Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study

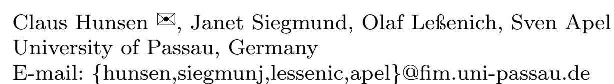
**Claus Hunsen · Bo Zhang ·
Janet Siegmund* · Christian Kästner ·
Olaf Leßenich · Martin Becker ·
Sven Apel**

Received: date / Accepted: date

Abstract Almost every sufficiently complex software system today is configurable. *Conditional compilation* is a simple variability-implementation mechanism that is widely used in open-source projects and industry. Especially, the C preprocessor (CPP) is very popular in practice, but it is also gaining (again) interest in academia. Although there have been several attempts to understand and improve CPP, there is a lack of understanding of how it is used in open-source and industrial systems and whether different usage patterns have emerged. The background is that much research on configurable systems and product lines concentrates on open-source systems, simply because they are available for study in the first place. This leads to the potentially problematic situation that it is unclear whether the results obtained from these studies are transferable to industrial systems. We aim at lowering this gap by compar-

* This author published previous work as Janet Feigenspan.

This work was partially supported by the DFG (German Research Foundation, 206/4-1, AP 206/5-1, AP 206/6-1) under the Priority Programme SPP1593 (Design For Future – Managed Software Evolution) and by NSF grant CCF-1318808. Furthermore, this work was partially sponsored by the Innovation Center Applied System Modeling, which is funded by Fraunhofer and the state Rhineland Palatinate of the Federal Republic of Germany.



Claus Hunsen ✉, Janet Siegmund, Olaf Leßenich, Sven Apel
University of Passau, Germany
E-mail: {hunsen,siegmunj,lessenic,apel}@fim.uni-passau.de

Bo Zhang
University of Kaiserslautern, Germany
E-mail: bo.zhang@cs.uni-kl.de

Christian Kästner
Carnegie Mellon University, USA
E-mail: kaestner@cs.cmu.edu

Martin Becker
Fraunhofer Institute of Experimental Software Engineering, Germany
E-mail: martin.becker@iese.fraunhofer.de

ing the use of CPP in open-source projects and industry—especially from the embedded-systems domain—based on a substantial set of subject systems and well-known variability metrics, including size, scattering, and tangling metrics. A key result of our empirical study is that, regarding almost all aspects we studied, the analyzed open-source systems and the considered embedded systems from industry are similar regarding most metrics, including systems that have been developed in industry and made open source at some point. So, our study indicates that, regarding CPP as variability-implementation mechanism, insights, methods, and tools developed based on studies of open-source systems are transferable to industrial systems—at least, with respect to the metrics we considered.

Keywords C preprocessor · CPPSTATS · variability · configurable systems · open-source systems · industrial systems · software product lines

1 Introduction

Almost every sufficiently complex software system today is configurable. *Configuration* and *variability* mechanisms form the technical basis for *software-product-line engineering*, as they facilitate to create families of software systems that share common assets [Pohl et al., 2005; Clements and Northrop, 2001]. The commonality and differences between the *variants* of a configurable system are often characterized by *features*—possibly optional or alternative end-user-visible behaviors or program characteristics [Kang et al., 1990; Clements and Northrop, 2001].

There are many mechanisms available to implement configurable systems, in general, and software product lines, in particular [Pohl et al., 2005; Czarnecki and Eisenecker, 2000; Clements and Northrop, 2001; Apel et al., 2013]. Here, we concentrate on *conditional compilation*, as supported by the C preprocessor (CPP). CPP supports conditional compilation through preprocessor directives (e.g., `#ifdef`), which enable the programmer to include or exclude parts of the code base by providing a corresponding configuration. CPP is widely used for the implementation of highly-configurable systems both in the open-source and the industrial world [Liebig et al., 2010; Ernst et al., 2002; Baxter and Mehlich, 2001; Ganesan et al., 2009; Pech et al., 2009; Jepsen and Beuche, 2009; Pearse and Oman, 1997]. One of the most prominent examples is the LINUX kernel, which uses the preprocessor to allow the developer to choose among 12 000 distinct options or features at build time [Tartler et al., 2011; Berger et al., 2010].

There have been many attempts to understand and improve CPP [McCloskey and Brewer, 2005; Weise and Crew, 1993; Favre, 1997; Vo and Chen, 1992; Krone and Snelting, 1994; Feigenspan et al., 2013; Kästner et al., 2008a; Adams et al., 2009; Kumar et al., 2012; Tomassetti and Ratiu, 2013; Kullbach and Riediger, 2001; Erwig and Walkingshaw, 2011; Singh et al., 2007; Ribeiro et al., 2011], but there is a lack of understanding of how it is used in open-source and industrial systems and whether different usage patterns have

emerged. Most research on configurable systems and product lines concentrates on open-source systems, simply because they are available for study in the first place. However, it is unclear whether study results from open-source systems are transferable to industrial systems.

Although open-source projects face the same challenges as industrial projects (e.g., quality assurance or release management), both differ in the means that are taken to solve the problems, including tools, development techniques, and organization structures [Mauerer and Jaeger, 2013]. For example, many open-source projects are characterized by large teams that are often spread all over the world [Mauerer and Jaeger, 2013]; or, as open-source software is available for everyone to use and adapt, novel ways of developing software have emerged, such as the use of open-source components in industrial systems [Godfrey and Germán, 2014].

As it is often hard for researchers to get hold of substantial industrial case studies, public availability is one of the main reasons for studying open-source systems in academia, although it is unknown whether the research results and conclusions apply also to industrial systems. We aim at shedding light at this issue by comparing the use of CPP in open-source projects and industry. The overarching goal is to understand similarities and differences of open-source and industrial systems regarding the use of CPP and to initiate a discussion and further research in this direction. Observing a similarity of both worlds would allow us to transfer insights, methods, and tools that have been proved useful for open-source systems also to industrial systems—at least, regarding the characteristics of the use of CPP that we study. A similarity would lead to a higher external validity of studies of software systems from either world, especially, as industrial case studies are rarely available. In the past, researchers have concentrated either only on open-source systems or on industrial systems [Liebig et al., 2010, 2011; Zhang et al., 2013; Ernst et al., 2002]; in this study, we compare both worlds to learn about their similarities and differences.

