Softw Syst Model (2017) 16:77–96
DOI 10.1007/s10270-015-0483-z

THEME SECTION PAPER

The shape of feature code: an analysis of twenty C-preprocessor-based systems

Rodrigo Queiroz[1] · **Leonardo Passos**[2] · **Marco Tulio Valente**[1] · **Claus Hunsen**[3] · **Sven Apel**[3] · **Krzysztof Czarnecki**[2]

Received: 24 October 2014 / Revised: 7 June 2015 / Accepted: 11 June 2015 / Published online: 16 July 2015

Abstract Feature annotations (e.g., code fragments guarded by `#ifdef` C-preprocessor directives) control code extensions related to features. Feature annotations have long been said to be undesirable. When maintaining features that control many annotations, there is a high risk of ripple effects. Also, excessive use of feature annotations leads to code clutter, hinder program comprehension and harden maintenance. To prevent such problems, developers should monitor the use of feature annotations, for example, by setting acceptable thresholds. Interestingly, little is known about how to extract thresholds in practice, and which values are representative for feature-related metrics. To address this issue, we analyze the statistical distribution of three feature-related metrics collected from a corpus of 20 well-known and long-lived C-preprocessor-based systems from different domains. We consider three metrics: scattering degree of feature constants, tangling degree of feature expressions, and nesting depth of preprocessor annotations. Our findings show that feature scattering is highly skewed; in 14 systems (70 %), the scattering distributions match a power law, making averages and standard deviations unreliable limits. Regarding tangling and nesting, the values tend to follow a uniform distribution; although outliers exist, they have little impact on the mean, suggesting that central statistics measures are reliable thresholds for tangling and nesting. Following our findings, we then propose thresholds from our benchmark data, as a basis for further investigations.

Keywords Software families · Preprocessor · Feature-related metrics · Thresholds · Power-law distribution

Communicated by Prof. Andrzej Wąsowski and Thorsten Weyer.

✉ Rodrigo Queiroz
rodrigoqueiroz@dcc.ufmg.br

Leonardo Passos
lnrdpss@acm.org

Marco Tulio Valente
mtov@dcc.ufmg.br

Claus Hunsen
claus.hunsen@uni-passau.de

Sven Apel
apel@uni-passau.de

Krzysztof Czarnecki
kczarnec@gsd.uwaterloo.ca

[1] Federal University of Minas Gerais, Belo Horizonte, Brazil

[2] University of Waterloo, Waterloo, Canada

[3] University of Passau, Passau, Germany

1 Introduction

Feature annotations, such as *ifdefs* (`#ifdef`, `#ifndef`, `#elif`, and `#if` C-preprocessor directives), are long said to be undesirable in source code [2,3,8,18,31]. Since annotations are often spread across the entire code base, they clutter source code, hinder program comprehension, and, consequently, complicate maintenance. Feature annotations relate code fragments to corresponding features. When maintaining the features of the system, each related extension is a potential code fragment that has to be maintained, increasing the likelihood of ripple effects. Despite these drawbacks, feature annotations are widely used in practice [8,17–20,31], in particular due to limitations of existing programming languages (e.g., see the tyranny of the dominant decomposition [16,32]). In any case, annotations provide a simple way to include new features into the code base, avoiding the upfront investment of creating modules and interfaces [15]. Still, to prevent an excessive use of feature annotations, developers

should monitor their use, for example, by setting acceptable limits, or *thresholds*.

To reveal how feature annotations are used in source code, metrics quantifying properties, such as scattering, tangling, or nesting, have been proposed in the literature [19]. However, different from other standard code metrics [1,24], these feature-annotation metrics have never been studied using rigorous statistical methods. At best, researchers report averages and standard deviations over large sets of system, as done by Liebig et al. [19]. Central tendency and dispersion measures (e.g., mean and standard deviation), however, might not result in representative values. Recent work [4,21,37] suggests that some code metrics follow *heavy-tailed* distributions, often matching a power-law distribution. In such distributions, the probability that an entity measure deviates from a typical value (e.g., arithmetic mean) is not negligible. That is, a significant fraction of code entities does not follow typical metric values, making centrality and dispersion statistics unreliable.

In previous work, we analyzed the distribution of feature scattering in five C-preprocessor-based open-source software systems [28]. For each system, we measured the scattering degree of each of its features (i.e., the number of *ifdefs* that refer to each feature), checking whether the resulting empirical cumulative distribution function (CDF) follows a power law. We found that, in four out of five systems, feature scattering follows indeed a power law, so that scattering was concentrated in few features. In this paper, we extend our preliminary study to 20 well-known C-preprocessor-based systems, including small, medium, and large systems from different functional domains. In addition to scattering, we consider the related metrics tangling degree and the nesting depth of *ifdef* annotations. In this extended setting, we found that feature scattering results in highly skewed distributions. In 14 systems (70 %), these distributions matched a power law. In these cases, reporting metrics in terms of averages and standard deviations is unreliable, although commonly done so. Hence, we raise awareness that feature-scattering thresholds based on central measures are not reliable in practice. Regarding tangling degree and nesting depth, the extracted metric values tend to follow a uniform distribution in all systems, with most values equal to one. Although outliers exist, these distributions are not as skewed as the ones seen in the scattering degree metric. This result suggests that mean values for tangling degree and nesting depth are in fact robust. Based on our analysis, we propose thresholds for the metrics we studied, which are derived such that they respect the statistical distributions we have observed. These thresholds are meant to be further evaluated by researchers and practitioners.

The remainder of this paper is organized as follows: In Sect. 2, we provide background information on the C preprocessor and on power-law distributions. Section 3 presents our methodology, including the subject systems, the process and tools we used to compute the feature-related metrics, and the procedure we followed in the statistical analysis of the collected data. Section 4 presents our results, including a discussion on the statistical distributions that describe our data best. Section 5 discusses implications of our findings, in particular regarding the extraction of thresholds for feature-related metrics. Section 6 reports threats to validity, and Sect. 7 presents related work. Section 8 concludes the paper and points to future work. All datasets, R scripts, and associated tooling are publicly available on a companion website.[1]

2 Background

In this section, we provide the necessary background on the C preprocessor (Sect. 2.1) and on power-law distributions (Sect. 2.2).

2.1 The C preprocessor

The C preprocessor enriches the C language with simple meta-programming facilities, supporting the implementation of software families [2,19,25,27]. Features can be denoted with macro names, which in turn are referenced by different compilation-guard conditions. There are different types of guard conditions: `#ifdef`, `#ifndef`, `#elif`, and `#if`. For brevity, we refer to all these constructs as *ifdefs*.

Figures 1 and 2 provide examples taken from the PYTHON INTERPRETER, one of our subject systems. In file `mmapmodule.c`, developers introduce some extensions conditionally, depending on the choice of the target operating system. This conditional code is controlled by the presence or absence of certain features. Lines 438–449, for instance, depend on the presence of feature `MS_WINDOWS`, while Lines 453–464 depend on the presence of feature `UNIX`. Nested in the code of `UNIX` feature, there is further variability that depends on the support of large files. Feature `HAVE_LARGEFILE_SUPPORT` implements this nested variable behavior at Line 460. In Fig. 2, there is another use of feature `MS_WINDOWS`, also taken from PYTHON INTERPRETER. In this case, the presence of either `MS_WINDOWS` or `__CYGWIN__` enables the compilation of the guarded code (Line 428).

2.2 Power-law distributions

A distribution is said to be a power-law distribution when the probability of measuring a particular value varies inversely as a power of that value [22]. The population of towns

[1] http://rodrigoqueiroz.bitbucket.org/sosym2015.html.

Fig. 1 Feature implementation example using *ifdefs* (`mmapmodule.c` file from the PYTHON INTERPRETER)

```
434 mmap_size_method(mmap_object *self, PyObject *unused)
435 {
436     CHECK_VALID(NULL);
437 #ifdef MS_WINDOWS
438     if (self->file_handle != INVALID_HANDLE_VALUE) {
439         DWORD low,high;
440         PY_LONG_LONG size;
441         low = GetFileSize(self->file_handle, &high);
442         (...)
443         if (!high && low < LONG_MAX)
444             return PyLong_FromLong((long)low);
445         size = (((PY_LONG_LONG)high)<<32) + low;
446         return PyLong_FromLongLong(size);
447     } else {
448         return PyLong_FromSsize_t(self->size);
449     }
450 #endif /* MS_WINDOWS */
451
452 #ifdef UNIX
453     {
454         struct stat buf;
455         if (-1 == fstat(self->fd, &buf)) {
456             PyErr_SetFromErrno(PyExc_OSError);
457             return NULL;
458         }
459 #ifdef HAVE_LARGEFILE_SUPPORT
460         return PyLong_FromLongLong(buf.st_size);
461 #else
462         return PyLong_FromLong(buf.st_size);
463 #endif
464     }
465 #endif /* UNIX */
466 }
```

Fig. 2 Tangled feature implementation example using *ifdefs* (`fileio.c` from the PYTHON INTERPRETER)

```
427 #if defined(MS_WINDOWS) || defined(__CYGWIN__)
428     _setmode(self->fd, O_BINARY);
429 #endif
430 (...)
```

and cities is a classic example of this type of distribution. Figure 3 shows a histogram with the distribution of the US city populations, extracted from the 2000 census,[2] as analyzed by Clauset et. al. [5] and Newman [22]. The histogram is highly right-skewed: While the bulk of the distribution refers to small-sized cities, a small number of very large cities produces the heavy tail to the right of the histogram.

In formal terms, a discrete power-law distribution (which we refer henceforth as power-law) is a distribution in which the probability that a discrete random variable X assumes a value x is proportional to x raised to the negative power of a positive constant k:

$$P(X = x) \propto cx^{-k} \quad \text{where} \quad c > 0, k > 0 \tag{1}$$

[2] Data available at http://tuvalu.santafe.edu/~aaronc/powerlaws/data.htm.

