How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness

Wolfram Fenske
University of Magdeburg
Magdeburg, Germany
wfenske@ovgu.de

Sandro Schulze
University of Magdeburg
Magdeburg, Germany
sanschul@ovgu.de

Gunter Saake
University of Magdeburg
Magdeburg, Germany
saake@ovgu.de

Abstract

Preprocessor annotations (e. g., `#ifdef` in C) enable the development of similar, but distinct software variants from a common code base. One particularly popular preprocessor is the C preprocessor, CPP. But the CPP is also widely criticized for impeding software maintenance by making code hard to understand and change. Yet, evidence to support this criticism is scarce. In this paper, we investigate the relation between CPP usage and maintenance effort, which we approximate with the frequency and extent of source code changes. To this end, we mined the version control repositories of eight open-source systems written in C. For each system, we measured if and how individual functions use CPP annotations and how they were changed. We found that functions containing CPP annotations are generally changed more frequently and more profoundly than other functions. However, when accounting for function size, the differences disappear or are greatly diminished. In summary, with respect to the frequency and extent of changes, our findings do not support the criticism of the CPP regarding maintainability.

CCS Concepts • **Software and its engineering → Software product lines**; *Maintaining software*;

Keywords preprocessors, annotations, variability, maintenance, change-proneness

ACM Reference Format:
Wolfram Fenske, Sandro Schulze, and Gunter Saake. 2017. How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experience (GPCE'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3136040.3136059

GPCE'17, October 23–24, 2017, Vancouver, BC, Canada

ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136059

1 Introduction

Highly configurable software systems are widely used to efficiently develop a whole set of similar programs and, thus, flexibly adapt to different requirements, such as resource constraints or different platforms. Such systems are successful mainly because development and maintenance takes place in a single code base in which optional or alternative parts of the implementation are expressed by variability implementation mechanisms. A popular mechanism are annotations, such as the `#ifdef` directives provided by the C preprocessor, CPP [12, 27, 32], which is shipped with the C programming language [23]. By using `#ifdef`s, developers can indicate code fragments that are only conditionally included in the final program, depending on the evaluation of the expression associated with the annotation. Due to its simple and language-independent nature, the CPP is a powerful, expressive means to implement variability [27].

Despite its advantages, the CPP is criticized for making programs harder to understand and maintain, and for being a frequent source of subtle errors [47, 13, 12, 28, 33, 32, 36]. For instance, an excessive number of `#ifdef`s, complex feature expressions, or the practice of annotating incomplete syntactical units (so-called *undisciplined annotations* [28, 44, 34, 30]) may obfuscate the source code of the underlying programming language, hindering program comprehension. Some progress has been made on investigating the CPP critique empirically [33, 44, 35, 32, 36, 30, 34], mainly focussing on the discipline of annotations and errors related to CPP usage. However, maintainability is only partially discussed [44, 30] and only for specific types of annotations.

We extend this research by investigating the effects of the CPP on maintainability. Specifically, we present a study of the correlation between `#ifdef` usage and change frequency as well as code churn, both of which are associated with increased fault-proneness and maintenance effort, respectively [9, 18, 10, 37, 45]. Our first objective is to find out whether the presence of CPP annotations affects maintenance effort. We investigate the following research questions:

RQ 1: *Is feature code harder to maintain than non-feature code?*

RQ 2: *Does the presence of feature code relate to the size of a piece of code?*

For RQ 1, we consider different properties of annotation usage, such as the number of `#ifdef` directives or nesting of

directives. With RQ 2, we acknowledge findings that many metrics correlate with the size of a piece of code [11, 49, 39, 25, 19]. Size, in turn, correlates with various maintenance aspects, such as change- and fault-proneness [45, 18].

Our second objective is understand *how* different properties of CPP annotations contribute their influence on maintenance. As stated by previous studies, the shape of variable code differs from system to system and even within a single system [41]. Hence, insights into how the shape of annotations (e. g., nesting, complex feature expressions, amount of annotated code) affects change-proneness help developers identify the main drivers of their maintenance efforts. Moreover, such information can be the basis for guidelines, best practices, and new refactorings. We formulate the following research question regarding our second objective:

RQ 3: *Considering all properties of annotation usage and size in context, what is the independent effect of each property on maintainability?*

To answer these questions, we analyzed how individual functions use annotations and how they are changed. The data were mined from the version control repositories of eight open-source systems written in C. Statistical analyses of this data suggest that functions with annotations are changed more frequently and more extensively than other functions. The difference, however, is small when differences in function size are accounted for. In particular, we make the following contributions:

- Quantitative insights into the relationship between fine-grained annotation use and change-proneness,
- A methodology and publicly available tool support[1] to mine fine-grained information on annotation use and changes from version control repositories,
- Publicly available replication data.[1]

2 C Preprocessor-Based Variability

In this section, we summarize how the C preprocessor, CPP, is used to implement variability in highly configurable software systems. Moreover, we introduce the terminology we use to characterize variable source code in this paper.

Invented as part of the C programming language, the task of CPP is to transform C source code for subsequent compilation by the C compiler. This transformation is controlled by *directives* in the source code and by *macros* specified on the command line or in the source code. Directives allow the inclusion of definitions from other files (via `#include`) or the definition (via `#define`) of short-hands, called macros, which CPP will expand during preprocessing. Most important for our work, however, is the family of directives that enable *conditional compilation*. The members of this family are `#if`, `#else` and `#endif`, as well as the convenience forms `#elif` (for `#else if`), `#ifdef` (for `#if defined`) and `#ifndef` (for `#if !defined`). In the remainder of this paper, we will refer to these directives as `#ifdef`s. Conditional compilation allows including or excluding pieces of source code during compilation depending on some condition. When CPP encounters an opening `#ifdef`, it evaluates the corresponding expression. If this expression evaluates to true, the following code is further preprocessed. Otherwise, CPP takes the code in the `#else` branch into account, if there is one. Conditional preprocessing stops at the closing `#endif`.

Conditional compilation is frequently used to implement variability in highly configurable software systems [1, 27]. In these systems, code that implements a particular *feature* (a. k. a. *feature code*) is guarded by `#ifdef`s. The expression controlling the `#ifdef` is called a *feature expression*. In this context, a feature is a requirement that is important to a customer or some other stakeholder [1], and the name of the macro in the feature expression is called a *feature constant*.

In Figure 1, we show an exemplary function from MySQL version 5.6.17 that makes heavy use of `#ifdef`s to implement variability. The function contains five blocks of feature code, e. g., on Lines 5–8, and 9–11. The last block, on Lines 20–22, is *nested* within the block on Lines 19–25. The CPP will only consider that last block if the feature expression of the outer block is true. Most feature expressions in this example are *atomic*, that is, they only depend on a single feature constant. For instance, the expression on Line 14 only depends on `USE_ALARM_THREAD`. Atomic expressions can be combined using boolean operators to form *complex expressions*. For instance, the expression on Line 5 means that the feature