In an empirical study, we studied 20 open-source systems from different domains and 7 industrial systems from the embedded domain, written in C with CPP as variability-implementation mechanism, and of varying sizes and ages. As references for comparison, we applied several established size, scattering, tangling, and nesting metrics. For measurement, we extended our tool CPPSTATS[1], and applied it to all subject systems. Based on the measurement results, we evaluated whether the use of CPP differs between open-source and industrial systems by means of statistical tests. To take into account that sometimes there is no sharp line between open-source and industrial systems, we have analyzed 7 further open-source software systems from varying domains that had been proprietary until some point in their history (e.g., NETSCAPE NAVIGATOR, which is developed open-source as SEAMONKEY today). We chose two versions of each of these systems—the first open-source version and the most recent open-source version—to study their evolution regarding the use of CPP (from closed source to open source).

[1] `http://www.fosd.net/cppstats/`

The key result of our study is that the analyzed open-source and industrial systems are similar regarding most metrics measuring the use of CPP (one exception is the fraction of variable code, which is significantly higher in the industrial systems). This suggests that previous research on preprocessor variability based on open-source systems, including insights, methods, and tools, is applicable to industrial systems, of course, in the limits of the metrics we studied. Furthermore, systems that made a transition from a closed-source industrial context to open-source projects are not *per se* similar to the analyzed "pure" open-source or industrial systems, neither in their early industrial versions nor in the latest open-source versions.

In summary, we make the following contributions:

- We collected a substantial set of open-source and industrial subject systems, which are of different sizes and from varying domains, including open-source systems that have been developed in an industrial context until some point in their history.
- We enhanced the tool CPPSTATS for our analysis of CPP directives, and we present our collected measurement results for all software systems under consideration.
- We performed a statistical analysis on the obtained data and discuss the implications.

All experimental data are available at a supplementary website.[2]

The remainder of this paper is structured as follows: In Section 2, we give a detailed problem statement by describing CPP-based variability implementation and related research questions, and we present our experimental design. In Section 3, we give an overview of our tool CPPSTATS, which we used for measurement, and the detailed execution of the experiment. Next, we analyze and discuss the results in the Sections 4 and 5, respectively. We address threats to validity in Section 6 and related work in Section 7. We conclude the paper in Section 8.

2 Problem Statement, Research Questions, and Metrics

By means of a running example, we introduce the C preprocessor (CPP) and the terminology that we use throughout the paper in Section 2.1. Then, we pose our research questions in Section 2.2, followed by our experimental design, for which we use the goal-question-metric approach (GQM) [Basili et al., 1994]. Finally, in Section 2.3, we refine our research hypotheses with respect to the CPP characteristics of interest, and we present the subject systems in Section 2.4.

[2] `http://www.fosd.de/oss_vs_is/`

2.1 Characteristics of the C Preprocessor

The C preprocessor (CPP) has been developed to enhance the C programming language with simple lightweight meta-programming capabilities [Kernighan and Ritchie, 1988]. However, CPP operates at the level of text tokens, so it is able to handle any textual language artifact, including JAVA, C#, C++, HTML, PHP, and so forth. CPP is heavily used in practice, for example, in all LINUX distributions and the APACHE HTTP SERVER. Furthermore, it has been studied in many research projects (cf. Section 7).

CPP supports file inclusion (`#include`), the definition of object-like and function-like macros (`#define`), and conditional compilation (`#ifdef`). Object-like macro definitions and conditional compilation are the two most-frequently used features of the preprocessor [Kernighan and Ritchie, 1988], and we set our focus on them. For readability reasons, we will refer to the various conditional-compilation directives (`#ifdef`, `#if`, `#ifndef`, `#else`, `#elif`, and `#endif`) concisely as `#ifdef`, as `#ifdef` annotations, or as CPP annotations.

In Figure 1, we illustrate the use of the C preprocessor by means of the method `log_file_line`, taken from the code base of the APACHE HTTP SERVER. When an error occurs during APACHE's execution, this method takes the error-environment information (represented by the `info` argument) and dumps the file name and line number, at which the error occurred, to the given buffer `buf`. First, this method retrieves the file path from the `info` argument as a string, and then processes it according to the system's architecture, which is expressed via CPP *configuration constants* (e.g., Lines 9 and 12). This way, system-specific file-path characteristics—such as the path separator "\" on Windows (Lines 12–16), in contrast to "/" on other systems—can be handled flexibly.

The source code in Figure 1 contains three distinct configuration constants: `_OSD_POSIX`, `WIN32`, and `__MVS__`. These three constants are associated with the three operating systems BS2000/OSD, WINDOWS 32-BIT, and MULTIPLE VIRTUAL STORAGE, on which APACHE can be installed, among others. Any other system supported by APACHE is covered by the `#else` branch between the Lines 33 and 40. The code for extracting the file name from the path is variable and works effectively on the system for which the APACHE WEBSERVER is built.

`NULL` in Line 4 is a macro that is defined in file `stdio.h`; it acts as a null-pointer replacement. This macro is not used for configuration though, but as an ordinary constant.

