: LISA, pp. 63–77. USENIX (2006)
16. Traullé, B., and Dalle, J.-M.: The Evolution of Developer Work Rhythms: An Analysis Using Signal Processing Techniques. In: Social Informatics (2018)
17. Wright, I., and Ziegler, A.: The Standard Coder: A Machine Learning Approach to Measuring the Effort Required to Produce Source Code Change. In: RAISE (2019)