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2    sigset_t old_mask;
3    if (thd_lib_detected == THD_LIB_LT &&
4        !pthread_equal(pthread_self(), alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13   }
14 #ifndef USE_ALARM_THREAD
15   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
16   mysql_mutex_lock(&LOCK_alarm);
17 #endif
18   process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(
       SIGNAL_HANDLER_RESET_ON_DELIVERY)
21   my_sigset(THR_SERVER_ALARM, process_alarm);
22 #endif
23   mysql_mutex_unlock(&LOCK_alarm);
24   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25 #endif
26   return;
27 }
```

Figure 1. Example of a function in MySQL making heavy use of `#ifdef`s to implement variability.

[1] https://wfenske.github.com/IfdefRevolver/ifdefs-vs-changes/

code on Lines 6 and 7 is only included if the feature constant `MAIN` is defined and if `__bsdi__` is not.

The use of `#ifdef`s leads to *tangling* and *scattering*. Both are believed to make code hard to understand and change [47, 12]. Tangling means that code related to one feature is mixed with code related to other features. Tangling is caused by complex feature expressions, nesting, or a combination of both (see Lines 5, 19 and 20 in the example). Scattering means that code related to one feature is located in multiple places. In the example, `SIGNAL_HANDLER_RESET_ON_DELIVERY` is implemented by code scattered across Lines 10 and 21.

3 Methodology and Research Design

In this section, we present the design of our exploratory study, including research hypotheses, data collection, and statistics to test our hypotheses.

3.1 Measuring Maintainability

In Section 1, we introduced three research questions, aimed at two objectives. First, we want to investigate whether the presence of CPP annotations affects maintainability. Second, we want to gain insights into the individual, independent effect of specific properties of CPP annotations on maintainability. For both objectives, we need a measure for maintainability. Unfortunately, a direct measurement requires (controlled) experiments. This limits the number of systems and annotations to be investigated. Hence, we use *change-proneness* as a proxy for maintenance effort. Change-proneness comes in two flavors: the frequency of changes (e. g., used in Khomh et al. [24, 25]) and the amount of changes (a. k. a. *code churn*; e. g., used in [42]). While the former is a good predictor of later defects [9, 18], the latter correlates with the effort of developers when performing maintenance tasks [10, 37, 45]. Our study considers both measures.

The level of granularity of our analysis is crucial. As CPP annotations can be used at a fine grain, a similarly fine-grained analysis is required to investigate possible effects. Hence, our analysis works at the function level, i. e., we measure annotation use and changes for each function individually.

3.2 Null-Hypotheses

In this section, we state the null-hypotheses for our research questions. Since we use two proxies of maintenance effort, we formulate two null-hypotheses for each question.

RQ 1 For the first question, we define four properties of preprocessor use: (1) Whether a function contains an `#ifdef` directive, (2) whether the `#ifdef` directives in a function reference two or more feature constants, (3) Whether the function contains nested directives, (4) Whether the directives in a function use negation. Note that each `#ifdef` directive references at least one feature constant. Therefore, we chose *two* feature constants as the threshold for the second property to distinguish it from property one. For each property, we investigate whether functions with this property are more change-prone than functions without. To this end, we formulate the following null-hypotheses and test them for all four properties (though we formulate them in general).

H_0 1.1 *Functions exhibiting a specific property of preprocessor use are changed just as frequently as functions without the property.*

H_0 1.2 *Functions exhibiting a specific property of preprocessor use are changed just as profoundly as functions without the property.*

RQ 2 We investigate the relationship between preprocessor usage and size by testing this null-hypothesis:

H_0 2.1 *The extent of preprocessor usage in functions is unrelated to the size of the functions.*

Moreover, we normalize the frequency and amount of changes by function size and reevaluate the answers to RQ 1 using these normalized measures. Again, we focus on the properties defined for RQ 1. The null-hypotheses are:

H_0 2.2 *The number of changes normalized by function size is equal for functions with and without a specific property of preprocessor use.*

H_0 2.3 *The number of lines changed normalized by function size is equal for functions with and without a specific property of preprocessor use.*

RQ 3 In RQ 1 and RQ 2, different properties of preprocessor usage are studied in isolation. For RQ 3, we investigate their combined effect and test the following null-hypotheses:

H_0 3.1 *When seen in combination, also taking function size into account, different aspects of annotation usage do not affect how frequently functions are changed.*

H_0 3.2 *When seen in combination, also taking function size into account, different aspects of annotation usage do not affect how profoundly functions are changed.*

3.3 Subject Systems

We chose our subjects from a set of well-known systems used in previous studies of preprocessor usage [27, 21, 41]. We used the following seven criteria to guide our selection.

1. implemented in C (as it uses CPP)
2. considerable fraction of feature code (since it is of main interest in our study)
3. open-source software (as we need access to the repository for extracting change information)
4. GIT version control system (due to our tooling)
5. considerable development history (to minimize the effect of outliers in small data sets)
6. at least medium-sized so that a sufficient amount of feature and non-feature code is available
7. systems from different domains (to avoid bias).

Based on these criteria, we selected eight systems for which we give an overview in Table 1. In particular, we

Table 1. Subject Systems

System	Analyzed Period	Commits	Files	f	(annot.)	kLOC$_f$	(annot.)	Domain
Apache	1/1996 – 5/2017	55 767	320	5 035	(7.4 %)	150.8	(3.9 %)	Webserver
BusyBox	10/1999 – 5/2017	14 993	641	4 446	(14.6 %)	134.0	(11.6 %)	Unix command line tool suite
glibc	5/1972 – 4/2017	45 112	8 277	11 337	(9.6 %)	330.7	(7.2 %)	GNU version of the C standard library
libxml2	7/1998 – 6/2017	4 623	111	5 653	(25.3 %)	179.4	(19.0 %)	XML parser and toolkit
OpenLDAP	8/1998 – 6/2016	29 566	514	5 402	(12.0 %)	229.3	(6.5 %)	LDAP directory service
OpenVPN	9/2005 – 6/2016	2 304	92	1 963	(16.5 %)	45.0	(12.9 %)	Secure network communication
Pidgin	3/2000 – 6/2017	38 656	623	13 248	(4.2 %)	283.7	(3.4 %)	Instant messaging client
SQLite	5/2000 – 5/2017	18 722	249	6 804	(10.5 %)	183.5	(9.0 %)	Database management system

specify the number of `.c` files (based on the latest version), the number of function definitions (column "f"), and, in parentheses, the percentage of functions using preprocessor annotations. Moreover, we report the number of non-blank, non-comment lines of code comprised by function definitions (column "kLOC$_f$") and, in parentheses, the percentage of that code enclosed in annotations.

Note that the average percentage of feature code in our subject systems is 9.2 %, much lower than the 24 % that Hunsen et al. reported for their subjects [21]. This difference is rooted in the way we measure feature code. Hunsen et al. measure feature code on the file level. Thus, if a whole function definition is enclosed in an `#ifdef` directive, they count the whole function definition as feature code. We, by contrast, measure feature code on the function level. Thus, we only consider a line as feature code if the enclosing `#ifdef` directive is inside the function body.

3.4 Data Collection

In this section, we describe which data we collected from our subject systems and how. The data falls into two categories: first, static source code properties (e. g., the number of `#ifdefs` in a function) and second, change metrics (e. g., the number of lines added or deleted). We list the metrics in Table 2 and Table 3. We discuss them in detail in Section 3.5.

Our data collection process comprises five tasks. First, we identify all relevant commits from the subject's repository. Second, we group relevant commits into *snapshots*, which can be seen as time slices of the system's development history. Third, we extract the proposed static and change-centric metrics for each snapshot. Fourth, we exclude snapshots exhibiting anomalies that might confound our analysis. Finally, snapshots are combined into larger units, called *commit windows*. Next, we describe these tasks in detail.