Generally, a developer, maintainer, or end-user can define configuration constants before compilation. Only if a constant and its associated behavior is desired, it is defined as a macro (e.g., `#define Win32` or `#define NULL 0`). Generally, macros can be defined within the same file or in files that are included. They can also be set by makefiles or by compiler flags that are passed during compiler invocation [Kernighan and Ritchie, 1988]. In our study, we consider only macros that occur in `#ifdef` conditions inside a project's source code (excluding system libraries) and that are used in the sense of configu-

```
 1  static int log_file_line(const ap_errorlog_info *info, const char *arg,
 2                           char *buf, int buflen)
 3  {
 4      if (info->file == NULL) {
 5          return 0;
 6      }
 7      else {
 8          const char *file = info->file;
 9  #if defined(_OSD_POSIX) || defined(WIN32) || defined(__MVS__)
10          char tmp[256];
11          char *e = strrchr(file, '/');
12  #ifdef WIN32
13          if (!e) {
14              e = strrchr(file, '\\');
15          }
16  #endif
17
18          /* In OSD/POSIX, the compiler returns for __FILE__
19           * a string like: __FILE__="*POSIX(/usr/include/stdio.h)"
20           * (it even returns an absolute path for sources in
21           * the current directory). Here we try to strip this
22           * down to the basename.
23           */
24          if (e != NULL && e[1] != '\0') {
25              apr_snprintf(tmp, sizeof(tmp), "%s", &e[1]);
26              e = &tmp[strlen(tmp)-1];
27              if (*e == ')') {
28                  *e = '\0';
29              }
30              file = tmp;
31          }
32
33  #else /* _OSD_POSIX || WIN32 */
34          const char *p;
35          /* On Unix, __FILE__ may be an absolute path in a
36           * VPATH build. */
37          if (file[0] == '/' && (p = ap_strrchr_c(file, '/')) != NULL) {
38              file = p + 1;
39          }
40  #endif /*_OSD_POSIX || WIN32 */
41          return apr_snprintf(buf, buflen, "%s(%d)", file, info->line);
42      }
43  }
```

Fig. 1 The variable method `log_file_line` in `server/log.c` of Apache HTTP Server (version 2.4.6)

ration constants to realize variability. We do not consider macro expansion outside `#ifdef` conditions (e.g., `NULL` in Line 4), function-like macros (which accept parameters, but are used mostly as abstractions and not for configuration purposes), and include guards (which ensure that the content of a file is only included once).

Defining different sets of configuration constants (macros) will result in different compiled programs—the *variants* of the software system. Configuration constants can control whether functionality is included (*features*, in this sense), decide among hardware-specific characteristics (`WIN32`, for example, to support the operating system Windows), and may affect other options. In this study, we do not distinguish between these different uses of configuration constants.

Configuration constants can be composed to form *conditional expressions* using bit operators (e.g., `&`) or logical operators (e.g., `&&`). These conditions are used in `#ifdef`s, such as in Line 9, and act as guards for the code that is enclosed between an opening `#ifdef` and its closing complement (Lines 10 to 32). If the given conditional expression in Line 9 evaluates to a non-zero value (i.e., *true*), the annotated code is included; otherwise the code within the `#else` branch is included (Lines 34 to 39). Thus, the definition of configuration constants influences the evaluation of conditional expressions and, consequently, the presence or absence of annotated code at any level of granularity. We refer to conditionally-compiled code along with their according `#ifdef` annotation as `#ifdef` *block* or *variation point* (VP) [Pohl et al., 2005].

Although the syntax of CPP is simple, several problems arise when using the CPP excessively. On the one hand, constants may be *scattered*, that is, they are used within multiple conditions or files (e.g., `WIN32` is used in the Lines 9 and 12). On the other hand, multiple constants may be used in one place (a single file or a single condition), which is called *tangling* (e.g., Line 9). Furthermore, `#ifdef`s may be *nested* into already existing `#ifdef`s, where the inner depends on the evaluation of its own condition and, additionally, on the evaluation of the surrounding `#ifdef` [Krone and Snelting, 1994].

The excessive use of CPP is also called "`#ifdef` hell" [Lohmann et al., 2006], because it may easily obfuscate the source code. In particular, high degrees of scattering, tangling, and nesting can complicate program comprehension and maintenance [Favre, 1996; Ernst et al., 2002; Kästner, 2010; Liebig et al., 2011]. Therefore, scattering, tangling, and nesting metrics are often used in empirical studies, to characterize and compare different systems. In our study, we use corresponding metrics, along with common size-based metrics, as we will introduce in Section 2.2.3.

2.2 Research Questions and Experimental Design

Although both industrial software systems (IS) and open-source software systems (OSS) have to cope with the same problems during development and evolution [Mauerer and Jaeger, 2013], their different organizational processes suggest that they also differ with respect to the use of variability. This difference may have an effect on the application of tools and development methods, for example, in that open-source projects use other tools than industrial inhouse projects, which in turn may lead to different patterns of using conditional compilation and different degrees of scattering, tangling, and nesting.

However, a similar use of variability implementations in industrial systems and open-source systems would ease tool development and application, because any tool developed for the special needs of industrial systems would be applicable to open-source systems, and vice versa. This situation would also be convenient for researchers who could base their work on open-source systems, and be certain that the results are transferable to industrial systems, at least, within certain limits.

Our study aims at providing a basis for answering questions regarding the use of CPP in industrial and open-source systems, in particular, with respect to scattering, tangling, and nesting. To this end, we analyze a substantial set of configurable open-source and industrial systems, which we describe in Section 2.4. In particular, we aim at answering the following questions:

> **RQ_1.** *How do open-source and industrial systems differ with respect to their use of* CPP *annotations?*

To substantiate our findings regarding RQ_1, we additionally select a set of formerly closed-source software systems (FCS) that are now open-source. These systems have been developed closed-source originally for several years, but their source code was released publicly later. We analyze these systems by comparing two versions: the very first version of the software systems that was publicly available (FCS_1) and the most recent released version (FCS_∞). The comparison of both versions shows us how formerly closed-source systems evolved regarding their use of CPP after the public release of their source code and how they are characterized in general. Furthermore, we ask whether either older revisions of the systems in this category or more recent revisions can be used as substitutes for industrial systems. This is relevant, especially, if there are significant differences between industrial systems and general open-source systems.

> **RQ_2.** *Are formerly closed-source systems more like industrial or more like open-source systems in terms of the* CPP *usage patterns? How do they evolve?*

Regarding the use of CPP, there is a large set of metrics that can be used to compare the different kinds of software systems (IS, OSS, and FCS). To conduct the analysis systematically, we pursue the goal-question-metric (GQM) approach [Basili et al., 1994], to specify our goals operationally and to link the goals directly to the actual measurement results. The GQM approach structures software measurement into three levels: goals, questions, and metrics. The goal level defines a set of conceptual goals to be achieved, which are described by questions at the question level. This second level characterizes the way of achievement of a specific goal operationally. The questions address selected quality aspects of the selected software products. The metrics are associated with the appropriate questions to answer them quantitatively. We show the GQM model of our study in Figure 2, and we discuss the three levels next.