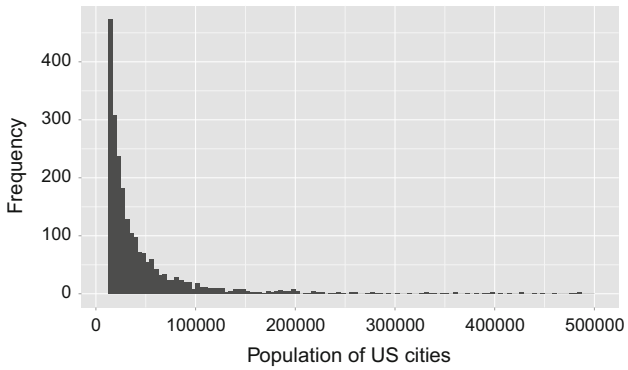

Fig. 3 Histogram of the population of US cities with population of 10,000 or more (data from the 2000 US Census)

A power law, as given by this equation, diverges when $x = 0$. In fact, it requires a lower-bound value $x_{\min} > 0$ to define a cutoff value as starting point ($x > x_{\min}$) from which a power-law behavior occurs [5]. As we shall see later

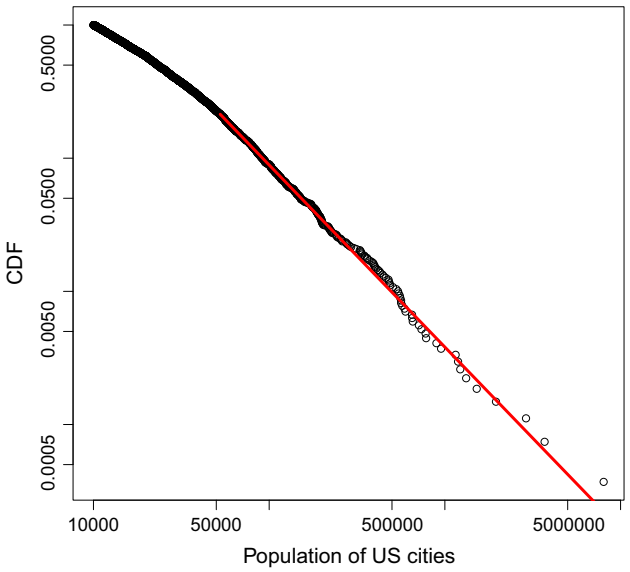

Fig. 4 Empirical cumulative density function of the populations of US cities (points) and the fitted power-law function in *red*, both in logarithmic scale (data from the 2000 US Census) (color figure online)

in Sect. 3, the parameters k and $x_{\min}$ play an important role when performing a goodness-of-fit analysis. Another important characteristic of a power-law distribution concerns its visualization: When plotted on a logarithmic scale in both axes, a power-law function appears as a decreasing line, as shown in Fig. 4.

In addition to city populations, power-law distributions appear in a diverse range of phenomena, including the size of earthquakes, solar flares, moon craters, wars, and people's personal fortunes [5,22]. In software engineering, different researchers report that the distribution of different source code metrics follows a power-law distribution [4,21,24,37]. However, it has not been investigated whether the same holds for feature-related metrics.

3 Methodology

In this section, we discuss the selection of subject systems (Sect. 3.1), the process and tools to compute feature-related metrics (Sect. 3.2), and the statistical analysis of the data we collect (Sect. 3.3).

3.1 Selection of subject systems

To analyze the statistical distributions describing feature-related metrics, we selected 20 open-source software systems that use C preprocessor directives to annotate feature code; Table 1 provides information on all subject systems.

Three criteria guided the selection of our subjects: First, we aimed at covering multiple application domains, therefore avoiding bias toward an specific domain. In Table 1, the 20 systems are distributed across 12 different domains. Second,

Table 1 Subject systems

System	Version	Year	Domain	Since	SLOC
VI[a]	50325	2005	Text editor	2000	22,275
LIGHTTPD	1.4.35	2014	Web server	2003	39,991
XFIG	3.2.5c	2013	Graphics editor	1985	74,713
SENDMAIL	8.14.9	2014	Network service	1983	92,204
SYLPHEED	3.4.2	2014	E-mail client	2000	116,454
GIT	2.1.0	2014	Version control	2005	152,018
APACHE	2.4.10	2014	Web server	1995	155,846
LIBXML2	2.9.1	2014	Programming library	1999	222,009
EMACS	24.3	2013	Text editor	1985	249,932
OPENLDAP	2.4.39	2014	Network service	1998	291,781
SUBVERSION	1.8.10	2014	Version control	2000	328,878
IMAGEMAGICK	6.8.9-7	2014	Image editor	1987	333,048
PYTHON	3.4.1	2014	Program interpreter	1989	353,485
PHP	5.6.0	2014	Program interpreter	1985	664,259
POSTGRESQL	9.3.5	2014	Database system	1995	676,435
GIMP	2.8.14	2014	Image editor	1996	703,435
GLIBC	2.20	2014	Programming library	1987	826,502
MYSQL	5.6.19	2014	Database system	1995	1,577,874
GCC	4.9.0	2014	Compiler framework	1987	3,209,684
LINUX KERNEL	3.15	2014	Operating system	1991	11,964,075

[a] The VI system we use is a port of the original VI (late 1970s) to modern Unix systems

each system has substantial history of development and use, as given by columns *Year* (the year of the release of the system under analysis) and *Since* (the year of the first release). The rationale is that mature systems are more likely to have found a practical balance to when and how much to scatter, tangle, and nest preprocessor annotations than immature systems. Third, the selection includes systems of different sizes, to avoid bias toward a particular system size. We measured size using Source Lines of Code (SLOC), which is the total number of source lines of code of a given system. These numbers excludes blank lines and comments. Moreover, sequences of multilines (lines ending with a backslash) are counted as a single line.[3] As shown in Table 1, our set of subjects includes four small systems (<100 KSLOC), nine medium-sized systems (100–400 KSLOC), and seven large software systems (>400 KSLOC).

3.2 Data collection and metrics

To extract feature-related metric values, we first parse the source files (C implementation and header files) of each subject system. Parsing is performed using the tool SRC2SRCML,[4] which creates an XML representation of the code. The resulting XML files preserve all the code, including preprocessor annotations (SRC2SRCML does not perform any preprocessing). With all annotations in place, we run a custom-made tool (FSCAT) to process the XML files produced by SRC2SRCML and to compute the following system-level metrics:

1. Number of feature constants (NOFC): The total number of macro names that are referred in, at least, one *ifdef*.
2. Number of feature expressions (NOFE): The total number of conditional expressions used in *ifdef* directives to control the inclusion or exclusion of variable code.
3. Number of top-level branches (NOTLB): The total number of top-level *ifdef* branches, including `#else` preprocessor directives. An *ifdef* branch is a block of code that is delimited by `#ifdef`, `#ifndef`, `#if`, or `#else` and closed by its `#endif` or followed by a `#else` or `#elif` (when applicable). A top-level *ifdef* branch is an *ifdef* or `#else` that is not inside an enclosing *ifdef* branch.

In addition, FSCAT also computes metrics for each feature constant, feature expression, and top-level *ifdef* branch of the system under analysis, as follows:

1. Scattering degree (SD): This degree is calculated per feature constant of a system. It counts the number of *ifdefs* that refer to a given feature constant. For example, considering the examples in Figs. 1 and 2, the scattering degree of `MS_WINDOWS` is 2.
2. Tangling degree (TD): This degree is calculated per feature expression of a system. It counts the number of feature constants that occur in a given feature expression. For example, in Fig. 2, the tangling degree of the feature expression in Line 427 is 2.
3. Nesting depth (ND): This metric is defined for each top-level *ifdef* branch in a system. ND is the depth of the tree of nested annotations in a given top-level *ifdef* branch. In this tree, the nodes are *ifdefs* and `#else` annotations, while the edges represent nested relations between such nodes. The root node is a top-level *ifdef* or `#else`, and the children of a node are the annotations it directly encloses. The depth of a node is the depth of its parent plus 1. By definition, the depth of the root node is one. The depth of the tree is the depth of its node with the maximal depth. For example, in Fig. 1, the ND of the top-level `#ifdef` in Line 437 is 1, and the ND of the top-level `#ifdef` in Line 452 is 2.

The proposed metrics are based on the metrics of Liebig et al. [19]. SD is based on the number of maintenance program locations that one potentially has to consider to maintain a feature and has been already used in other studies [13,14,19,26]. Likewise, our definition of TD is also based on Liebig et al. However, the tool used by Liebig et al. for computing SD and TD (CPPSTATS) applies transformations on the annotations of the code (e.g., by propagating the condition of outer *ifdefs* to inner ones and conjoining each `#elif`/`#else` condition with the condition of preceding branches). These transformations influence the scattering and tangling values, causing higher values. In contrast, the tooling we use when collecting these metrics (FSCAT) does not perform any transformation on annotations, taking them as explicitly defined in the source code. As an example, Fig. 5 shows two fragments of the same code. In the original code (a), as computed by FSCAT, FEATURE_A has SD = 1 and the feature expression at line 3 has TD = 1. In the fragment (b), we show the code after the transformations performed by CPPSTATS. In this case, CPPSTATS returns that FEATURE_A has SD = 3 and that the feature expression at line 3 has TD = 2.

The third metric, however, differs from the ones proposed by Liebig et al. Originally, Liebig et al. define the average nesting depth (AND) as the average depth of nested *ifdefs*. Instead, we propose the metric nesting depth (ND) representing the depth of each top-level *ifdef* branch. We argue that ND is more robust than AND, because it is not based on averages. Furthermore, while performing maintenance tasks,

[3] Multilines are convenient when spanning a long line across multiple ones; during compilation, sequences of multilines are taken as a single line.

[4] http://www.srcml.org/.

Fig. 5 Example of code, as considered by FSCAT (**a**) and after the transformations performed by CPPSTATS (**b**)

(a) original code

```
1 #ifdef FEATURE_A
2     (...)
3 #ifdef FEATURE_B
4     (...)
5 #endif
6 #elif FEATURE_C
7     (...)
8 #endif
```

(b) code with transformations

```
1 #ifdef FEATURE_A
2     (...)
3 #ifdef FEATURE_A && FEATURE_B
4     (...)
5 #endif
6 #elif !FEATURE_A && FEATURE_C
7     (...)
8 #endif
```

a programmer has to be aware of inner *ifdefs* when reasoning about the code. ND supports the developer to estimate the complexity of the code fragment inside a top-level branch. The AND metric, however, gives only a rough estimation of the complexity of the whole file. As an example, in Fig. 5, AND is the average nesting depth of each *ifdef* block in this file (AND = $(1 + 2)/2 = 1.5$). In contrast, ND captures the nesting of each top-level branch, e.g., ND of the top-level *ifdef* branch at lines 1–5 is 2, and the ND of the top-level *ifdef* branch at lines 6–8 is 1.