Identifying relevant commits As the first step in our data collection process, we extract all relevant commits from the respective Git repository. A commit is relevant if it modifies at least one `.c` file as this is where functions in C are defined. Commits that only modify header files, shell scripts, etc. are irrelevant. We also ignore merge commits, which are created when integrating changes from another branch.

Creating snapshots Enhancements, bug fixes, etc. are increasingly developed in different branches. Upon task completion, all commits of a branch are merged into the master branch. Since many branches can exist in parallel, chronologically consecutive commits may not represent logically consecutive steps of development. To restore the logical order, we group commits into *chains*, where each chain forms an unbroken sequence of commits that are in a parent-child relationship. For merge commits, which have more than one parent, we must decide which chain to end and which to continue. To this end, we use a simple greedy heuristic that favors creating longer chains over shorter ones. Since the chains can comprise different numbers of commits, we divide them into equal-sized *snapshots*, so that all snapshots represent roughly the same development effort.

Determining snapshot size is a trade-off between speed and precision. While smaller snapshot sizes increase storage consumption and analysis time, too large snapshots bear the risk of losing information, thus, confounding our analysis. In particular, we compute our static metrics (cf. Table 2) just once per snapshot. Hence, if snapshots are too large, these metrics may become invalid due to excessive changes in the course of the snapshot. Also, the risk increases that functions are added in the middle of a snapshot, which we would miss as we only track changes to functions that already exist at the beginning of a snapshot. To determine an appropriate size, we performed an initial experiment with snapshot sizes of 50, 100, 200, and 400 on OpenLDAP.[2] We found few negative effects for a snapshot size of 50, and only slightly more for 100. Afterwards, negative effects increase noticeably. As a compromise between quality and speed, we chose 100 as the snapshot size for all systems.

Processing snapshots Each snapshot is processed in five steps, (1) checkout, (2) preprocessing, (3) gathering static source code metrics, (4) change analysis, and, finally, (5) combining static metrics and change information.

The first step is to checkout the first revision of each snapshot and copy all `.c` files to a dedicated directory.

In the second step, preprocessing, the files are converted to SrcML [4], an XML representation that is easier to parse than

[2] Detailed results can be found on the complementary website.

Table 2. Static source code metrics extracted for each function

Metric	Description
FL	Number of blocks annotated with an `#ifdef`. An `#ifdef` containing a complex expression, such as `#if defined(A) && defined(B)`, counts as a single feature location. An `#ifdef` with an `#else` or `#elif` branch counts as two feature locations.
FC	Number of feature constants referenced by the feature locations in the function. Constants referenced multiple times are counted only once.
CND	Cumulative nesting depth of annotations. An `#ifdef` that is not enclosed by another `#ifdef` (a *top-level* `#ifdef`) has a nesting depth of 0; an `#ifdef` within a top-level `#ifdef` has a nesting depth of 1, and so on. *CND* is accumulated over all feature locations within the function. Thus, for example, if there are two feature locations in a function, each with a nesting depth of 1, *CND* of the function will be 2.
NEG	The number of negations in the `#ifdef` directives in a function. Both `#ifndef X` and `#if !defined(X)` increase *NEG* by 1. `#else` branches also increase *NEG* because `#if <expr> ... #else ...` is treated as `#if <expr> ... #endif #if !<expr> ...`
LOAC	Source lines of code in all feature locations within the function. Lines that occur in a nested feature location are counted only once.
LOC	Source lines of code of the function, ignoring blank lines and comments.
$LOAC_{/LOC}$	Proportion of *LOAC* to all code in the function, i. e., *LOAC* / *LOC*.

Table 3. Change metrics extracted for each function

Metric	Description
COMMITS	The number of commits that have modified the function definition.
LCHG	The number of lines added plus the number of lines removed, accumulated over a period of time.
$COMMITS_{/LOC}$	The number of commits to a function divided by its *LOC*.
$LCHG_{/LOC}$	The number of lines changed in a function divided by its *LOC*.

raw C code. Using the SrcML representation, comments and empty lines are removed. Moreover, `#ifdef`s are normalized to ease subsequent analysis. For instance, `#ifndef FEATURE` is transformed to the equivalent `#if !defined(FEATURE)`.

Third, we parse all SrcML files to identify the function definitions that exist at the beginning of the snapshot. For each function, we collect the metrics listed in Table 2.

In the fourth step, we analyze each commit within the snapshot. For each modification we identify, we determine the function definition it belongs to and how many lines of code were added to or removed from the function body. Subsequently, we aggregate this change data for all commits of the snapshot, yielding the metrics listed in Table 3.

Finally, the data from the previous two steps is combined. The result is a list of all functions in the snapshot, along with their static source code metrics and change information (amount and frequency).

Exclusion of snapshots Several subject systems contain a small number of anomalous snapshots. To prevent them from distorting our analyses, they are excluded. For instance, OpenLDAP includes a snapshot from mid 2011, starting at revision `d620d4368` and consisting of only two files related to a helper library. By contrast, the snapshots immediately before and after this one comprise around 470 files. As there may be several reasons for such anomalies, we checked for such snapshots manually and excluded them.

Formation of commit windows The small number of commits within our snapshots restrains our ability to observe functions that undergo a large number of commits and heavy changes. This makes it difficult to distinguish between functions that are truly change-prone and those that are not.

We thus grouped 10 snapshots each into a *commit window*. To preserve the logical order of changes, only snapshots belonging to the same chain were grouped. If less than 10 snapshots were left within a chain, we discarded them. However, this was the case for only a few snapshots.

During grouping, we match records from multiple snapshots that belong to the same function and aggregate them into a single record. As a result, the aggregated record is computed by averaging the static metrics (e. g., *LOC*), summing up the change metrics (e. g., *COMMITS*), and recomputing the ratios (e. g., $COMMITS_{/LOC}$).

Compared to the individual snapshots, the frequency and amount of changes captured by a commit window increases substantially. For example, in OpenLDAP, the average amount of changed functions per window increased from 7 % to 31 %. Among the changed functions, the portion of functions undergoing more than one commit rose from 23 % to 44 %. In summary, commit windows facilitate the distinction between change-prone functions and other functions and makes our analysis more robust.

3.5 Statistical Analyses

We answer our research questions by investigating correlations between the use of preprocessor annotations and change-proneness. To this end, we apply different statistical methods. In this section, we describe those methods and the independent and dependent variables involved.

3.5.1 Answering RQ 1

We use a binary classification scheme to test H_0 1.1 and H_0 1.2. Basically, we combine the data of all commit windows for a system into one large data set, which is then split into an experimental group and a control group. Functions that exhibit one of the four properties of preprocessor use defined

in Section 3.2 belong to the experimental group, the others to the control group. We then test if and how much change-proneness differs between experimental and control group.

Due to the four different properties, we create four pairs of experimental and control group, one for each property. To this end, we introduce four binary variables, $fl_{>0}$, $fc_{>1}$, $cnd_{>0}$, $neg_{>0}$, which serve as our independent variables. The definitions are: Let f be a function, then $fl_{>0} = 1$ if f's value for the *FL* metric exceeds 0; otherwise, $fl_{>0} = 0$. We define the other variables similarly, based on the corresponding metrics. Given these definitions, a function belongs to the experimental group if and only if the respective variable is 1.