2.2.1 Goals

The main goal of our research is to explore whether CPP is used differently in industrial and open-source systems (G1) to answer the question of whether research is transferable from one system category to the other. This would affect tools, techniques, and development methods.

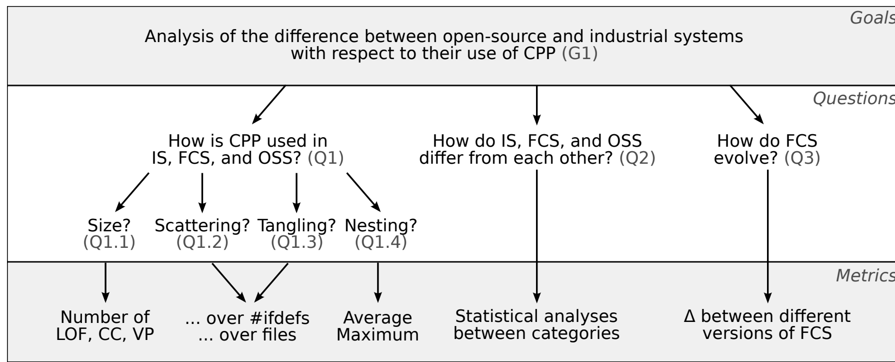


Fig. 2 GQM model of our study. (LOF: lines of normalized CPP-annotated code; VP: number of variation points (`#ifdef` blocks); CC: number of configuration constants; IS: industrial systems; OSS: open-source systems; FCS: formerly closed-source systems)

2.2.2 Questions

At the question level, we first examine how industrial and open-source systems use CPP to implement variability (Q1). This includes questions regarding general use and size (Q1.1), scattering (Q1.2), tangling (Q1.3), and nesting (Q1.4) of `#ifdef`s and configuration constants, respectively, as they characterize conditionally-compiled software systems in practice. Next, and based on the answer of question Q1, we study differences and similarities of open-source, industrial, and formerly closed-source systems, to determine whether the three system categories differ in their use of CPP (Q2). Finally, we ask how formerly closed-source systems evolve (Q3), which is especially important if there are differences found while answering Q2.

While the question Q1 is the basis for answering the other questions, the question Q2 mainly refers to our research question RQ_1 and (parts of) RQ_2. Question Q3 focuses on the last part of research question RQ_2.

2.2.3 Metrics

The metrics level of our GQM model contains all metrics that we selected to measure CPP usage, and also provides methods for comparison and for monitoring the evolution of the systems from the FCS category. We give more details on the measurement process we used in our study in Section 3.

Measurement of CPP *usage.* Measuring the characteristics of using CPP in the software systems under consideration answers the questions Q1.1 to Q1.4. To this end, we collected a set of metrics based on previous work [Liebig et al., 2010, 2011; Zhang et al., 2013], as listed in Table 1. We divided the metrics into four categories according to the questions stated before: size, scattering, tangling, and nesting metrics. Next, we introduce the metrics.

Table 1 List of metrics and corresponding descriptions, based on previous work [Liebig et al., 2010, 2011; Zhang et al., 2013]

Metrics	Description
Size metrics	
LOC	Lines of normalized code
LOF	Lines of normalized CPP-annotated code
PLOF	Fraction of CPP-annotated code (LOF/LOC)
VP	Number of variation points (`#ifdef` blocks)
CC	Number of configuration constants
Scattering metrics	
$SD_{\#ifdef}$	Average number of `#ifdef`s per CC
SD_{file}	Average number of files per CC
Tangling metrics	
$TD_{\#ifdef}$	Average number of CCs per `#ifdef`
TD_{file}	Average number of CCs per file
Nesting metrics	
ND_{avg}	Average nesting depth of `#ifdef`s
ND_{max}	Maximum nesting depth of `#ifdef`s

The *size metrics* quantify how many configuration constants are used (configuration constants, CC), and in how many `#ifdef` annotations (variation points, VP). Moreover, we measure the overall system size (lines of code, LOC) and the total number of CPP-annotated lines of code (LOF). The fraction of annotated lines of code (PLOF) characterizes the dimension of using CPP relative to the system's size. For measuring *scattering*, we count in how many `#ifdef` annotations ($SD_{\#ifdef}$) and files (SD_{file}) a configuration constant is used, on average. To characterize the degree of *tangling*, we count how many configuration constants are used, on average, per `#ifdef` annotation ($TD_{\#ifdef}$) and per file (TD_{file}). For both scattering and tangling metrics, we combine syntactically equivalent `#ifdef` conditions within one file. Finally, we measure the average and maximum nesting depth (ND_{avg} and ND_{max}, respectively) to characterize the *nesting* degree of a software system. A non-nested `#ifdef` annotation has a nesting degree of 1, and the value increases with each nesting level by 1.

We omit metrics that characterize granularity and extension type, although such metrics have been used in previous studies [Liebig et al., 2010, 2011], because we could not measure these consistently in all systems. The *granularity* measure determines at which level the CPP is used to make extensions to the source code (e.g., addition of whole functions, extensions of expressions, or optional parameters to a function). The *extension type* captures the fact that a single configuration expression can be used in different parts of the systems to manage distinct extensions to the code within each block (heterogenous extensions) or the same extension within all blocks using code duplicates (homogeneous extensions). For some industrial systems, our industrial partners provided only an abstraction of the source code that contained all preprocessor

directives and the number of lines of code they contained, but no C code, so we could not compute the described granularity and extension-type metrics.

Comparison of industrial and open-source systems. Regarding Q2, we compare the different categories of software systems (OSS, IS, and FCS) with respect to the metrics of Table 1. To this end, we employ a statistical analysis to determine the differences between categories, as we explain in Section 3.