3.3 Statistical analysis

After collecting the metrics, we inspect the histograms and standard descriptive statistics describing the distributions of SD, TD, and ND for each of our 20 subject systems. In addition, we computed the Gini coefficient to measure the degree of concentration of the metric values inside each distribution. The Gini coefficient has been proposed as an economic indicator to measure and compare income distributions, but can be adapted to the distribution of software metrics, providing an aggregated metric that is system size independent [29,35,36].

In this initial analysis step, we check whether the collected distributions have characteristics of a power-law distribution. Then, we proceed with a rigorous test of the power-law hypothesis. Basically, to decide whether the metric values follow a power-law distribution, we used the POWERLAW package [10] from the R statistical environment.[5] First, we define the two parameters k and $x_{\min}$ of the best-fit power law ($\mathbb{P}_0$) that approximates as close as possible the empirical CDF (cumulative distribution function) of the distribution ($\mathbb{P}$) of a system under analysis. When searching for $\mathbb{P}_0$, we rely on the maximum-likelihood estimator method (MLE), while choosing $x_{\min}$ as the value minimizing the Kolmogorov Statistic (KS). The KS is given by the maximum distance $|\mathbb{P}_0(x) - \mathbb{P}(x)|$, for all $x > x_{\min}$. For further details, the reader is referred to elsewhere [5,10,11].

Once we estimated k and x_{min}, we perform a hypothesis test to verify whether a power-law distribution is a plausible model for the behavior of each empirical CDF. Following Clauset et al. [5], we perform a goodness-of-fit test via a bootstrap procedure following the algorithms and steps outlined by Gillespie [11]. Simply put, we generate 2500 datasets from the $\mathbb{P}_0$ model and then try to re-infer a new best-fit power law for each generated dataset. The p value of the simulation process corresponds to the fraction of times in which the KS of the best-fit model of the generated dataset is higher than the one obtained for $\mathbb{P}$. Our hypotheses are the following:

- Null hypothesis: $\mathbb{P}_0$ is a plausible fit for $\mathbb{P}$
- Alternative hypothesis: $\mathbb{P}_0$ is not a plausible fit for $\mathbb{P}$

As Clauset et al. argue, if the test reports a p value larger than 0.1, one shall accept the null hypothesis. Otherwise, it should be rejected in favor of the alternative hypothesis. In the latter case, $\mathbb{P}$ is unlikely to conform to a power-law distribution; rather, the empirical CDF function better fits another model and may or may not be heavy-tailed.

For each metric of each subject system, the fit is initially done based on the whole set of metric data. However, when we do not have strong evidence that the data fit a power law (p value < 0.1), we verify whether it is possible to fit, at least, part of the data, which is valid as we want to prove that the distribution follows a power law, asymptotically. To this end, we iteratively crop single data points from the right of the tail, removing at each step the highest metric value. The iteration continues until we find a fit up to a certain value $x_{\max}$. The procedure stops if we remove 1 % of the data points and we still do not find strong evidence for a power law.

A similar procedure has been reported by Baxter et al. [4], when analyzing standard metrics for Java software.

4 Results

In this section, we describe the results of our distribution fitting analysis for the three metrics: scattering degree (Sect. 4.1), tangling degree (Sect. 4.2), and nesting depth (Sect. 4.3).

4.1 Scattering degree

To assess the distribution of the feature-scattering degrees, we initially plotted the histograms of the extracted SD values for each system, shown in (Fig. 6). We can observe that

[5] http://www.r-project.org/.

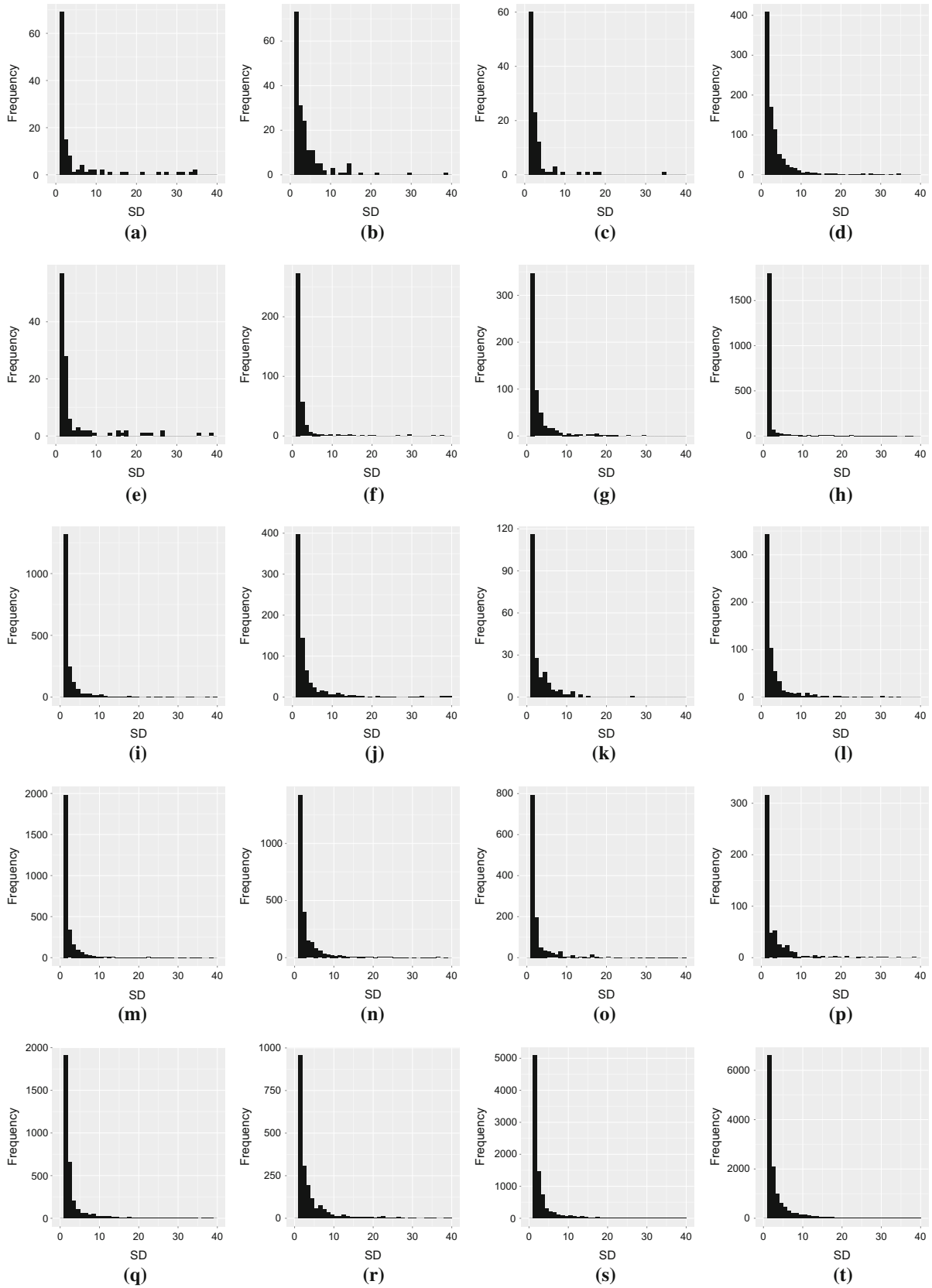

Fig. 6 Histograms of scattering degrees (SD). **a** VI, **b** LIGHTTPD, **c** XFIG, **d** SENDMAIL, **e** SYLPHEED, **f** GIT, **g** APACHE, **h** LIBXML2, **i** EMACS, **j** OPENLDAP, **k** SUBVERSION, **l** IMAGEMAGICK, **m** PYTHON, **n** PHP, **o** POSTGRESQL, **p** GIMP, **q** GLIBC, **r** MYSQL, **s** GCC, **t** LINUX KERNEL

Table 2 Scattering degree (SD) descriptive measures (NOFC: Number of Feature Constants)

System	NOFC	Mean	Median	95th	Max	Mode	%	Gini
VI	118	4.72	1	27.45	53	1	58.4	0.66
LIGHTTPD	179	4.20	2	14.00	48	1	40.7	0.57
XFIG	110	3.87	1	14.10	83	1	54.5	0.64
SENDMAIL	905	3.91	2	11.00	204	1	45.0	0.59
SYLPHEED	121	8.90	2	35.00	242	1	47.1	0.77
GIT	383	2.62	1	7.90	92	1	71.2	0.56
APACHE	606	3.30	1	12.00	114	1	57.2	0.58
LIBXML2	2,095	4.14	1	13.00	379	1	86.0	0.73
EMACS	1,970	3.50	1	9.00	211	1	66.8	0.64
OPENLDAP	784	4.10	1	14.00	85	1	50.6	0.62
SUBVERSION	217	5.63	1	11.00	339	1	53.4	0.72
IMAGEMAGICK	636	5.46	1	13.00	429	1	53.9	0.72
PYTHON	2,849	2.71	1	7.00	322	1	69.5	0.56
PHP	2,502	4.37	1	12.00	674	1	56.9	0.67
POSTGRESQL	1,264	4.49	1	13.85	569	1	62.7	0.70
GIMP	557	4.66	1	18.00	156	1	56.7	0.66
GLIBC	3,370	4.94	1	14.00	662	1	56.5	0.71
MYSQL	1,990	6.93	2	18.00	652	1	47.9	0.75
GCC	8,898	4.38	1	13.00	1,867	1	57.1	0.68
LINUX KERNEL	12,661	5.40	1	14.00	2,698	1	52.1	0.71

all histograms are right-skewed, meaning that, while most SD values are small (equal to one, in most cases), we also observe high and very high SD values, suggesting a heavy-tailed—possibly a power-law—distribution. Table 2 shows the maximum SD for each system, along with its mean and 95th percentile value. The scattering degree reaches extreme values in GCC (max SD = 1867) and in the LINUX KERNEL (max SD = 2698), while other systems have lower degrees, for example, VI (max SD = 53) and LIGHTTPD (max SD = 48). Furthermore, the mean SD value is, in all cases, too far apart from the values in the last 5 % of the features. For example, in the LINUX KERNEL, the mean SD is 5.40, and the 95th percentile of SD is 14. This nicely illustrates that the mean should not be used as a reference or centrality measure when analyzing SD. For example, in the LINUX KERNEL, the mean is not only very different from the small SD values (equal to 1) that represent the bulk of the distribution, but it also does not represent the high SD values in the right part of the histogram ($\geq$5.4). Therefore, it is fundamental to know the statistical distribution describing the collected SD data, before investigating reference values, thresholds, and similar quantitative guidelines for this metric.