To test H_0 1.1, we consider *COMMITS* as our dependent variable and apply the *Mann-Whitney-U* test. The test will reveal if there is a statistically significant difference in the values of *COMMITS* in the experimental group, compared to the values of *COMMITS* in the control group. If the difference is significant, we compute the effect size, which tells us how big the difference is. To this end, we compute *Cliff's Delta* [2]. We proceed in the same way to test H_0 1.2, with the exception that *LCHG* is our dependent variable.

3.5.2 Answering RQ 2

We answer this question in three steps. First, we investigate whether the metrics *FL*, *FC*, *CND*, and *NEG* correlate with *LOC*. To this end, we compute the Spearman rank correlation coefficients [3]. We test H_0 2.1 based on the test outcomes.

Second, we create an additional pair of experimental and control group to investigate the effect of size alone on change-proneness. To this end, we introduce a new (binary) independent variable, loc^+. For functions whose *LOC* value is above the median *LOC* of all functions, loc^+=1. These functions are in the experimental group, the others (i. e., loc^+=0) are in the control group. The dependent variables are *COMMITS* and *LCHG*, and we perform the same tests as in Section 3.5.1.

Finally, we test H_0 2.2 and H_0 2.3 by repeating the tests for RQ 1, but based on a definition of change-proneness that takes function size into account. In testing H_0 2.2, we use $COMMITS_{/LOC}$ (see Table 3) as the dependent variable, which is the number of changes to a function, divided by the function's size. In testing H_0 2.3, we use $LCHG_{/LOC}$ (also in Table 3), the number of lines changed in a function, divided by the function's size.

3.5.3 Answering RQ 3

For the previous research questions, we simplified our analysis to a binary classification, i. e., whether code is annotated or not. While this gives initial insights, it prevents us from studying the individual contribution of each independent variable when put in the context of other variables. Moreover, we cannot study the relationship between independent and dependent variables over the whole range of values. Hence, for RQ 3, we create multivariate regression models for each of our subject systems. These models handle the full range of values and reveal for each independent variable how much it contributes to change-proneness, and whether the contribution is statistically significant.

We include the metrics *FL*, *FC*, *CND*, *NEG*, $LOAC_{/LOC}$ and *LOC* in our models. *LOC* is included in log-transformed form, as $\log_2(LOC)$. The other metrics are not transformed. $LOAC_{/LOC}$ is included to investigate the influence of the amount of annotated code in a function. We chose $LOAC_{/LOC}$ over *LOAC* because *LOAC* correlates more strongly with *LOC*, potentially causing multicollinearity problems. Calculation of the *variation inflation factor (VIF)* [8] of different model variants proved $LOAC_{/LOC}$ to be less problematic.

We create regression models for each of our subject systems. Because we consider two dependent variables, *COMMITS* and *LCHG*, two models are created for each subject. As input, we use the combined data of all commit windows of a system. Again, we focus on *COMMITS* to test H_0 3.1, and on *LCHG* to test H_0 3.2.

Our dependent variables constitute count data, which we found not to be normally distributed. This ruled out many statistical methods, which assume a normal distribution. Also, a Poisson distribution cannot be used to describe our counts, as we found strong evidence of *overdispersion* [8], i. e., the variance exceeds the mean. Hence, we chose *negative binomial regression* [20], which estimates the coefficients $\beta_1, \beta_2 \ldots \beta_n$ of the formula

$$\ln(v_d) = intercept + \beta_1 v_{i1} + \beta_2 v_{i2} \ldots \beta_n v_{in} \qquad (1)$$

In this formula, v_d is the dependent variable (e. g., *COMMITS*), $v_{i1}, v_{i2} \ldots v_{in}$ are the independent variables (e. g., *FL*, $\log_2(LOC)$), and *intercept* is a constant. A coefficient describes the magnitude of the effect of an independent variable. Regression also estimates whether the effect is significant.

We report significance and effect size based on the regression estimates. Moreover, we report the McFadden statistic of each model as a measure of how well the model predicts the dependent variable [31].

4 Results

In this section, we present the results of our analysis. We mostly report aggregated values, e. g., values aggregated over all subjects systems. To characterize the average and the variation of aggregated values, we report the arithmetic mean (M) and standard deviation (SD) in the format $M \pm SD$.

4.1 RQ1: Preprocessor Usage and Change-Proneness

H_0 1.1 We show the results for preprocessor usage and change frequency in the upper part of Table 4, indicated by the dependent variable *COMMITS*. Column "Sig." refers to the number of subjects where the difference in change frequency between experimental and control group was significant at $p<0.01$. The number of insignificant results is given in parentheses. Additionally, we report the effect size in column "Cliff's Delta." The effect size d is *negligible* if $|d|<0.147$, *small*

Table 4. Effect of Individual Annotation Metrics on Changes

Independent	Dependent	Sig.[1]	Cliff's Delta	Magnitude[2]
$fl_{>0}$	*COMMITS*	8 (0)	0.27±0.11	○ ○ ◐
$fc_{>1}$	*COMMITS*	8 (0)	0.39±0.12	○ ◐ ●
$cnd_{>0}$	*COMMITS*	8 (0)	0.40±0.14	○ ◐ ●
$neg_{>0}$	*COMMITS*	8 (0)	0.32±0.12	○ ○ ◐
$fl_{>0}$	*LCHG*	8 (0)	0.27±0.11	○ ○ ◐
$fc_{>1}$	*LCHG*	8 (0)	0.39±0.12	○ ◐ ●
$cnd_{>0}$	*LCHG*	8 (0)	0.40±0.14	○ ◐ ●
$neg_{>0}$	*LCHG*	8 (0)	0.32±0.11	○ ○ ◐
loc^{+}	*COMMITS*	8 (0)	0.24±0.05	○ ○ ○
loc^{+}	*LCHG*	8 (0)	0.25±0.05	○ ○ ○
$fl_{>0}$	$COMMITS_{/LOC}$	7 (1)	0.22±0.07	○ ○ ○
$fc_{>1}$	$COMMITS_{/LOC}$	8 (0)	0.27±0.09	○ ○ ◐
$cnd_{>0}$	$COMMITS_{/LOC}$	7 (1)	0.29±0.08	○ ○ ◐
$neg_{>0}$	$COMMITS_{/LOC}$	8 (0)	0.24±0.10	– ○ ◐
$fl_{>0}$	$LCHG_{/LOC}$	8 (0)	0.22±0.10	– ○ ○
$fc_{>1}$	$LCHG_{/LOC}$	8 (0)	0.31±0.10	○ ○ ◐
$cnd_{>0}$	$LCHG_{/LOC}$	8 (0)	0.32±0.12	○ ○ ◐
$neg_{>0}$	$LCHG_{/LOC}$	8 (0)	0.27±0.10	○ ○ ◐

[1] Number of subject where the difference was significant at $p<0.01$ or not significant (in parentheses).

[2] Magnitude of d (Cliff's Delta), for $M(d)$–$SD(d)$, $M(d)$, and $M(d)$+$SD(d)$. – : negligible, ○ : small, ◐ : medium, ● : large

if $|d|<0.33$, *medium* if $|d|<0.474$, and *large* otherwise [17]. We depict the magnitude of the effect (cf. column "Magnitude") for three cases: (a) one standard deviation below the mean value of d, (b) for the mean value of d and (c) for one standard deviation above the mean value of d.

Our data reveal a medium positive correlation to *COMMITS* for $fc_{>0}$ and $cnd_{>0}$, and small positive correlations for $fl_{>1}$ and $neg_{>0}$. Out of all properties, cumulative nesting ($cnd_{>0}$) has the largest effect (d=0.4±0.14). Regarding all subjects, we observed the smallest effect, averaged over all four properties, for GLIBC (d=0.19±0.08), and the largest one for SQLITE (d=0.45±0.08). Differences in change frequency are highly significant for all eight subjects ($p<0.01$).

Based on these statistics, we conclude that preprocessor annotations have an effect on change frequency, thus, *rejecting* H_0 1.1. Instead, we accept the following, alternative hypothesis: **H_a 1.1:** *Regarding the four properties of preprocessor use defined in Section 3.2, functions exhibiting those properties are changed more frequently than other functions.*

H_0 1.2 We show the results regarding preprocessor usage and the extent of changes in the upper half of Table 4, indicated by the dependent variable *LCHG*. As for H_0 1.1 before, correlations are highly significant, with small to medium average effect sizes. The subjects with the smallest and largest overall effects are the same one as for RQ 1. They are GLIBC (d=0.2±0.08) and SQLITE (d=0.45±0.08).

Hence, we *reject* H_0 1.2,too, and accept the alternative hypothesis **H_a 1.2:** *Regarding the four properties of preprocessor use defined in Section 3.2, functions exhibiting those properties are changed more profoundly than other functions.*

4.2 RQ2: Preprocessor Usage and Function Size

As shown by previous studies, function size may also influence source code characteristics, and thus, may confound our results for RQ 1. Hence, we investigate the relation of function size and our four preprocessor properties. To this end, we initially illustrate this relation by means of Figure 2. In particular, we consider the metrics that correspond to our preprocessor properties (*FL*, *FC*, *CND*, and *NEG*) as independent variables and relate them to function size (measured in *LOC*) as the depend variable. For each independent variable, we differentiate between three metric values (0, 1 and 2 or greater), resulting in three boxplots per metric. The width of each box depends on the number of functions, with wider boxes representing more functions. Due to limited space, we show only the boxplots for OPENLDAP, though the other subjects reveal the same trends.