Evolution of formerly closed-source systems. To answer Q3, we analyze the evolution of the seven formerly closed-source systems. We compare their first version of the open-source releases (FCS_1) and the current version (FCS_∞) to the "pure" open-source and industrial systems with respect to the defined metrics, and we characterize their evolution in terms of searching for changes regarding our metrics.

2.3 Hypotheses

As previously mentioned during the discussion of our research questions, we expect that the use of CPP indeed differs between industrial and open-source systems. Furthermore, we expect that formerly closed-source systems, in their earlier versions, use CPP similarly to industrial systems and, in their later versions, similarly to open-source systems—that is, we expect that their transition from closed source to open source manifests in measurable changes.

Overall, this leads to the following hypotheses. For simplicity, we omit to mention the expected evolution of the formerly closed-source systems and transitively implied comparisons in each hypothesis.

> **RH_1.** *Industrial systems and the first public releases of formerly closed-source systems have larger fractions of code annotated with* `#ifdef`*s than the systems from the other categories.*
>
> $$\mathrm{PLOF(IS)} > \mathrm{PLOF(OSS)} \qquad \mathrm{PLOF}(\mathrm{FCS}_1) > \mathrm{PLOF}(\mathrm{FCS}_\infty)$$
> $$\mathrm{PLOF(IS)} \approx \mathrm{PLOF}(\mathrm{FCS}_1) \qquad \mathrm{PLOF}(\mathrm{FCS}_\infty) \approx \mathrm{PLOF(OSS)}$$
>
> **PLOF**: fraction of CPP-annotated code; **IS**: industrial systems; **OSS**: open-source systems; **FCS**: formerly closed-source systems

The rationale of RH_1 is that the industrial systems follow a close and likely rigid development process that facilitates systematic variability management and implementation, as well as enforced coding guidelines. Furthermore, industrial projects are often closer tied to specific market segments and the pressure to serve them timely and efficiently, but also individually in the form of customer-specific extensions. These aspects may lead to a higher fraction of CPP-annotated code in the industrial systems.

RH₂. *Industrial systems and the first public releases of formerly closed-source systems are coarser grained with regard to their variability implementation than the systems from the other categories.*

$$\mathrm{SD}_{\mathrm{file}}(\mathrm{IS}) < \mathrm{SD}_{\mathrm{file}}(\mathrm{OSS}) \qquad \mathrm{SD}_{\mathrm{file}}(\mathrm{FCS}_1) < \mathrm{SD}_{\mathrm{file}}(\mathrm{FCS}_\infty)$$
$$\mathrm{SD}_{\mathrm{file}}(\mathrm{IS}) \approx \mathrm{SD}_{\mathrm{file}}(\mathrm{FCS}_1) \qquad \mathrm{SD}_{\mathrm{file}}(\mathrm{FCS}_\infty) \approx \mathrm{SD}_{\mathrm{file}}(\mathrm{OSS})$$

$$\mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{IS}) < \mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{OSS}) \qquad \mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_1) < \mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_\infty)$$
$$\mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{IS}) \approx \mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_1) \qquad \mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_\infty) \approx \mathrm{SD}_{\mathtt{\#ifdef}}(\mathrm{OSS})$$

$\mathbf{SD_{file}}$: average number of files per configuration constant (CC); $\mathbf{SD_{\#ifdef}}$: average number of `#ifdef`s per CC; **IS**: industrial systems; **OSS**: open-source systems; **FCS**: formerly closed-source systems

As RH_2, we expect a lower degree of scattering across files and `#ifdef`s in the industrial systems, because we expect CPP annotations being used there in a more disciplined and planned way, due to a more hierarchical organizational structure and more cohesive development teams. The delivery pressure of product development in industry likely influences the organization and the upfront planning of the industrial systems in an important way. This would affect not only the establishment of responsible teams per module or system part, but would accordingly also affect coding guidelines and the systems' architecture (according to Conway's law [Conway, 1968]). Although the organization of open-source systems evolved much over the last years [Fitzgerald, 2006], we expect to see a significantly more modular implementation with less CPP scattering in the industrial systems and less sharing.

RH₃. *Industrial code and code of the first public releases of formerly closed-source systems have a higher tangling degree regarding* `#ifdef`*s and files than the systems from the other categories.*

$$\mathrm{TD}_{\mathrm{file}}(\mathrm{IS}) > \mathrm{TD}_{\mathrm{file}}(\mathrm{OSS}) \qquad \mathrm{TD}_{\mathrm{file}}(\mathrm{FCS}_1) > \mathrm{TD}_{\mathrm{file}}(\mathrm{FCS}_\infty)$$
$$\mathrm{TD}_{\mathrm{file}}(\mathrm{IS}) \approx \mathrm{TD}_{\mathrm{file}}(\mathrm{FCS}_1) \qquad \mathrm{TD}_{\mathrm{file}}(\mathrm{FCS}_\infty) \approx \mathrm{TD}_{\mathrm{file}}(\mathrm{OSS})$$

$$\mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{IS}) > \mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{OSS}) \qquad \mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_1) > \mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_\infty)$$
$$\mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{IS}) \approx \mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_1) \qquad \mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{FCS}_\infty) \approx \mathrm{TD}_{\mathtt{\#ifdef}}(\mathrm{OSS})$$

$\mathbf{TD_{file}}$: average number of configuration constants (CC) per file; $\mathbf{TD_{\#ifdef}}$: average number of CCs per `#ifdef`; **IS**: industrial systems; **OSS**: open-source systems; **FCS**: formerly closed-source systems

The idea behind expecting higher tangling in industrial systems is that, given a planned variability and development phase in general to serve customers timely, efficiently, and also individually, the industrial systems may exhibit a higher number of configuration constants in their files in comparison to the other systems. In industrial systems that are shipped to customers or resellers, it is not uncommon to include customer-specific adaptations in the form of patches or features in the code base. We expect that such practice further increases the number of `#ifdef`s and the tangling degree in files as well as `#ifdef`s.