Finally, as presented in Table 2, the SD distribution's Gini coefficients range from 0.56 (GIT) to 0.77 (SYLPHEED), which suggests a high level of inequality inside each distribution. All aforementioned characteristics are necessary, yet not sufficient, to claim that feature scattering follows a power-law distribution.

Following the methodology described in Sect. 3.3, as a next step, we estimate the parameters of the power-law model (k and $x_{\min}$) that best fit our empirical CDF. In Fig. 7, we plot the empirical CDF and the fitted power law in logarithmic scale, of which the latter appears as a red decreasing line in each graph of figure. As we stated in Sect. 2.2, the resulting line is a distinct characteristic of power-law distributions. The resulting plot reveals that the points approximate the line of the power law, strengthening our understanding that feature scattering indeed follows a power-law distribution.

To statistically check whether the fitted power laws are plausible models (null hypothesis), we perform a bootstrapping hypothesis test, following the methodology described in Sect. 3.3. Table 3 shows the p values obtained for each test. In the case of 14 systems, we found statistically significant models (p values > 0.1), leading us to accept the null hypothesis—the power-law model is a plausible explanation model. Similar to the estimation of $x_{\min}$, in the case of three systems (SUBVERSION, GIMP, and MYSQL), the best fit to a power law requires an upper- bound value ($x_{\max}$). This cropping of a small number of features ($<$0.5 %) in the end of the distribution is necessary when the power-law behavior seems to fit most of the distribution, but does not hold for some few higher values. For the remaining six systems, we reject the null hypothesis in favor of the alternative hypothesis (the power-law model is not a plausible explanation model). Note that this is not the same as concluding that the scattering distribution of these six systems is not heavy-tailed.

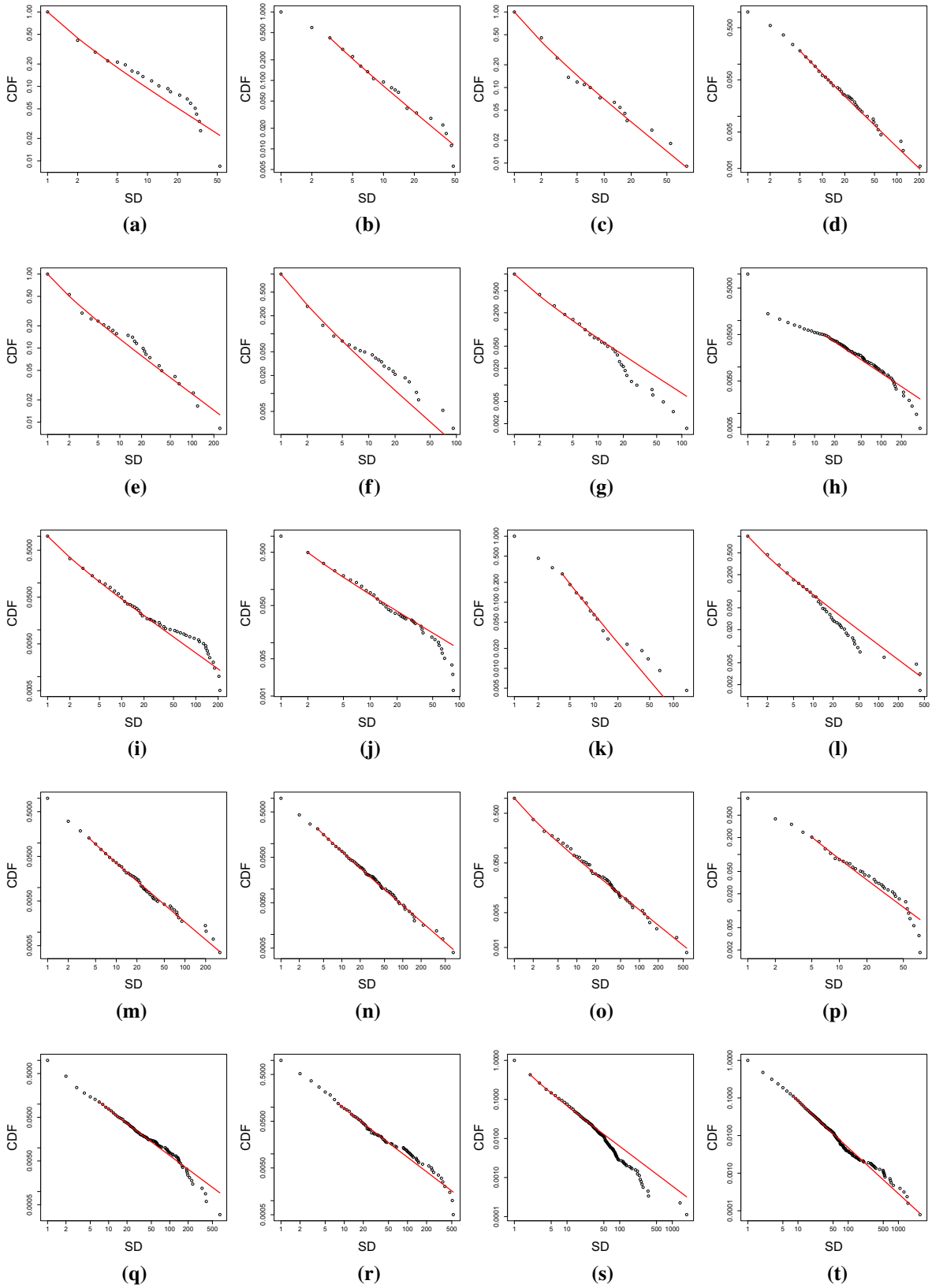

Fig. 7 Empirical CDFs of the scattering degrees (points) and the fitted power law (*red line*), both in logarithmic scale. **a** VI, **b** LIGHTTPD, **c** XFIG, **d** SENDMAIL, **e** SYLPHEED, **f** GIT, **g** APACHE, **h** LIBXML2, **i** EMACS, **j** OPENLDAP, **k** SUBVERSION, **l** IMAGEMAGICK, **m** PYTHON, **n** PHP, **o** POSTGRESQL, **p** GIMP, **q** GLIBC, **r** MYSQL, **s** GCC, **t** LINUX KERNEL (color figure online)

Table 3 Power-law best-fit analysis for scattering degree (SD)

System	k	x_{min}	x_{max}	% crop	p value
VI	1.8542	1	53	0	**0.2692**
LIGHTTPD	2.2239	3	48	0	**0.7928**
XFIG	1.9645	1	83	0	**0.1388**
SENDMAIL	2.3729	5	204	0	**0.7392**
SYLPHEED	1.7286	1	242	0	**0.1728**
GIT	2.3003	1	92	0	**0.1592**
APACHE	1.9704	1	114	0	0.0632
LIBXML2	1.9535	14	379	0	**0.1880**
EMACS	2.1288	1	211	0	0.0020
OPENLDAP	2.0032	2	85	0	0.0392
SUBVERSION	2.4064	4	146	0.46	**0.4984**
IMAGEMAGICK	1.8915	1	429	0	0.0220
PYTHON	2.2993	4	322	0	**0.7960**
PHP	2.1652	4	674	0	**0.9636**
POSTGRESQL	2.0145	1	569	0	0.0292
GIMP	2.1997	5	76	0.35	**0.1272**
GLIBC	2.0358	7	662	0	**0.7840**
MYSQL	2.0255	8	528	0.05	**0.1128**
GCC	2.0146	2	1867	0	0.0000
LINUX KERNEL	2.2216	8	2698	0	**0.5516**

Significant results (p value > 0.1) are bold

In fact, Fig. 7 suggests a heavy-tailed distribution for all 20 systems, some of which possibly follow an alternative distribution (e.g., stretched exponential or log-normal). Table 3 lists the inferred parameters and the resulting p values.

4.2 Tangling degree

To analyze the distribution of tangling degrees, we also plotted the histograms of the extracted TD values for each system, as shown in Fig. 8. We observe that the histograms follow a different pattern from the ones for scattering (Fig. 6). In particular, most TD values are equal to one, and we have very few feature expressions with higher TD values. Table 4 shows the number of feature expressions (NOFE) in our subject systems, as well as the mean, median, 95th percentile, maximal value, the mode value, and its frequency in the TD distributions of each system. The table provides also the Gini coefficients computed over the TD values of a system.

First of all, we can observe that, in all systems, the mean is close to one and both the median and the mode are equal to one. Second, the 95th percentile is less or equal to two in 19 systems (only in GCC, it is equal to three). In 13 systems, the mode represents, at least, 90 % of the measured values. For example, 99.2 % of the feature expressions in VI have a TD value equal to one. IMAGEMAGICK is the system with the lowest frequency of TD values that are equal to one (76.3 %). Finally, the Gini coefficients for TD—across all subjects—are less than 0.21, which shows that tangled degree is nearly equally distributed. For all systems, we found that a power-law distribution is not a plausible model for the reported TD values. Although it is possible to fit a power law to the tail of each TD distribution, the number of data points in the right of the tail is too small to claim statistical power.

4.3 Nesting depth

As shown in Fig. 9, the histograms of the ND values for each system are similar to the ones for tangling: Most ND values are equal to one, and we have very few branches with nested *ifdefs* and `#else` annotations. Table 5 shows the number of top-level branches (NOTLB) in the subject systems, as well as the mean, median, 95th percentile, maximal value, mode value, and its frequency in the considered ND distributions.