Our plots reveal that all four annotation metrics correlate positively with function size, indicating that functions with more CPP annotations are, on average, longer. Moreover, with increasing metric values (a) function size varies more (indicated by longer whiskers), and (b) the number of functions decreases considerably (indicated by leaner boxes).

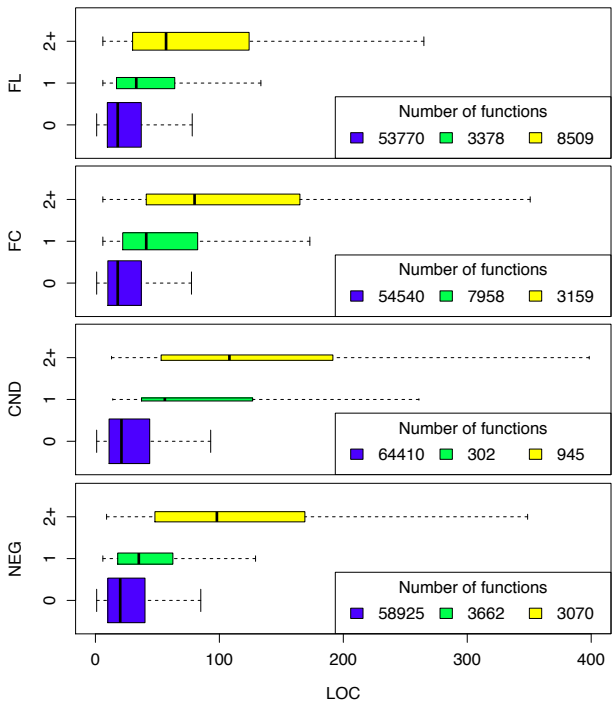

Figure 2. Size and number of functions for different values of *FL*, *FC*, *CND*, and *NEG* in OPENLDAP

In summary, we conclude that functions making heavy use of (complex) CPP annotations tend to be considerably larger. Moreover, such functions occur less frequently compared to functions that make no use of CPP annotations.

H_0 2.1 We calculated Spearman's rank correlation coefficient, r_S, to measure the relationship between annotation metrics and function size. The values of r_S range from -1 to +1, indicating strong positive or strong negative correlations, respectively. The correlation is *very weak* for $|r_S|<0.2$, *weak* for $0.2\leq|r_S|<0.4$, *moderate* for $0.4\leq|r_S|<0.6$, *strong* for $0.6\leq|r_S|<0.8$, and *very strong* for $|r_S|\geq0.8$.

We show correlation coefficients, summarized over all subjects, in Table 5. As already indicated by Figure 2, all annotation metrics correlate positively with function size, with *CND* being the least (r_s=0.14±0.04) and *FL* being the most correlated (r_s=0.34±0.07). The correlation is significant ($p\ll0.01$) for all subjects, but generally weak ($r_S<0.4$).

Based on these results, we *reject* H_0 2.1 for all four annotation metrics and accept the following alternative hypothesis: **H_a 2.1:** *Functions using preprocessor annotations to a greater extent than other functions will, on average, also be larger.*

Function size and change-proneness While H_0 2.1 confirms a relation between CPP annotations and function size, it remains open whether function size solely has an impact on change-proneness. We summarize the corresponding results in the middle part of Table 4. Our data reveals that there is a positive effect of function size on both, frequency and amount of changes. All results are significant. Although the effect is small, it varies little between subjects.

H_0 2.2 To mitigate the effect of function size on change-proneness, we normalize change frequency by function size. Our data reveal that all effects compared to H_0 1.1 have diminished, that is, the mean effect size decreases. However, the standard deviations remain similar, indicating less stability of the effects across subjects. Notably, all properties except $cnd_{>0}$ had negligible effects in GLIBC. By contrast, in OPENVPN, we observed two medium-large effects for $fc_{>1}$ and $cnd_{>0}$ (d=0.4). Two results for LIBXML2 were insignificant (for $fl_{>0}$ at p=0.61 and $cnd_{>0}$ at p=0.07).

Based on these results, we *reject* H_0 2.2 and accept the alternative hypothesis **H_a 2.2:** *Regarding the four properties of preprocessor use defined in Section 3.2, functions exhibiting those properties are changed more frequently than other functions, given a normalized change frequency.*

H_0 2.3 Similarly to H_0 2.2, we normalize the amount of changes and show the results in the last four rows of Table 4. The results are similar in that effects have diminished. The mean effects are small and range from d=0.22±0.10 (for $fl_{>0}$) to d=0.32±0.12 ($cnd_{>0}$). All results are significant.

Based on these results, we *reject* H_0 2.3 for all four properties and accept the following alternative hypothesis. **H_a 2.3:** *Regarding the properties stated in Section 3.2, functions exhibiting one of those properties are changed more profoundly than other functions, given a normalized change profundity.*

Table 5. Correlation of Annotation Metrics & Function Size

Metric	*FL*	*FC*	*CND*	*NEG*
r_S	0.34 ± 0.07	0.33 ± 0.07	0.14 ± 0.04	0.22 ± 0.06

4.3 Putting RQ 1 and RQ 2 in Relation

As detailed in the above paragraphs, our data reveal considerable differences in change-proneness between RQ 1 and RQ 2. Given the only difference is the normalization wrt. function size, our data indicate that function size has a major impact on change-proneness of annotated functions. To visualize this observation we show the corresponding measurements (i. e., number of commits and number of lines changed) for the *FL* in Figure 3 and Figure 4 for OPENLDAP.

Change frequency: In Figure 3, we show how change-frequency evolves with and without normalization. We chose commit windows as the unit on the x-axis and the average *COMMITS* per function (left side) and per LOC (i. e., number of commits to a function divided by function size; right side) as the unit of the y-axis. We computed these averages for three different values of the *FL* metric: 0, 1, and >1, indicating no, one and more than one location(s) that are annotated with `#ifdef`. The trend for each group of functions is indicated by the red, green, and blue smoothing lines.

When comparing the two plots in Figure 3, we can observe two peculiarities. First, the number of commits generally decreases with ongoing evolution, with a slight exception for the metric value *FL=1*. Second, when comparing the two charts, there are differences regarding the values for *FL*. In particular, for the left (unnormalized) chart, the number of commits correlates with the metric value, i. e., the higher the value, the higher the number of commits. However, when normalizing the change frequency by function size, this difference disappears, and thus, in the right chart, almost no differences can be observed. This observation coincides with our data in Table 4, indicating that the number of `#ifdef` directives has a rather low impact on change frequency when normalized to function size.

Extent of changes: In Figure 4, we show the evolution for the extent of changes over time. To this end, we display the average *LCHG* (number of lines changed) per function (left side) and per *LOC* (i. e., lines changed in a function divided by function size; right side) on the y-axis. The grouping, symbols, and colors are the same as in Figure 3.

The observations when comparing the two charts are similar to those, made for change frequency, that is, when normalizing *LCHG* by function size, the effect of CPP annotations dissolve. However, we also found a difference for the extent of changes: in the range of commit window 1–8, the (right)

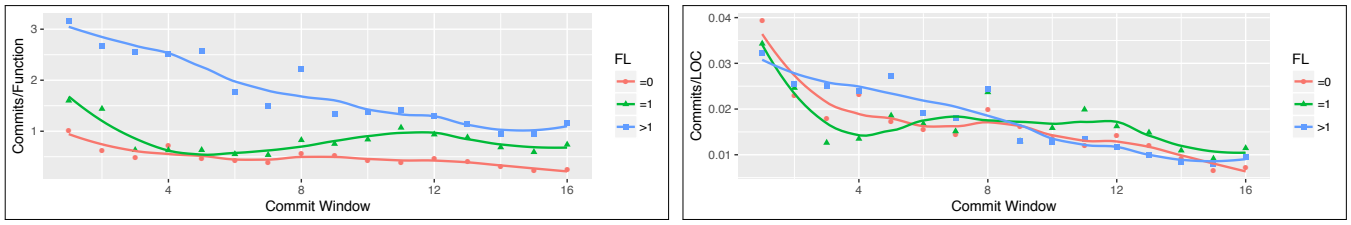

Figure 3. Raw change frequency (left side) and normalized change frequency (right side) for different values of *FL* in OpenLDAP

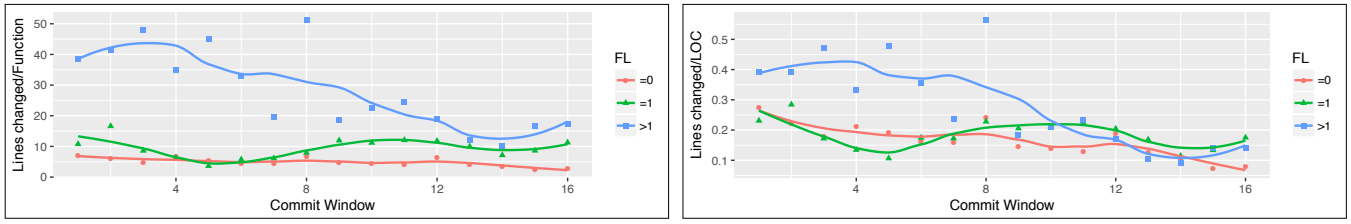

Figure 4. Raw lines changed (left side) and normalized lines changed (right side) for different values of *FL* in OpenLDAP

chart actually shows a difference, indicating an impact of the annotations (i. e., the *FL* metric) and the amount of changes, even though normalized. While we can not explain this peculiarity, it coincides with the data of Table 4, which revealed a higher effect for the *LCHG/LOG* metric than for the *COMMITS/LOC* metric. Hence, we conclude that, especially in early development stages, the number of annotations has an impact on the extent of changes, even when normalized by functions size; however, this this observation is inconsistent as it dissolves in later stages of development.

Summary: We created similar plots for the other subjects and for the other metrics (*FC*, *CND*, and *NEG*), observing the same trends as presented above. We conclude that apparent differences in change-proneness between functions with preprocessor annotations and functions without them are best explained by differences in function size. By itself, the presence of preprocessor annotations has a small, inconsistent effect on change-proneness.

4.4 RQ3: Independent Effects

The results discussed so far indicate that function size is the main driver of maintenance effort, whereas CPP annotations have only a minor, inconsistent effect. However, we still lack insights into the independent effects of our independent variables when all are considered in context. To investigate these effects, we created regression models for all independent variables, with results shown in Table 6.

Column "v_i" contains the name of an independent variable and the next three columns contain the exact results for the models created for one example subject (here: OpenLDAP). "β" is the coefficient estimate, "z" is the z-score, and p the significance level. The following columns summarize the models for all subjects by computing the averages. These averages only include coefficients and z-scores of independent variables that are significant at $p < 0.01$. If a variable was not significant for a subject, its coefficient and z-score is excluded. The last column, "Sig.", is the number of subjects where the independent variable was significant.