RH$_4$. *Industrial systems and the first public releases of formerly closed-source systems have higher nesting degrees than the systems from the other categories.*

$$ND_{avg}(IS) > ND_{avg}(OSS) \quad ND_{avg}(FCS_1) > ND_{avg}(FCS_\infty)$$
$$ND_{avg}(IS) \approx ND_{avg}(FCS_1) \quad ND_{avg}(FCS_\infty) \approx ND_{avg}(OSS)$$

ND$_{avg}$: average nesting depth of `#ifdefs`; **IS**: industrial systems; **OSS**: open-source systems; **FCS**: formerly closed-source systems

Finally, we expect that industrial code is more complex than open-source code in terms of `#ifdef` nesting. Nesting is closely related to scattering and tangling, but represents an independent dimension of complexity. Nesting `#ifdef`s is useful to maximize code reuse, but at the cost of code maintainability [Spencer and Collyer, 1992]. Along the lines of the argumentation for RH$_1$, we expect that the systematic planning for variability and reuse, as found in industrial systems, will manifest in more deeply nested `#ifdef` directives. Furthermore, customer-specific extensions of the code base may also influence the number of dependencies between different configuration options in industrial systems and thus lead to deeper nesting.

2.4 Subject Systems

We consider different categories of subject systems. On the one side, we study configurable industrial systems from the embedded domain, which are distributed as proprietary closed-source applications. On the other side, we examine two types of open-source systems from varying domains: "pure" open-source software and formerly closed-source software systems. The "pure" open-source systems have been developed as open-source projects from the beginning. The formerly closed-source software systems have been initially developed as closed source, but have been relicensed as open source later in their history, and have been maintained by a broader community for several years until today. These systems may be used as a substitute for industrial systems in research if a researcher is not able to get hold of a sufficient industrial case study—relying on the assumption that there is difference between industrial and open-source systems and, additionally, a substantial similarity of the formerly closed-source systems in their early versions to the industrial ones. We visualize our classification in Figure 3.

In total, we have selected 20 open-source and 7 industrial systems, as well as 7 formerly closed-source software systems. We provide a complete list of all subject systems in Table 2.

All systems are written mostly in C and use CPP as variability-implementation mechanism. Some of the systems do not consist of C code only (but also of C++ and other code). C is the most-used language in all systems. Note that some of the considered systems do not use only CPP for variability implementation (e.g., build-system variability, or dynamically loadable mod-

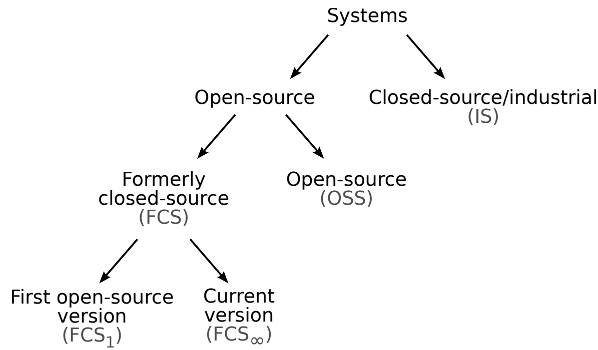


Fig. 3 Subject-system classification by closed-source and open-source licensing; abbreviations in brackets.

ules) [Tartler, 2013], but CPP makes up a substantial part of the systems' variability. However, we analyzed only CPP in our study.

To increase external validity, we attempted to sample systems of varying sizes (from 12 000 up to 10 million LOC), varying ages (two to 30 years of development), and from various domains. For open-source systems, we could select from a large number of potential systems to study. For industrial systems and formerly-open-source systems, we had less choice and had to rely on convenience sampling, which biased the selection of industrial systems toward the embedded-systems domain, but also the selection of formerly-closed-source systems, as we will describe.

Next, we describe the selection process of the subject systems by category.

2.4.1 Open-Source Systems (OSS)

The systems from the open-source category are taken from a corpus that has been used in previous studies by Liebig et al. [2010, 2011]. We selected a subset only to maintain a better balance (in terms of the number of subject systems) with the other system categories, which contain only 7 systems each. We selected the subset randomly from the corpus, but it still covers various sizes and domains, such as graphic editors, web servers, operating systems, and virtual machines. All systems have been developed for more than 4 years by their communities and for 17.9 years, on average.

2.4.2 Formerly Closed-Source Systems (FCS)

For the category of formerly closed-source systems, we opportunistically selected all systems written in C that were relicensed as open source and that we could get hold of. Our sample spans multiple domains, including operating systems, security applications, and virtual machines. All systems have been developed closed-source by a company for years until some point in their history. Most are widely known and commonly used in the open-source community. For each of these systems, we consider the first publicly available

Table 2 Full list of subject systems, divided by licensing aspects.