The table provides also the Gini coefficients computed over the ND values extracted for each system. Similarly to TD, the mode is one in all systems and it represents, at least, 86 % of the measured ND values. For example, 94.7 % of the *ifdef* branches in APACHE have ND values equal to one, and GIT is the system with the lowest number of branches without nested *ifdefs* (86.2 %). Much like for TD, the Gini coefficients are quite low, <0.15, suggesting a uniform distribution, even more than the ones observed for tangling. As a consequence, we found that a power-law distribution is not a plausible model for the reported ND values. As we concluded for TD, although it is possible to fit a power law to the tail of each ND distribution, the number of data points in the right of the tail is too small to claim statistical power.

5 Thresholds for feature-related metrics

Our empirical study shows that feature scattering is highly skewed. In fact, 14 out of 20 systems show strong evidence that feature scattering is heavy-tailed, and the underlying distributions follow a power law. Tangling and nesting have a more uniform distribution for all subject systems, with most values equal to one. These findings are of great relevance for the extraction of thresholds for the studied metrics. Most importantly, feature-scattering thresholds should not be based on centrality statistic measures (e.g., mean and standard deviation). In contrast, tangling and nesting have only a small number of outliers, which are not too far apart from the most typical values of these two metrics. Thus, mean values are robust thresholds for the tangling and nesting metrics.

Based on our statistical analysis, we derive thresholds for feature scattering, tangling, and nesting in a way that respects the distributions found in our study. Taking data skew into account, we rely on the notion of *relative thresholds*, which we explain next.

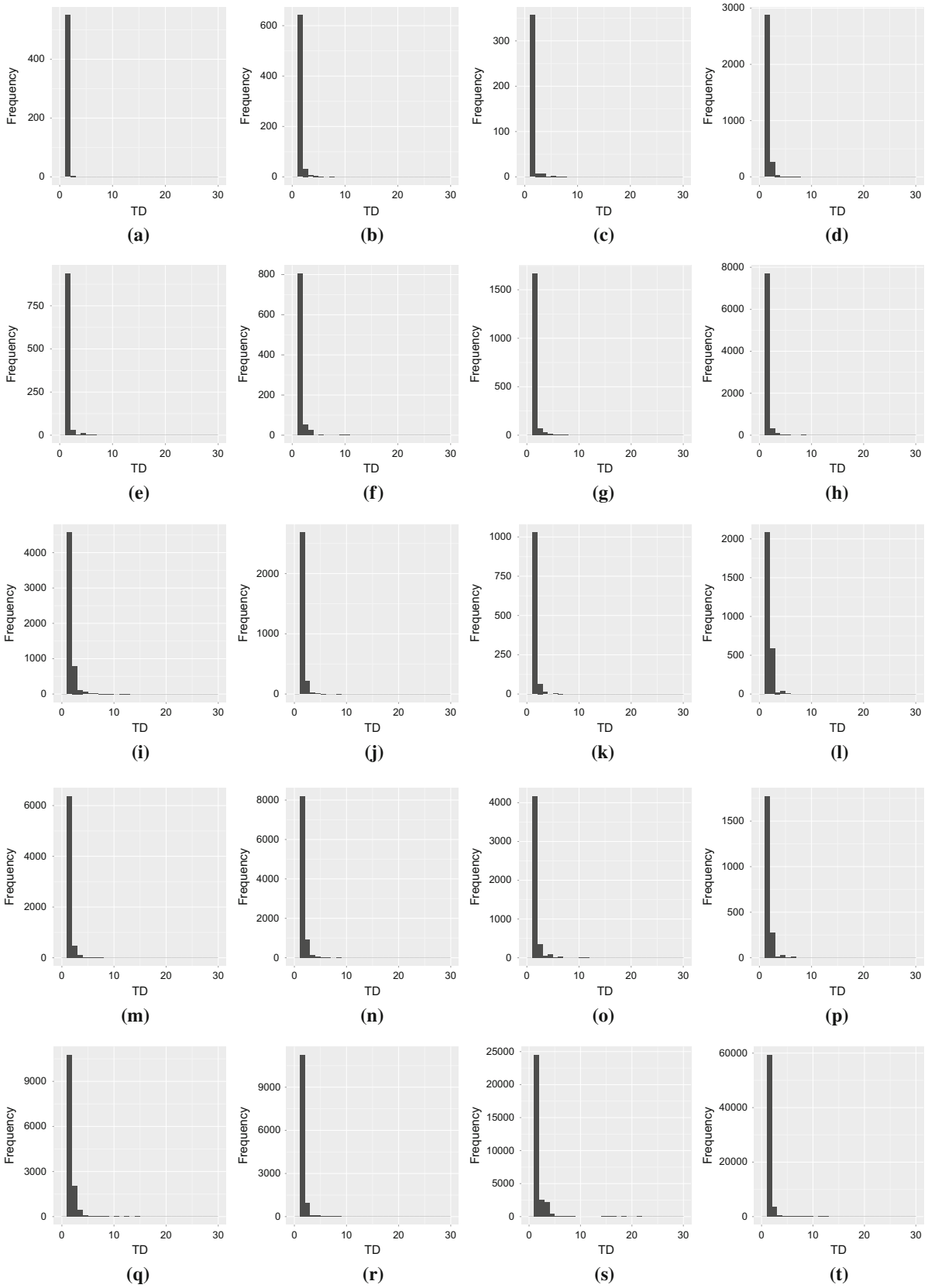

Fig. 8 Histogram of tangling degrees (TD). **a** VI, **b** LIGHTTPD, **c** XFIG, **d** SENDMAIL, **e** SYLPHEED, **f** GIT, **g** APACHE, **h** LIBXML2, **i** EMACS, **j** OPENLDAP, **k** SUBVERSION, **l** IMAGEMAGICK, **m** PYTHON, **n** PHP, **o** POSTGRESQL, **p** GIMP, **q** GLIBC, **r** MYSQL, **s** GCC, **t** LINUX KERNEL

Table 4 Tangling degree (TD) descriptive measures (NOFE: number of feature expressions)

System	NOFE	Mean	Median	95th	Max	Mode	%	Gini
VI	554	1.00	1	1	2	1	99.2	0.01
LIGHTTPD	686	1.09	1	2	7	1	93.5	0.08
XFIG	378	1.12	1	2	7	1	94.4	0.10
SENDMAIL	3176	1.11	1	2	7	1	90.5	0.09
SYLPHEED	986	1.09	1	1	6	1	95.1	0.08
GIT	885	1.13	1	2	10	1	91.0	0.11
APACHE	1,788	1.12	1	2	7	1	93.3	0.10
LIBXML2	8,127	1.06	1	2	8	1	94.8	0.06
EMACS	5,565	1.23	1	2	12	1	82.2	0.16
OPENLDAP	2,930	1.09	1	2	8	1	91.7	0.08
SUBVERSION	1,113	1.09	1	2	6	1	92.5	0.08
IMAGEMAGICK	2,732	1.27	1	2	5	1	76.3	0.16
PYTHON	6,969	1.11	1	2	7	1	91.5	0.09
PHP	9,373	1.16	1	2	8	1	87.5	0.12
POSTGRESQL	4,717	1.20	1	2	11	1	88.5	0.15
GIMP	2,118	1.22	1	2	6	1	83.8	0.16
GLIBC	13,345	1.24	1	2	14	1	80.6	0.16
MYSQL	12,359	1.11	1	2	8	1	91.0	0.09
GCC	29,842	1.30	1	3	21	1	82.1	0.20
LINUX KERNEL	63,482	1.07	1	2	12	1	93.5	0.06

5.1 Relative thresholds

Several code metrics, measuring properties such as size, coupling, and cohesion are well known following heavy-tailed distributions [4,21,37]. For this reason, previous work proposed techniques to extract thresholds that do not rely on the mean or the standard deviation. For example, Oliveira et al. [24] proposed the notion of relative thresholds for evaluating heavy-tailed metric values, along with a set of functions that obtain such thresholds from a set of subject systems (*Corpus*). Relative thresholds have the following format:

at least $p\%$ *of the entities should have* $M \leq k$

where M is a metric calculated for a given source code entity, k is an upper limit, and p is the minimal percentage of entities that should be below this upper limit. The goal is to establish upper limits for metric values that should be followed by most entities, not necessarily all, though. The reason is that, in heavy-tailed distributions, the high metric values of the distribution make it challenging to define thresholds for all entities. Thus, relative thresholds attempt to balance two forces: (i) on one hand, relative thresholds should reflect real design rules, followed by most subjects in the target system; (ii) on the other hand, the prescribed thresholds should not be based on lenient upper limits. For example, a threshold stating that "95 % of the feature constants in a system should have a scattering degree less than 3K" is probably satisfied by most systems.

Figure 10 presents the functions introduced by Oliveira et al. to calculate the parameters p and k that define the relative threshold for a given metric M. First, function $ComplianceRate[p, k]$ returns the percentage of systems in the *Corpus* that follows the relative threshold defined by the pair $[p, k]$. To determine the best p and k, *ComplianceRate* is maximized, while accounting for a minimal *CompliancePenalty*. The latter is the sum of penalties introduced by two functions:

- $penalty_1[p, k]$: a $ComplianceRate[p, k] < 90\,\%$ receives a penalty proportional to its distance to 90 %, This penalty fosters thresholds followed by, at least, 90 % of the systems in the *Corpus*.
- $penalty_2[k]$: a $ComplianceRate[p, k]$ receives the second penalty proportional to the distance between k and the median of the 90th percentiles of the values of M in each system in the *Corpus*, denoted by *Median90*.