If the regression did not converge, we excluded the corresponding model values from the summary in the table. This was the case for LIBXML2, where neither the model for H_0 3.1, nor for H_0 3.2 converged.[3]

The coefficient estimates have the following meaning: Let β_i be the coefficient of the independent variable v_i. If the values of all other independent variables are held constant, then an increase of v_i by 1 will result in the dependent variable being multiplied by $1 + \beta$. For instance, the coefficient of *FC* is 0.06 in the *COMMITS* model of OpenLDAP. Then, for a function f in OpenLDAP, if f's *FC* increases by 1 (i. e., one more feature constant in f), we expect f's *COMMITS* to rise to 1.06 its previous value. Put differently, *COMMITS* will increase by 6 %. The Intercept summarizes the average effect of independent variables that are *not* part of our models.

The z-score is the coefficient estimate divided by its standard error. High absolute values of $|z|$ indicate that the independent variable reliably predicts the dependent variable. They also correspond to high significance levels. Conversely, low values, especially < 2, indicate unreliable predictors.

H_0 3.1 For change frequency (in OpenLDAP), our data reveal that metrics *FL* and *NEG* are not significant at $p < 0.01$.

[3] We suspect that LIBXML2's test suite causes the convergence problems. Regression *does* converge if functions and files whose name contains "test" are omitted. The test suite functions use annotations a bit more than other functions ($p \ll 0.01$, $d = 0.14 \pm 0.14$, negligible to small effects), but change less ($p \ll 0.01$, $d = -0.41$, medium effect). This goes against the trend that functions with more annotations also change (a bit) more.

Table 6. Regression Models for RQ 3

	v_i	β	z	p	β	z	Sig.
		OpenLDAP			All Systems		
COMMITS	(Intercept)	-3.48	-120.5	<0.001	-3.16±0.44	-98.4±37.2	7
	FL	-0.00	-0.1	0.939	-0.11±0.06	-5.1±01.9	4
	FC	0.06	5.7	<0.001	0.20±0.19	7.3±01.5	6
	CND	-0.06	-7.8	<0.001	-0.07±0.02	-7.0±01.7	3
	NEG	0.02	1.9	0.058	-0.00±0.14	-1.1±09.9	3
	$LOAC_{/LOC}$	0.66	13.9	<0.001	0.44±0.25	7.5±04.9	6
	$\log_2(LOC)$	0.58	104.4	<0.001	0.54±0.05	79.9±29.4	7
LCHG	(Intercept)	-2.83	-69.9	<0.001	-2.71±0.43	-59.3±21.4	7
	FL	0.00	0.3	0.729	*na*	*na*	0
	FC	-0.04	-1.8	0.071	0.23±0.20	3.2±00.3	2
	CND	-0.04	-2.9	0.003	-0.06±0.02	-3.6±01.0	2
	NEG	0.02	1.1	0.268	0.02±0.16	0.6±05.5	2
	$LOAC_{/LOC}$	0.96	10.9	<0.001	0.69±0.30	6.4±03.6	5
	$\log_2(LOC)$	0.87	101.1	<0.001	0.87±0.04	83.5±29.4	7

Consequently, neither metric is consistently correlated with change frequency. All others metrics are significant, but with small coefficients (e. g., *FC*, *CND*, *NEG*). For instance, the coefficient for *FC* is 0.06, thus, 16 additional feature constants are required for *COMMITS* to double. Given that over all subjects, functions with $FC \geq 16$ account for only 0.04 % of all functions (0.03 % for OpenLDAP), this is rather unrealistic. Moreover, we have to consider that functions rarely change. Averaged over all systems, functions receive 0.6 commits per 1000 commits (0.5 for OpenLDAP). Since our data reveal similar results for the other metrics, we conclude that the effect of our preprocessor properties on change frequency is quite small and that this observation is significant.

The coefficient of $LOAC_{/LOC}$ suggests a comparatively large effect. However, $LOAC_{/LOC}$ is a ratio, taking values from 0.0 (no annotated code) to 1.0 (all code is annotated). Thus, a conceivable increase is 0.1 or 10 %. For such an increase, *COMMITS* will rise by 6.6 %, which is, again, small.

Interestingly, *CND* has a small negative effect, meaning we can expect functions in OpenLDAP that exhibit nesting to be changed slightly *less* frequently than functions without nesting. A possible explanation might be that developers are more reluctant to change functions that exhibit nesting.

Compared to our annotation metrics, function size has a larger impact. We included size as $\log_2(LOC)$, thus, the *LOC* has to double for $\log_2(LOC)$ to increase by 1. Hence, if a function in OpenLDAP doubles in size, it will change 58 % more frequently. Our data, especially the absolute z-scores of $\log_2(LOC)$, reveal that $\log_2(LOC)$ is by far the most reliable predictor of *COMMITS*.

Considering all models together (right side of Table 6), we observe the same pattern as for OpenLDAP: Most annotation-related independent variables have small coefficients. Moreover, compared to the coefficients' mean values, the standard deviations are high, indicating that effects vary considerably between subjects. Finally, only function size ($\log_2(LOC)$) significantly affects change frequency in each subject. The small standard deviation indicates a stable effect.

We also computed the McFadden effects of our (converged) models to estimate how well they fit the data. On average, the effect is 0.088±0.020. This is a low value, because it means that the models explain less than 9 % of the observed variance in *COMMITS*. We compared the full models to models that are only based on function size, i. e., the only independent variable was $\log_2(LOC)$. Regression for the size-only models converged for all subjects, including libxml2. The McFadden effects were 0.083±0.015. Thus, in comparison, the full models perform slightly better than the size-only models, but the improvement—0.5 %—is negligible.

In summary, none of the annotation metrics are consistently correlated with change frequency. If a significant correlation exists, the effect is small. We therefore *reject* H_0 3.1 for *FC* and $LOAC_{/LOC}$ and accept the alternative hypothesis **H_a 3.1:** *Functions with a higher number of feature constants or a higher percentage of annotated lines of code are changed more frequently in some systems than other functions. The increase is likely small.* We *cannot reject* H_0 3.1 for the other annotation metrics (*FL*, *CND*, *NEG*).

H_0 3.2 For the amount of changes (lower part of Table 6, our results are even more diverse in the sense that (1) the annotation metrics for *LCHG* are even less significant than for *COMMITS*, and (2) the effect of function size is significant for only 5 of the subjects, but has increased compared to the models for *COMMITS* (cf. β=0.69±0.30).

The average McFadden effect is 0.047±0.007, which means that our models explain the extent of changes poorly. Again, we compared the full models with simple models based on function size alone. These models converged for all subjects. Their McFadden effects are 0.047 ± 0.006, indicating the same (poor) quality. Thus, taking annotation metrics into account *fails* to improve the prediction of the extent of changes.

We conclude that only $LOAC_{/LOC}$, has a significant, yet small, positive effect on the extent of changes in most, but not all systems. Except for $LOAC_{/LOC}$, the number and complexity of preprocessor annotations in a function is not consistently correlated with the extent of changes to that function. We therefore *reject* H_0 3.2 for $LOAC_{/LOC}$ in favor of the alternative hypothesis **H_a 3.2:** *Functions with a higher percentage of annotated lines of code are changed more profoundly in some systems than other functions. The increase is likely small.* We *cannot reject* H_0 3.2 for the other annotation metrics.

5 Discussion

In this section, we summarize the answers to our three research questions and relate them to each other. Moreover, we discuss the implications of our findings from a broader perspective.

CPP Annotations Have an Inconsistent Effect: Although annotation use generally correlates significantly and positively with change-proneness, effect sizes vary between subjects. During regression, we found that, when seen in context with each other and with size, few of our annotation-related metrics correlate significantly with change-proneness in the majority of subjects. Among them, $LOAC_{/LOC}$ was the most reliable predictor of change-proneness.

The Effect of Annotation Use Is Small: Where we found significant correlations between annotation use and change-proneness, the average effect size was small. The results for RQ 1 revealed medium and even large effects in at least some systems. However, the results for RQ 2 indicate that these effects occur as a corollary of annotated functions being larger. Using regression models to predict change-proneness, we found that prediction quality improves only by a negligible amount when annotation metrics were added to the models.

Size Has a Consistent Effect: Function size correlates consistently and significantly with change-proneness in all of our tests with effect sizes varying only little between subjects. This was expected, given previous findings by other researchers, and emphasizes the importance of controlling for size as a potential confounding factor.