Software system	Version	Dev. Time	Domain	C [%]
Open-source systems (OSS)				
APACHE	2.4.6	(2013; 18)	Web server	95
BERKELEYDB	6.0.20	(2013; 27)	Database system	69
BUSYBOX	1.21.1	(2013; 19)	Unix tool collection	98
CHEROKEE	1.2.101	(2013; 12)	Web server	64
FREEBSD	9.1.0	(2013; 20)	Operating system	87
GIMP	2.8.6	(2013; 15)	Graphics editor	91
GNUMERIC	1.10.15	(2011; 10)	Spreadsheet application	97
GNUPLOT	4.6.3	(2013; 27)	Plotting tool	81
LIBXML2	2.9.0	(2012; 13)	XML library	85
LINUX	3.9	(2013; 22)	Operating system	97
OPENVPN	2.3.2	(2013; 11)	Security application	90
PARROT	5.0.0	(2013; 4)	High-level virtual machine	39
POSTGRESQL	9.3.0	(2013; 18)	Database system	97
QEMU	1.6.1	(2013; 10)	System-level virtual machine	97
SENDMAIL	8.14.7	(2013; 30)	Mail transfer agent	96
SQLITE	3.8.0.2	(2013; 13)	Database system	90
SUBVERSION	1.8.1	(2013; 13)	Revision control system	86
VIM	7.3	(2010; 19)	Text editor	65
XFIG	3.2.5b	(2013; 28)	Vector graphics editor	100
XTERM	296	(2013; 29)	Terminal emulator	95
Formerly closed-source systems: first version (FCS_1) and current version (FCS_∞)				
ANDROID SYSTEM CORE	1.0	(2007; 2)	Mobile operating system	88
	4.4_r1.2	(2013; 6)		80
BLENDER	2.26	(2003; 5)	3D graphics editor	99
	2.69	(2013; 10)		57
KORNSHELL (KSH93)	12-02-29	(2000; 17)	Terminal emulator	85
	12-08-01	(2012; 12)		85
MDNSRESPONDER	22	(2002; 4)	mDNS networking service	93
	541	(2013; 11)		85
NETSCAPE/SEAMONKEY	98-3-31	(1998; 4)	Internet suite	77
	2.23	(2013; 15)		25
(OPEN-)SOLARIS	1.0	(2005; 14)	Operating system	88
	13-10-28	(2013; 8)		95
VIRTUALBOX	1.6.0	(2007; 3)	System-level virtual machine	61
	4.3.2	(2013; 6)		66
Industrial systems (IS)				
A	–	(2013; 8)	Combustion-engine control	–
B	–	(2013; 7)	Frequency converter	–
C	–	(2009; 5)	Embedded automation controller	–
D	–	(2011; 6)	Inertial sensor controller	–
E	–	(2012; 7)	Frequency converter rail domain	–
F	–	(2013; 10)	Audio processing solutions	–
G	–	(2013; 7)	Inertial sensor controller	–

Versions without explicit version numbering are marked with the date of commit (YY-MM-DD). The development time is given in the format: (year of release; years of development time before release). The percental share of the C programming language is calculated using GITHUB's tool LINGUIST (`https://github.com/github/linguist`).

version (FCS_1) and the most recent version (FCS_∞). All considered systems have been developed as closed-source systems for, at least, 2 years (7 years, on average), and after that for, at least, 6 years (9.7 years, on average) with an open-source license. Moreover, some of the systems are mainly developed by companies (e.g., MDSNRESPONDER and ANDROID), others evolved into community-driven projects (e.g., BLENDER and OPENSOLARIS).

The selected systems have been all developed for many years as closed-source systems, and also a long time as open-source; we considered systems that have different ratios of closed-source and open-source development time, as shown in the second column of Table 3. Additionally, all systems (but one) changed substantially as a open-source systems; hence, we can expect significant changes in their use of CPP between the first (FCS_1) and most recent open-source release (FCS_∞). As shown in the last column of Table 3, the relative code churn (changed lines of code/total lines of code) between the two analyzed releases is higher than 1 for all systems except for KORNSHELL (0.015), which indicates a quite active development for all but one of the systems. In the case of KORNSHELL, the lower amount of code churn seems not very unusual for a shell—the very popular BASH has a relative code churn of 0.6 over the last 13 years despite two major releases and quite a lot of new functionality. Due to the mentioned considerations, we think of the selected subject systems as good representatives for formerly closed-source systems.

Likely the best known system in the FCS category is SEAMONKEY. The core of this Internet application suite was developed and distributed by the corporation NETSCAPE under the name NETSCAPE COMMUNICATOR until 1998. The software system was released as MOZILLA SUITE later—after NETSCAPE has made the source code publicly available through the MOZILLA ORGANIZATION. Today, the suite has been superseded by SEAMONKEY, which is developed by the community. So, in our comparison, we consider the first open-source version of NETSCAPE's source code[3] from March 31st, 1998, and the current SEAMONKEY version 2.23. The other systems have a similar history from closed-source to open-source development.

Table 3 List of formerly closed-source systems, including ratio of development time and code churn.

Software System	Ratio Open-/Closed-Source Development Time [years]	Relative Code Churn
ANDROID SYSTEM CORE	6/ 2	1.618
BLENDER	10/ 5	1.224
KORNSHELL (KSH93)	12/17	0.015
MDNSRESPONDER	11/ 4	1.056
NETSCAPE/SEAMONKEY	15/ 4	1.206
(OPEN-)SOLARIS	8/14	1.586
VIRTUALBOX	6/ 3	1.326

2.4.3 Industrial Systems (IS)

Also for the industrial category, we opportunistically included all systems we were able to locate and that have been developed primarily in C/C++ with CPP. We obtained the industrial systems from contractors and partners of the Fraunhofer Institute for Experimental Software Engineering. This leads

[3] ftp://ftp.mozilla.org/pub/mozilla.org/mozilla/source/

to a biased selection with a focus on the embedded-software domain. Within this domain, the systems cover multiple application domains, ranging from electrical frequency converter, to combustion-engine control, to inertial sensor control, to audio-media processing.

Each industrial system is highly configurable, developed for, at least, 5 years, and is managed to a large extent as a product line that contains commonality and variability to derive a portfolio of related products. All analyzed systems use CPP to implement variability.

3 Execution

Our study execution consists of two parts, measurement and statistical analysis, as we explain next.

3.1 Measurement

To compute the metrics described in Section 2.2.3, we extended our tool CPPSTATS, which captures the variability of C source code implemented with CPP. As the first processing step, CPPSTATS applies syntactic normalizations to the C source code, so that different programming styles become comparable: it strips all comments and blank lines, removes all indentation, and formats the code uniformly (pretty printing). Additionally, CPPSTATS removes all include guards, and it rewrites all nested `#ifdef`s as separate `#ifdef` blocks, each with a condition that conjoins their own conditional expression with the enclosing ones; information on the nesting depth is preserved.

After normalization, CPPSTATS transforms the source code into an XML representation using SRC2SRCML.[4] The XML representation holds all information of the C code (including the preprocessor statements) in the form of an approximated *abstract syntax tree* (AST), on which we perform our analysis.