5.2 Thresholds for scattering degree

As we found that feature-scattering degrees follow a heavy-tailed distribution, we computed relative thresholds for this metric, using our sample of 20 systems and the compli-

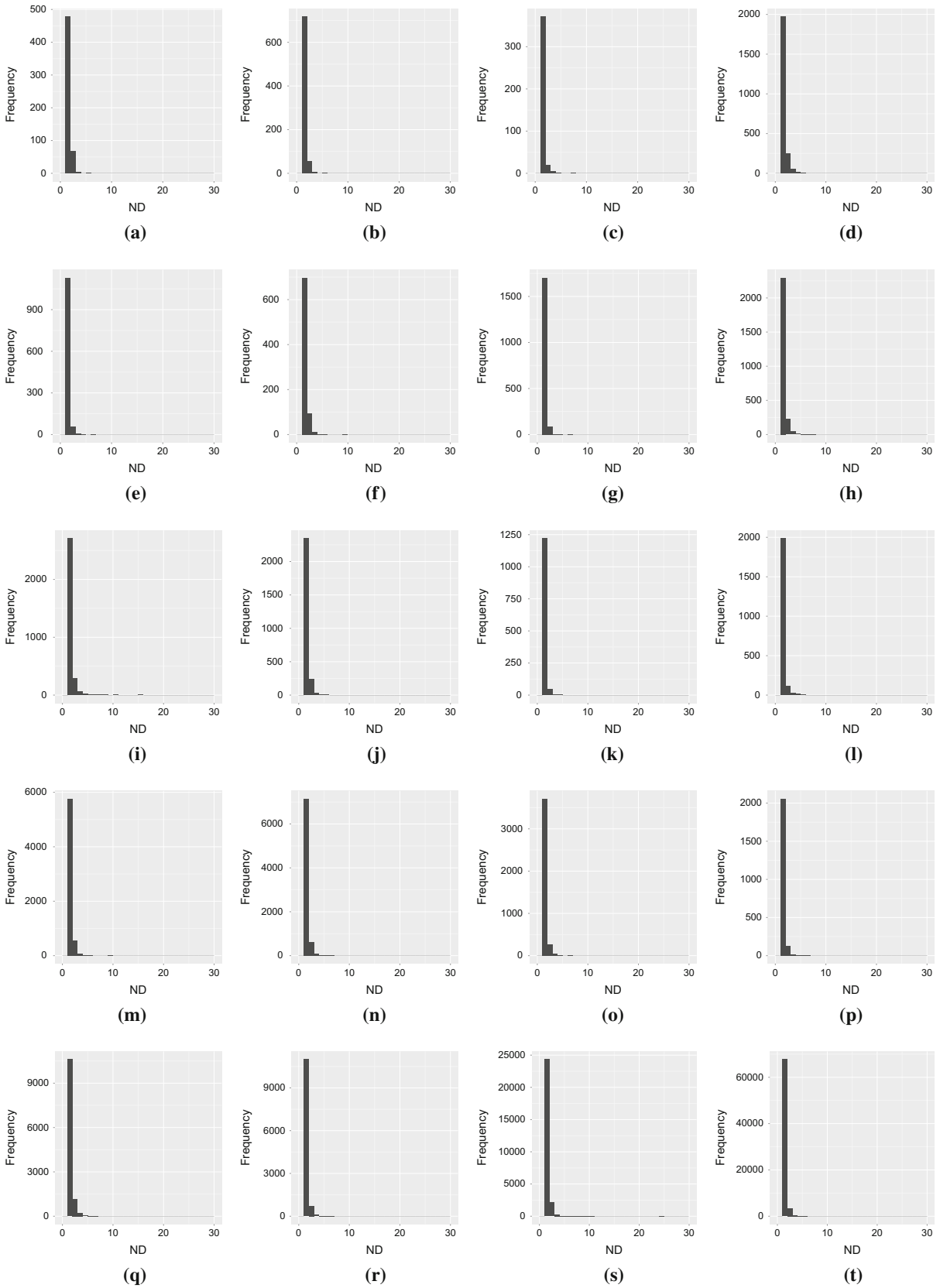

Fig. 9 Histogram of nesting depth (ND). **a** VI, **b** LIGHTTPD, **c** XFIG, **d** SENDMAIL, **e** SYLPHEED, **f** GIT, **g** APACHE, **h** LIBXML2, **i** EMACS, **j** OPENLDAP, **k** SUBVERSION, **l** IMAGEMAGICK, **m** PYTHON, **n** PHP, **o** POSTGRESQL, **p** GIMP, **q** GLIBC, **r** MYSQL, **s** GCC, **t** LINUX KERNEL

Table 5 Nesting depth (ND) descriptive measures (NOTLB: number of top-level branches)

System	NOTLB	Mean	Median	95th	Max	Mode	%	Gini
VI	551	1.14	1	2	5	1	86.9	0.10
LIGHTTPD	780	1.08	1	2	5	1	92.1	0.07
XFIG	398	1.09	1	2	7	1	93.4	0.08
SENDMAIL	2,290	1.17	1	2	5	1	86.2	0.13
SYLPHEED	1,197	1.06	1	2	6	1	94.4	0.05
GIT	809	1.17	1	2	9	1	86.2	0.12
APACHE	1,799	1.05	1	2	6	1	94.7	0.05
LIBXML2	2,585	1.14	1	2	7	1	88.8	0.11
EMACS	3,106	1.18	1	2	15	1	87.3	0.14
OPENLDAP	2,626	1.12	1	2	5	1	89.5	0.09
SUBVERSION	1,277	1.04	1	1	4	1	95.7	0.04
IMAGEMAGICK	2,139	1.09	1	2	5	1	93.0	0.08
PYTHON	6,416	1.12	1	2	9	1	89.8	0.09
PHP	7,868	1.10	1	2	6	1	90.8	0.09
POSTGRESQL	4,044	1.11	1	2	6	1	91.8	0.09
GIMP	2,216	1.09	1	2	6	1	92.8	0.08
GLIBC	12,062	1.14	1	2	6	1	88.0	0.11
MYSQL	11,850	1.08	1	2	6	1	92.8	0.07
GCC	26,888	1.11	1	2	24	1	90.6	0.09
LINUX KERNEL	71,591	1.05	1	2	5	1	94.6	0.05

Fig. 10 *ComplianceRate* and *CompliancePenalty* functions [24]

$$ComplianceRate[p,k] = \frac{|\{ S \in Corpus \mid p\% \text{ of the entities in } S \text{ have } M \leq k\}|}{|Corpus|}$$

$$penalty_1[p,k] = \begin{cases} \dfrac{90 - ComplianceRate[p,k]}{90} & \text{if } ComplianceRate[p,k] < 90 \\ 0 & \text{otherwise} \end{cases}$$

$$penalty_2[k] = \begin{cases} \dfrac{k - Median90}{Median90} & \text{if } k > Median90 \\ 0 & \text{otherwise} \end{cases}$$

$$CompliancePenalty[p,k] = penalty_1[p,k] + penalty_2[k]$$

ance functions described in Sect. 5.1, obtaining the following result:

> at least, 85 % of the feature constants in a system should have SD ≤ 6

In fact, this threshold holds for all systems in our corpus except VI and SYLPHEED, which exceed the threshold only marginally. In VI, we observe that 83 % of the feature constants have a SD $\leq$ 6 and, in SYLPHEED, this percentage is 82 %. However, the proposed relative threshold holds for large and complex systems, with thousands of *ifdefs*, such as the LINUX KERNEL, GCC, and MYSQL. Figure 11 shows the percentile functions for the SD values of each subject system. The x-axis represents the percentiles, and the y-axis represents the upper SD values of the feature constants matching the percentile. The plot nicely illustrates that SD values are heavy-tailed, as already concluded in Sect. 4.1 . However, there are two systems whose SD values begin to grow earlier, around the 85th percentile, which are exactly VI and SYLPHEED.

These results suggest that the proposed relative threshold reflects the most common scattering distributions found in our corpus. Another corpus, however, may yield a different threshold (e.g., a corpus with systems of a particular domain). However, assuming that we selected a representative sample of C-preprocessor-based systems, including small, medium, and large systems, we expect that different corpora would not produce radically different thresholds. In other words, not following the thresholds by a small margin—like in VI and SYLPHEED—does not necessarily mean a serious design flaw. However, if only 50 % of the feature constants in a system have SD $\leq$ 6, this would certainly raise more serious concerns on the quality of the feature implementation

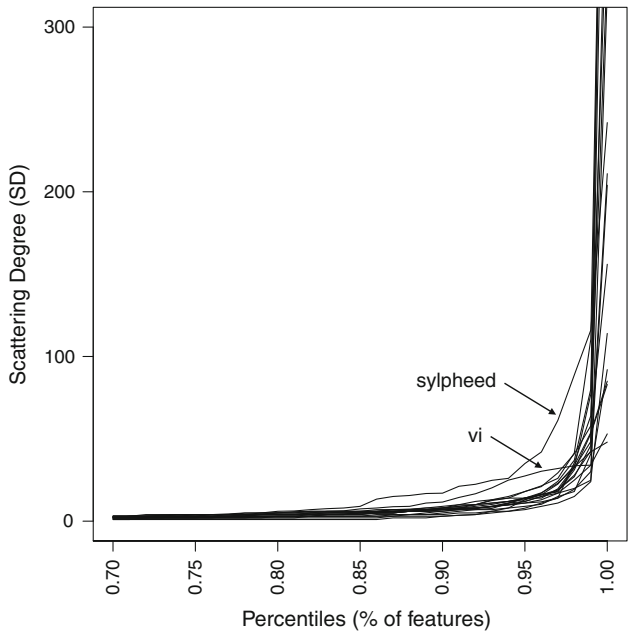

Fig. 11 Percentile plots of scattering degrees (SD)

structure. For example, Spencer and Collyer [31] claim that ifdef-based implementation should follow basic principles of software engineering, including clean interfaces and information hiding. More specifically, *ifdefs* should be hidden behind interfaces, making it possible to implement the bulk of the software as a single version using these interfaces. Code that does not follow the proposed thresholds for scattering degree might, for example, have many *ifdefs* that do not follow this general principle.

5.3 Thresholds for tangling degree

Tangling degrees approximate an uniform distribution, allowing to directly define thresholds. After inspecting the results in Table 4, specifically the mode and its relative frequency (%), we propose the following threshold:

at least, 80 % of the feature expressions in a system should have TD = 1

To define the threshold we assume that the mode of the TD distributions (which is equal to one in all systems) should correspond to, at least, 80 % of the feature expressions in each system. In other words, we assume that systems where the mode corresponds to less than 80 % of the feature expressions deviate from an uniform distribution and therefore are outliers. All systems in our sample follow this threshold, except IMAGEMAGICK.