Long Functions with Annotations May Still Be a Bad Idea: We found only small effects of annotation-usage on change-proneness. However, that does not mean that annotations should be used carelessly, especially since functions with annotations also tend to be longer (see Section 4.2). Previous work indicates that annotation usage affects other maintenance aspects, such as fault-proneness and program comprehension (e. g., [44, 32, 36, 30]). Since the effects accumulate, the combination of preprocessor annotations and long functions likely spells trouble.

Important Predictors of Change-Proneness Are Missing: Our regression models generally predicted change-proneness poorly. Hall et al. made similar observations when predicting fault-proneness based on static code metrics, which coincides with our results [19]. However, the poor predictions show that we are missing other important factors that have a more profound impact on change-proneness than preprocessor annotations. For instance, process-metrics, such as the age of the code and previous changes [38], are likely to play important roles.

6 Threats to Validity

In this section, we discuss possible threats to the validity of our findings and how we mitigated these threats.

6.1 Internal Validity

Our study can only reveal *correlations* between preprocessor usage and change-proneness. However, even where we found correlations, we cannot claim that annotations *cause* change-proneness. Our methodology is based in large part on studies of the effects of *code smells* on change- or fault-proneness (e. g., [39, 25, 19, 43]). Although we investigate different phenomena (preprocessor use, not code smells), the underlying questions are similar, and the same methods apply. Thus, we at least conform to the state of the art.

Most statistics are sensitive to the distribution of the data. Where possible, we used robust statistical tests so as not to violate any assumptions about the analyzed data. We chose negative binomial regression as our regression technique based on analyses of distribution characteristics and additional statistics on model fitness. Others used such models for similarly shaped data [40, 48, 16, 19].

We use the frequency and the amount of changes as a proxy for maintenance effort. This is common in software engineering research (e. g., [5, 6, 29, 39]), but has known limitations. For instance, preprocessor directives may hinder program comprehension, which we cannot measure with our methodology. Hence, we only claim that the use of preprocessor annotations is largely unrelated to frequency and amount of changes *as per version control information.*

Bugs in the tools we developed and in third-party tools could confound our analyses. We mitigate this threat by relying on mature tools where possible (SRC2SRCML [4], CPPSTATS [27, 21], REPODRILLER [46], and EGIT[4]). We checked for bugs in our own tools using regression tests and sample-based inspection of output data. A small number of files (<0.1 %) could not be parsed due to errors in our tool-chain. This is unlikely to skew our data to any relevant degree.

We collect data using a snapshot technique, which entails some imprecisions. For example, the static metrics of a function could change considerably in the course of a snapshot. Moreover, we had to discard some commits during snapshot creation to keep snapshots comparable. However, we carefully investigated these threats in preliminary experiments (cf. Section 3.4) and chose the snapshot size accordingly. Other studies of change-proneness use releases of a software system as snapshots (e. g., [7, 25, 19]). Since our snapshots are much shorter than typical release cycles, our analysis is at least as precise as the current state of the art. Moreover, any remaining imprecision will affect annotated as well as un-annotated functions equally. Hence, our statistics and conclusion remain valid.

6.2 External Validity

We cover many well-known aspects of preprocessor use, e. g., the number of `#ifdef` directives, nesting and negation. Nevertheless, we miss some aspects, such as annotation discipline. We cannot generalize our findings to these aspects.

Software systems differ in how they use preprocessor annotations and how changes are performed, depending on

[4] www.eclipse.org/egit/

their domain. We mitigate these threats by choosing systems that differ in size and domains. All of our subject systems are open source; no industrial systems were analyzed. Hunsen et al. showed that annotation usage is the same for open- and closed-source systems [21]. Hence our results should be generalizable to at least other systems in the same domains, both open- and closed-source.

We only consider subjects written in C and using CPP annotations. We expect our findings to be generalizable to other procedural languages and other preprocessors that implement conditional compilation similarly to the CPP.

7 Related Work

C Preprocessor Usage and Variability-Related Problems In recent years, several researchers studied empirically how CPP annotations relate to fault-proneness and code comprehension. Syntax errors caused by an incorrect use of annotations were found to be rare, but once introduced, they are particularly long-lived [33, 35, 32]. Moreover, developers perceive CPP-related bugs as easier to introduce, harder to fix and more critical than other bugs. Melo et al. showed that developers find bugs more slowly and less precisely when the amount of variability increases [36]. Ferreira et al. suggests that functions with security vulnerabilities exhibit more preprocessor annotations than non-vulnerable functions [15].

Another line of work explored the use of colors to support or replace CPP-based variability [22, 14, 26]. Specifically, highlighting CPP-annotated code with background colors helps program comprehension in some (but not all) situations [14]. A combination of background colors and virtual separation of concerns [22] was proposed as an alternative to CPP-based variability [26]. Experiments showed that this alternative improves efficiency and correctness of program comprehension compared to using plain CPP directives.

It is an ongoing debate whether *undisciplined* annotations matter with regards to the speed and precision of bug-finding. Such annotations encompass only parts of a syntactical unit, for example, a parameter in a function declaration. Our previous study suggested that discipline did not matter [44], but newer studies suggest it does [30, 34]. It also matters to developers: They prefer disciplined annotations [32, 34, 30]. We will investigate how preprocessor discipline affects change-proneness in an extension of the present paper.

These studies relate preprocessor use to program comprehension and fault-proneness. We complement this work with quantitative empirical findings on a different maintenance aspect, namely change-proneness.

C Preprocessor Usage in General Other work analyzed CPP use in highly configurable software, for instance, with respect to scattering and tangling (e. g., [12, 27, 28, 21, 41]). They do not relate CPP usage to maintenance, as we do. Nevertheless, their insights into the statistical distributions of CPP usage metrics helped us choose appropriate tests in our study, and we build on some of their tooling.

8 Conclusion

Conditional compilation using preprocessor annotations, such as the `#ifdef`s, is a common means to achieve fine-grained variability in the source code of highly configurable software. Although widely used, annotations have long been criticized for making code hard to understand and change, and more fault-prone. Recent studies analyzed this critique empirically, suggesting that annotation indeed hinder code comprehension and increases fault-proneness.

We complement these studies with a quantitative analysis of whether and how annotations relate to another important maintenance aspect, change-proneness. To this end, we collected data about the presence and complexity of annotations in individual functions and the change behavior of those functions for eight systems, written in C. Statistical analyses of the data suggest a significant, positive correlation between annotation use and change-proneness. However, the effect size is small, especially when controlling for differences in function size. Effect sizes vary widely, with medium-sized effects in some systems and negligible effects in others. Among the properties of annotation use we studied, the percentage of annotated code to un-annotated code was the one most consistently associated with increases in change-proneness. Our findings call into question the criticism that preprocessor annotations make code harder to change. Measured by the number of commits and the number of lines changed, the impact is small at best.