5.4 Thresholds for nesting depth

After inspecting the measures of Table 5, specially the mode and its frequency, we propose the following threshold:

at least, 85 % of the top-level branches in a system should have ND = 1

To define this threshold we followed the assumptions of the TD threshold. However, in this case, we are requiring the mode of the ND distribution (ND = 1) to correspond to, at least, 85 % of the top-level *ifdef* branches (and not 80 %, as in the TD threshold). The reason is that in our sample, the mode of ND is one in all systems, and it corresponds to, at least, 86.2 % of the top-level *ifdef* branches in each system. Therefore, the proposed threshold is followed by all systems in our sample. The absence of systems not following the proposed threshold is explained by the fact that the ND distributions are quite similar across all subject systems.

5.5 Discussion

The proposed thresholds can be used to check whether a system implementation includes a complex usage of *ifdefs*, at least when compared with other relevant systems (i.e., the systems considered in our Corpus). To illustrate this usage, we applied our thresholds on XTERM (version 3.1.8), the standard terminal emulator for the XWindow system. Existing research shows that XTERM makes a heavy and complex usage of *ifdefs*. For example, Liebig et al. show that almost 40 % of XTERM's lines of code are enclosed by *ifdefs* [19]. Moreover, almost 10 % of the *ifdefs* in XTERM are undisciplined annotations, i.e., they delimit tokens that do not align with the syntactic code structure, e.g., with entire statements, functions, and type declarations [20]. Furthermore, we inspected the description of 318 patches of XTERM, from 1996 to 2015.[6] We found that 82 patches (26 %) included 110 changes in *ifdefs*, to correct bugs, to implement new features, or due to refactorings. We provide three examples of such changes:

- Patch #315: *"fix an ifdef'ing problem, where –disable-dec-locator would turn off logic needed for DECIC and DECDC"*.
- Patch #275: *"adjust ifdef's for putenv and unsetenv in case only one of those is provided on a given platform"*.
- Patch #216: *"ifdef'd Sun function-key feature to make it optional, like HP and SCO"*.

Therefore, we hypothesize that XTERM *should be classified as an outlier system*, according to the thresholds for SD, TD, or ND derived in the previous sections. To check this hypothesis, we used FSCAT to compute the distribution of the SD, TD, and ND values in XTERM. These distributions are presented in Fig. 12.

[6] XTERM change log is available at http://invisible-island.net/xterm/xterm.log.html.

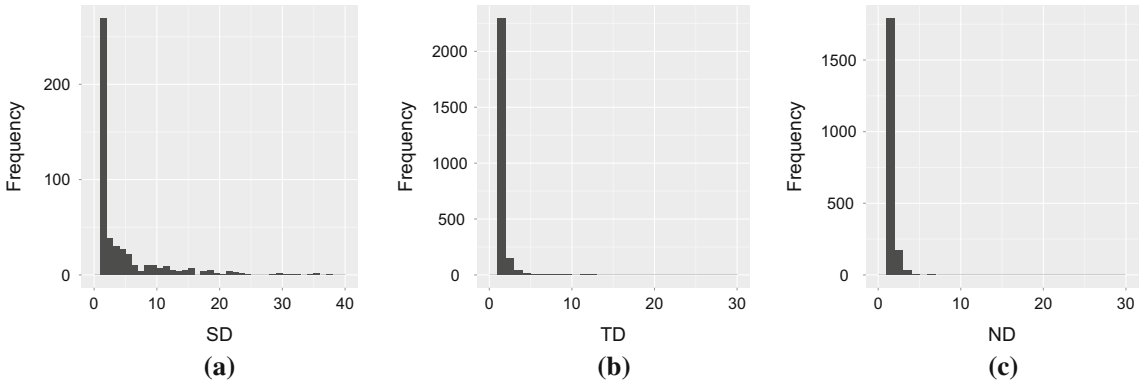

Fig. 12 Histogram of **a** scattering degree (SD), **b** tangling degree (TD), **c** nesting depth (ND) in XTERM

We then checked whether XTERM follows the proposed thresholds, with the following results:

- The threshold derived for SD states that a system should have, at least, 85 % of the *ifdefs* with $SD \leq 6$. However, in XTERM, only 79 % of the *ifdefs* have $SD \leq 6$. Therefore, XTERM is indeed an outlier regarding SD.
- The threshold for TD states that a system should have, at least, 80 % of the *ifdefs* with TD = 1. Indeed, XTERM has 91 % of the *ifdefs* with TD = 1. Therefore, it is not an outlier for TD.
- The threshold for ND states that a system should have, at least, 85 % of the *ifdefs* with ND = 1. Indeed, XTERM has 89 % of the *ifdefs* with ND = 1. Therefore, it is not an outlier for ND.

To conclude, the derived thresholds indeed indicate that XTERM has a complex usage of *ifdefs*, which manifests mainly in scattering. Regarding tangling and nesting, XTERM is not different from the systems in our corpus.

6 Threats to validity

A threat to external validity of our conclusions is the selection of the subject systems. We acknowledge that the current selection does not support us to conclude that our findings are applicable to every C-preprocessor-based system. Specially, the proposed thresholds for SD, TD, and ND should be used with caution, as they heavily depend on context, as usual with software metrics [30,38]. However, we attempted to increase external validity by carefully selecting mature systems of different sizes from different application domains.

The mechanisms in which features are implemented also pose threat to external validity. Since features may be implemented in different ways depending on the programming language, scattering, tangling, and nesting may not have the same behavior as observed in C-preprocessor-based systems.

Different programming styles used by developers to write *ifdefs* may affect the measured degrees, a threat to construct validity. As an example, Fig. 13 shows two fragments of *ifdef* code, with exactly the same behavior. However, since they have different *ifdef* structures, the measured metric values are different (in Style 1, `FEATURE_A` has SD = 1, but in Style 2, `FEATURE_A` has SD = 2). We attempt to mitigate this threat to validity by analyzing different systems, from different application domains. This way, we are not favouring one style over the other.

Another threat to internal validity arises when computing the three metrics we considered. When using FSCAT, we consider all the C source code of each system, and we do not distinguish files that are automatically generated (e.g., those produced by parser generators) from those that are not. We also do not discard unit test files. Thus, our results are, to some extent, subject to the influence of the file type. We argue, however, that the majority of the files we take for analysis are not automatically generated (as we confirmed after a random inspection).

Last but not least, our results indicate that feature scattering follows a power-law distribution in 14 out of 20 of our subjects. However, it might be the case that other distributions different from power laws are in fact a better fit (e.g., log-normal or stretched exponential). Even if that turns out to be true, conclusions will be the same (i.e., scattering will still be a heavy-tailed distribution).

7 Related work

Liebig et al. [19] analyzed 40 systems written in C showing how developers use the C preprocessor when implementing features and their associated *ifdefs* in source code. The authors consider not only scattering, tangling, and nesting, but also metrics measuring the granularity of annotations (the syntactic location where an *ifdef* occurs—e.g., at a global level, inside a function, and inside a block) and the type of annotated code (homogeneous, meaning that a verbatim

Fig. 13 Implementing *ifdefs* with different programming styles

```
//Style 1:

#ifdef FEATURE_A
 //code a
#elif FEATURE_B
 //code b
#endif

//Style 2:

#ifdef FEATURE_A
 //code a
#endif
#if !defined(FEATURE_A) && defined (FEATURE_B)
// code b
#endif
```

copy of the annotated block also appears in other annotated code; heterogeneous, with distinct extensions; or a mix of the two). The authors report their results using centrality and dispersion statistics, including mean and standard deviations. However, the properties of the underlying distributions have not been analyzed (e.g., whether they are symmetric, as in Gaussian distributions, or whether they are heavy-tailed, as in power-law distributions), which may turn results not representative of true typical values.

Hunsen et al. [13] used the same metrics and tools, including the transformations on the ifdef conditions, as Liebig et al. to compare metric values for open-source and industrial systems. While the authors report the metrics for the individual systems using centrality and dispersion statistics, they used distribution-independent statistical tests (i.e., the Mann-Whitney U test) to check their hypotheses regarding the difference between open-source and closed-source systems.

Eaddy et al. [7] investigated the relation between scattering and bugs, but do not prescribe a threshold limiting the degree of scattering. Nonetheless, they provide evidence that simple metrics, such as the scattering degree (a.k.a. *concern diffusion metric*), correlate with the number of bugs in a system, independent of its size.

Passos et al. [26] conducted a longitudinal case study of scattered features in the Linux kernel focusing on driver features. They analyze their evolution by analyzing scattering thresholds, linking findings to the kernel architectural decomposition, and studying how scattered driver features differ from non-scattered ones.

Outside the feature-oriented and product-line communities, there are different pieces of work checking the characteristic distribution of size, coupling, and cohesion-related metrics. Louridas et al. [21] studied the existence of power-law distributions in different kinds of software components, including Java classes, Perl packages, shared Unix Libraries, and Windows dynamic linked libraries (DLLs). The authors conclude that heavy-tailed distributions, usually power laws, appear at various levels of abstraction, in many domains, operating systems, and languages. Concas et al. [6] study ten different properties related to classes and methods of a large Smalltalk system, consistently finding non-Gaussian distributions of these properties. The authors then conclude that "*the usual evaluation of systems based on mean and standard deviation of metrics can be misleading*". Baxter et al. [4] report that some structural properties of Java software follow power-law distributions, while others do not. They conjecture that metrics measuring local properties that programmers are inherently aware about (e.g., out-degree distributions or number of method parameters) tend to follow distributions that are not power-law distributions. In fact, this is the case for tangling (TD) and nesting (ND) considered here. Finally, Taube-Shock et al. [33] studied connectivity in 97 open-source software systems, and they found that all these systems exhibit a similar scale-free dependency structure, with regard to both the overall connectivity and between-modules connectivity. For this reason, they concluded that high coupling is never entirely eliminated from software design and that, in fact, some degree of high coupling might be quite reasonable. A similar conclusion appears to apply to scattering in C-preprocessor-based systems.

From the observation that source code metrics may follow heavy-tailed distributions, researchers have recently proposed techniques for extracting reliable thresholds for existing metrics [1,9,23,24]. In this paper, we used the techniques of Oliveira et al. [23,24] for extracting reliable thresholds for the metrics we evaluated.

8 Conclusion

In our empirical study, we analyzed the statistical distribution of the scattering, tangling, and nesting degrees in 20 open-

source C- preprocessor-based systems. Our study revealed that feature scattering, as measured by the SD metric, follows a heavy-tailed distribution in all subject systems. In 14 systems (70 %), these heavy-tailed distributions matched a power law. Regarding tangling and nesting degrees, the metric values in all systems tend to a uniform distribution, with most values equal to one for both metrics and a few occurrences of slighter higher values. Based on these findings, we proposed thresholds for all three metrics, which are meant to be evaluated in follow-up studies.

In future work, we plan to extend our analysis to a larger set of systems. Another direction for future investigation is to assess feature-related metrics in systems written in languages other than C (e.g., in object-oriented languages) or using other preprocessor mechanisms, such as visual annotations [15,34]. Furthermore, we plan to explore other techniques for extracting thresholds, such as the one proposed by Alves et al. [1], or boxplots adjusted to skewed distributions [12]. Finally, we plan to check whether following (or not following) the proposed thresholds has an impact on other software properties, such as bugs and maintenance effort.

Acknowledgements We thank CNPq, CAPES, FAPEMIG, and the German Research Foundation (AP 206/4, AP 206/5, AP 206/6) for partially funding this project.

References

1. Alves, T.L., Ypma, C., Visser, J.: Deriving Metric Thresholds from Benchmark Data. In: Proceedings of the International Conference on Software Maintenance, pp. 1–10. IEEE (2010)
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Berlin (2013)
3. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Trans. Softw. Eng. **34**(2), 162–180 (2008)
4. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.: Understanding the Shape of Java Software. In: Proceedings of the International Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 397–412. ACM (2006)
5. Clauset, A., Shalizi, C.R., Newman, M.E.J.: Power-law distributions in empirical data. Soc. Ind. Appl. Math. Rev. **51**(4), 661–703 (2009)
6. Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-laws in a large object-oriented software system. IEEE Trans. Softw. Eng. **33**(10), 687–708 (2007)
7. Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do crosscutting concerns cause defects? IEEE Trans. Softw. Eng. **34**(4), 497–515 (2008)
8. Favre, J.M.: Preprocessors from an Abstract Point of View. In: Proceedings of the International Conference on Software Maintenance, pp. 287–296. IEEE (1996)
9. Ferreira, K., Bigonha, M., Bigonha, R., Mendes, L., Almeida, H.: Identifying thresholds for object-oriented software metrics. J. Syst. Softw. **85**(2), 244–257 (2011)
10. Gillespie, C.S.: Fitting Heavy-Tailed Distributions: The PoweRlaw Package (2014). R package version 0.20.5
11. Gillespie, C.S.: The PoweRlaw Package: A General Overview (2014)
12. Hubert, M., Vandervieren, E.: An adjusted boxplot for skewed distributions. Comput. Stat. Data Anal. **52**(12), 5186–5201 (2008)
13. Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., Apel, S.: Preprocessor-based variability in open-source and industrial software systems: an empirical study. Empir. Softw. Eng. 1–34 (2015)
14. Jbara, A., Feitelson, D.: Characterization and Assessment of the Linux Configuration Complexity. In: International Working Conference on Source Code Analysis and Manipulation, pp. 11–20. IEEE (2013)
15. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: Proceedings of the International Conference on Software Engineering, pp. 311–320. ACM (2008)
16. Kästner, C., Apel, S., Ostermann, K.: The Road to Feature Modularity? In: Proceedings of the International Workshop on Feature-Oriented Software Development, pp. 1–8. ACM (2011)
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming, pp. 220–242. Springer (1997)
18. Krone M.; Snelting, G.: On the Inference of Configuration Structures from Source Code. In: Proceedings of the International Conference on Software Engineering, pp. 49–57. IEEE (1994)
19. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In: Proceedings of the International Conference on Software Engineering, pp. 105–114. ACM (2010)
20. Liebig, J., Kästner, C., Apel, S.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In: Proceedings of the International Conference on Aspect-Oriented Software Development, pp. 191–202. ACM (2011)
21. Louridas, P., Spinellis, D., Vlachos, V.: Power laws in software. ACM Trans. Softw. Eng. Methodol. **18**, 1–26 (2008)
22. Newman, M.: Power laws, Pareto distributions and Zipf's law. Contemp. Phys. **46**, 323–351 (2005)
23. Oliveira, P., Lima, F., Valente, M.T., Alexander, S.: RTTOOL: A Tool for Extracting Relative Thresholds for Source Code Metrics. In: Proceedings of the International Conference on Software Maintenance and Evolution (Tool Demo Track), pp. 1–4 (2014)
24. Oliveira, P., Valente, M., Paim Lima, F.: Extracting Relative Thresholds for Source Code Metrics. In: Proceedings of the International Conference on Software Maintenance, Reengineering and Reverse Engineering, pp. 254–263. IEEE (2014)
25. Passos, L., Guo, J., Teixeira, L., Czarnecki, K., Wasowski, A., Borba, P.: Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In: Proceedings of the International Software Product Line Conference, pp. 91–100. ACM (2013)
26. Passos, L., Padilla, J., Berger, T., Apel, S., Czarnecki, K., Valente, M.T.: Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In: Proceedings of the International Conference on Modularity, pp. 1–12. ACM (2015)
27. Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wasowski, A., Czarnecki, K., Borba, P., Guo, J.: Coevolution of variability models and related software artifacts. Empir. Softw. Eng. 1–50 (2015)
28. Queiroz, R., Passos, L., Valente, M.T., Apel, S., Czarnecki, K.: Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Systems. In: Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD), pp. 23–29. ACM (2014)

29. Serebrenik, A., van den Brand, M.: Theil Index for Aggregation of Software Metrics Values. In: Proceedings of the International Conference on Software Maintenance, pp. 1–9. IEEE (2010)
30. Souza, L., Maia, M.: Do software Categories Impact Coupling Metrics? In: Proceedings of the Working Conference on Mining Software Repositories, pp. 217–220. IEEE (2013)
31. Spencer, H., Collyer, G.: #ifdef Considered Harmful, or Portability Experience with C News. In: Proceedings of the USENIX Technical Conference, pp. 185–197. USENIX Association (1992)
32. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information Hiding Interfaces for Aspect-Oriented Design. In: Proceedings of the International Symposium on Foundations of Software Engineering, pp. 166–175. ACM (2005)
33. Taube-Schock, C., Walker, R.J., Witten, I.H.: Can We Avoid High Coupling? In: Proceedings of the European Conference on Object-Oriented Programming, pp. 204–228. Springer (2011)
34. Valente, M.T., Borges, V., Passos, L.: A semi-automatic approach for extracting software product lines. IEEE Trans. Softw. Eng. **38**(4), 737–754 (2012)
35. Vasa, R., Lumpe, M., Branchand, P., Nierstrasz, O.: Comparative Analysis of Evolving Software Systems Using the Gini Coefficient. In: Proceedings of the International Conference on Software Maintenance, pp. 179–188. IEEE (2009)
36. Vasilescu, B., Serebrenik, A., van den Brand, M.: You Can't Control the Unfamiliar: A Study on the Relations Between Aggregation Techniques for Software Metrics. In: Proceedings of the International Conference on Software Maintenance, pp. 313–322. IEEE (2011)
37. Wheeldon, R., Counsell, S.: Power Law Distributions in Class Relationships. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation, pp. 45–54. IEEE (2003)
38. Zhang, F., Mockus, A., Zou, Y., Khomh, F., Hassan, A.E.: How does Context affect the Distribution of Software Maintainability Metrics? In: Proceedings of the International Conference on Software Maintainability, pp. 1–10. IEEE (2013)

Rodrigo Queiroz is currently a M.Sc. candidate in Computer Science at the Federal University of Minas Gerais, Brazil. His main research interests include software architecture, programming languages, and software product lines.

Leonardo Passos is currently a Ph.D. candidate at the Electrical and Computer Engineering at the University of Waterloo, Canada. Passos has a master's degree in Computer Science from the Federal University of Minas Gerais, Brazil (2007). His main research interests include programming languages, software product lines, and software evolution.

Marco Tulio Valente received his Ph.D. degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an associate professor in the Computer Science Department. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. He is a "Researcher I-D" of the Brazilian National Research Council (CNPq). He also holds a "Researcher from Minas Gerais State" scholarship, from FAPEMIG. Valente has co-authored more than 70 refereed papers in international conferences and journals. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG.

Claus Hunsen is a Ph.D. Student at the Chair of Software Product Lines, University of Passau. He received his Master's Degree in 2013 from the University of Passau for his empirical work on type-checking strategies for software product lines. In his research, he focuses on product-line evolution, including techniques and prediction models for sustainable product-line engineering. He especially concentrates on preprocessor-based variability and feature-dependency networks. He is currently the main developer of the tool CPPSTATS.

Sven Apel is a full professor at the University of Passau, Germany, and holds the Chair of Software Product Lines. The chair is funded by the esteemed Emmy-Noether and Heisenberg Programs of the German Research Foundation (DFG). Prof. Apel received his Ph.D. in Computer Science in 2007 from the University of Magdeburg, Germany. His research interests include novel programming paradigms, software engineering and product lines, and formal and empirical methods. He is the author or co-author of over a hundred peer-reviewed scientific publications. Sven Apel is a member of the IFIP Working Group 2.11 (Program Generation). He serves regularly in program committees of top-ranked international conferences, he is member of the editorial board of IEEE Software, and he will be program-committee co-chair of the 31st International Conference on Automated Software Engineering (ASE). His work has received awards by the Ernst-Denert Foundation and the Karin-Witte Foundation.

Krzysztof Czarnecki is a Professor of Electrical and Computer Engineering at the University of Waterloo. Before coming to Waterloo, he was a researcher at Daimler Chrysler Research (1995–2002), Germany, focusing on improving software development practices and technologies in enterprise, automotive, and aerospace domains. He co-authored the book on "Generative Programming" (Addison-Wesley, 2000), which deals with automating software component assembly based on domain-specific languages. While at Waterloo, he held the NSERC/Bank of Nova Scotia Industrial Research Chair in Requirements Engineering of Service-oriented Software Systems (2008–2013) and has worked on a range of topics in model-driven systems and software engineering, including product lines engineering, design synthesis, variability modeling, model transformation, and domain-specific languages. He received the Premier's Research Excellence Award in 2004 and the British Computing Society in Upper Canada Award for Outstanding Contributions to IT Industry in 2008.