In future work, we will apply our methodology to investigate the relationship of annotation use to other properties of maintenance and evolution. Our first objective is to study the effect on fault-proneness, but other aspects, for instance, co-changes, are also of interest. Controlled experiments are another interesting line of future work. They would allow us to measure the effects of annotation use on maintenance effort directly instead of using changes as a proxy. Experiments may also increase our insights into other aspects, such as program comprehension.

Acknowledgments

This work was partly funded by project EXPLANT of the German Research Foundation (DFG, grant SA 465/49-1). We would like to thank Hannes Klawuhn, Reimar Schröter, and Jens Meinicke for their contributions and fruitful discussions at the early stages of this work. Additional thanks go to Sven Apel and the participants of the FOSD'16 meeting for suggestions on improving the methodology.

References

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[2] N. Cliff. *Ordinal Methods for Behavioral Data Analysis*. Erlbaum, 1996.

[3] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. 2nd ed. Erlbaum, 1988.

[4] M. L. Collard, H. H. Kagdi, and J. I. Maletic. "An XML-Based Lightweight C++ Fact Extractor". In: *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE, 2003, pp. 134–143.

[5] M. D'Ambros, A. Bacchelli, and M. Lanza. "On the Impact of Design Flaws on Software Defects". In: *Proceedings of the International Conference on Quality Software (QSIC)*. IEEE, 2010, pp. 23–31.

[6] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos. "An Empirical Investigation of an Object-Oriented Design Heuristic for Maintainability". In: *Journal of Systems and Software* 65.2 (2003), pp. 127–139.

[7] M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. "An Empirical Study of the Relationships Between Design Pattern Roles and Class Change Proneness". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2008, pp. 217–226.

[8] A. J. Dobson and A. Barnett. *An Introduction to Generalized Linear Models*. 3rd ed. CRC Press, 2008.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. "Does Code Decay? Assessing the Evidence from Change Management Data". In: *IEEE Transactions on Software Engineering (TSE)* 27.1 (Jan. 2001), pp. 1–12.

[10] K. El Emam. *A Methodology for Validating Software Product Metrics*. Tech. rep. NCR 44142. Ottawa, Ontario, Canada: National Research Council, June 2000.

[11] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". In: *IEEE Transactions on Software Engineering (TSE)* 27.7 (2001), pp. 630–650.

[12] M. D. Ernst, G. J. Badros, and D. Notkin. "An Empirical Analysis of C Preprocessor Use". In: *IEEE Transactions on Software Engineering (TSE)* 28.12 (Dec. 2002), pp. 1146–1170.

[13] J.-M. Favre. "Understanding-in-the-Large". In: *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE, 1997, pp. 29–38.

[14] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. "Do Background Colors Improve Program Comprehension in the #ifdef Hell?" In: *Empirical Software Engineering* 18.4 (Aug. 2013), pp. 699–745.

[15] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel. "Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel". In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2016, pp. 65–73.

[16] K. Gao and T. M. Khoshgoftaar. "A Comprehensive Empirical Study of Count Models for Software Fault Prediction". In: *IEEE Transactions on Reliability* 56.2 (2007), pp. 223–236.

[17] R. J. Grissom and J. J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Erlbaum, 2005.

[18] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering". In: *IEEE Transactions on Software Engineering (TSE)* 38.6 (2012), pp. 1276–1304.

[19] T. Hall, M. Zhang, D. Bowes, and Y. Sun. "Some Code Smells Have a Significant but Small Effect on Faults". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.4 (Sept. 2014), 33:1–33:39.

[20] J. M. Hilbe. *Negative Binomial Regression*. 2nd ed. Cambridge University Press, 2011.

[21] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. "Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study". In: *Empirical Software Engineering* 21.2 (2016), pp. 449–482.

[22] C. Kästner and S. Apel. "Virtual Separation of Concerns – A Second Chance for Preprocessors". In: *Journal of Object Technology* 8.6 (Sept. 2009), pp. 59–78.

[23] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[24] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc. "An Exploratory Study of the Impact of Code Smells on Software Change-Proneness". In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.

[25] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness". In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[26] D. Le, E. Walkingshaw, and M. Erwig. "#ifdef Confirmed Harmful: Promoting Understandable Software Variation". In: *Proceedings of the Symposium on Visual Languages and Human Centric Computing (VL/HCC)*. IEEE, 2011, pp. 143–150.

[27] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[28] J. Liebig, C. Kästner, and S. Apel. "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code". In: *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202.

[29] A. Lozano and M. Wermelinger. "Assessing the Effect of Clones on Changeability". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2008, pp. 227–236.

[30] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. "The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter". In: *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 297–307.

[31] D. McFadden. *Quantitative Methods for Analyzing Travel Behavior of Individuals: Some Recent Developments*. Institute of Transportation Studies, University of California, 1977.

[32] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 495–518.

[33] F. Medeiros, M. Ribeiro, and R. Gheyi. "Investigating Preprocessor-Based Syntax Errors". In: *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 75–84.

[34] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kastner, B. Ferreira, L. Carvalho, and B. Fonseca. "Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell". In: *IEEE Transactions on Software Engineering (TSE)* (2017). accepted for publication, p. 16.

[35] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. "An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers". In: *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2015, pp. 35–44.

[36] J. Melo, C. Brabrand, and A. Wąsowski. "How Does the Degree of Variability Affect Bug Finding?" In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 679–690.

[37] A. Mockus and L. G. Votta. "Identifying Reasons for Software Changes Using Historic Databases". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2000, pp. 120–130.

[38] R. Moser, W. Pedrycz, and G. Succi. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 181–190.

[39] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg. "Are All Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[40] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. "Predicting the Location and Number of Faults in Large Software Systems". In: *IEEE Transactions on Software Engineering (TSE)* 31.4 (2005), pp. 340–355.

[41] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki. "The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems". In: *Software & Systems Modeling (SOSYM)* 16.1 (2017), pp. 77–96.

[42] D. Romano and M. Pinzger. "Using Source Code Metrics to Predict Change-Prone Java Interfaces". In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2007, pp. 303–312.

[43] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. "An Empirical Study of Code Smells in JavaScript Projects". In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2017, pp. 294–305.

[44] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. "Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment". In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM. 2013, pp. 65–74.

[45] D. I. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba. "Quantifying the Effect of Code Smells on Maintenance Effort". In: *IEEE Transactions on Software Engineering (TSE)* 39.8 (2013), pp. 1144–1156.

[46] F. Z. Sokol, M. Finavaro Aniche, and M. A. Gerosa. "MetricMiner: Supporting Researchers in Mining Software Repositories". In: *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2013, pp. 142–146.

[47] H. Spencer and G. Collyer. "#ifdef Considered Harmful, or Portability Experience With C News". In: *Proceedings of the USENIX Technical Conference*. USENIX Association, 1992, pp. 185–197.

[48] G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller. "Practical Assessment of the Models for Identification of Defect-Prone Classes in Object-Oriented Commercial Systems Using Design Metrics". In: *Journal of Systems and Software* 65.1 (2003), pp. 1–12.

[49] Y. Zhou, H. Leung, and B. Xu. "Examining the Potentially Confounding Effect of Class Size on the Associations Between Object-Oriented Metrics and Change-Proneness". In: *IEEE Transactions on Software Engineering (TSE)* 35.5 (2009), pp. 607–623.