For our initial example from Figure 1, CPPSTATS reports the following measurements: The total number of code lines (LOC) is 33 (after normalization). CPPSTATS identifies 3 different variation points (VP) in the code: the explicit ones on Lines 9 and 12, and the `#else` annotation on Line 33, which is rewritten as negation of the presence condition of Line 9. Additionally, the `#ifdef` on Line 12 is transformed into a conjunction with the enclosing `#ifdef` from Line 9. This will yield `#if (defined(_OSD_POSIX) || defined(WIN32) || defined(__MVS__)) && (defined(WIN32))` as the conditional expression. After transformation, all three `#ifdef`s consist of 3 configuration constants (CC) and enclose 17 lines in total (LOF). Consequently, 52% of the code sample is annotated by CPP directives (PLOF). Furthermore, we obtain scattering ($SD_{\texttt{\#ifdef}}$) and tangling degrees ($TD_{\texttt{\#ifdef}}$) of 3. The file-related scattering (SD_{file}) is 1, as there is only one file. The tangling degree over files (TD_{file}) is 3, because the code contains 3 distinct configuration constants.

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

Regarding the nesting degree, the outer `#ifdef` starts in Line 9 and ends in Line 40 (nesting degree of 1), whereas there is one nested `#ifdef` (nesting degree of 2). Thus, the average nesting depth is 1.5 (ND_{avg}); the maximum is 2 (ND_{max}).

3.2 Statistical Analysis

After measurement, we compare the measurement results obtained from the systems of the categories IS, OSS, and FCS_1, and FCS_∞. For this purpose, we use the tool R for the statistical analysis.[5]

Using a statistical analysis of the four different system categories, we check the influence of the system category on the measured values for this category, or, statistically spoken, we analyze whether the systems differ significantly. We used the following procedure [Anderson and Finn, 1996]: We conducted a *Shapiro-Wilk test* to check whether the values of a metric are normally distributed—we found that, for all metrics, the data are not normally distributed. Due to the non-normally distributed data and the small sizes of the different categories of software systems, we used the non-parametric *Mann-Whitney U test* and applied it in a pair-wise manner, instead of using ANOVA (analysis of variance), which is not stable in the given circumstances. We combined the statistical test with *False Discovery Rate (FDR) control*, which is a common method for multiple comparisons to minimize the Type-I error [Benjamini and Hochberg, 1995]. To measure the magnitudes of the significant differences that have been found with the Mann-Whitney U tests, we calculate *Cliff's Delta* as a statistical measure of effect size [Cliff, 1996].

The test's null hypothesis is that the data of the different system categories do not differ. We rejected the null hypothesis for p-values < 0.05, which is a commonly used level of significance [Cowles and Davis, 1982].

4 Measurement Results

In this section, we present the measurement results guided by the research hypotheses of Section 2.3. In Table 4 on Page 34, we provide the collected raw data of our analysis. We discuss data relevant for our hypotheses by means of scatter and violin plots, along with the results of the statistical analysis.[6]

All data are characterized either by absolute values or arithmetic mean values and standard deviations ($a \pm s$). We test all hypotheses using the procedure described in Section 3.2.

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

[6] In a violin plot, the white dot indicates the median, the small black horizontal bar shows the mean value, the wide black vertical bar spans from first to third quartile, and the shape describes the kernel density.

4.1 RH_1: Fraction of CPP-annotated code (PLOF)

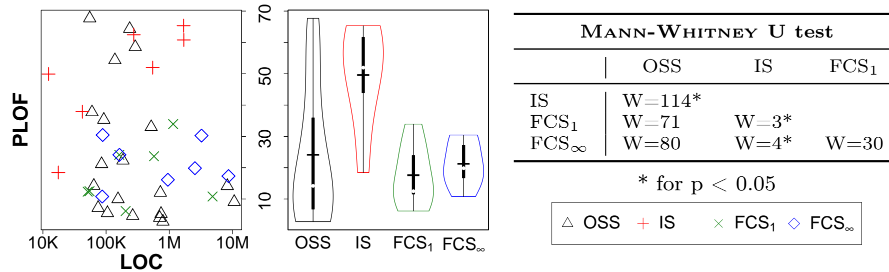
The open-source systems have an average of $24 \pm 22\%$ of CPP-annotated code lines. Looking closer, 13 open-source systems have a smaller fraction of annotated code than the average, while only few systems have a value of above 50% (e.g., LIBXML2 and OPENVPN). LINUX and FREEBSD, as operating systems, have only 9% and 14% of all code lines annotated with CPP.

The industrial systems have a larger fraction of CPP-annotated C code: $50 \pm 17\%$, on average, which is twice as high as for the open-source systems. Only the systems G (18%) and D (38%) have a percentage below 50; all other systems have more than half of their total lines of code annotated with CPP directives.

The early versions of the formerly closed-source systems (FCS_1) consist of $18 \pm 10\%$ of conditionally-compiled lines of code, on average. The current versions of these systems (FCS_∞) have an increased average fraction of $21 \pm 7\%$; four systems have a smaller or equal amount of conditional code than before; three have more.

In Figure 4, we show the scatter and violin plots and the results of hypothesis testing for RH_1. The results show that there is a significant difference between the industrial systems and all other system categories. We expected a difference of the industrial systems and the categories FCS_∞ and OSS in RH_1, and indeed both comparisons exhibit a substantial difference of 0.84 and 0.62, respectively, in terms of Cliff's Delta (effect size). The first open-source versions (FCS_1) and the industrial systems (IS) differ significantly in the fraction of CPP-annotated code, but we hypothesized in RH_1 that they are comparable. Cliff's Delta even reports an effect size of 0.88 for these two categories of software systems. We also found no significant difference between the FCS_1 versions and their more recent counterparts from the FCS_∞ category, although we assumed that in RH_1. Furthermore, there is no significant difference between open-source systems and both versions of the formerly closed-source systems. So, we reject this research hypothesis.

RH_1: Rejected.



MANN-WHITNEY U test	OSS	IS	FCS_1
IS	W=114*		
FCS_1	W=71	W=3*	
FCS_∞	W=80	W=4*	W=30

* for $p < 0.05$

△ OSS + IS × FCS_1 ◇ FCS_∞

Fig. 4 Results for the metric **PLOF** (fraction of CPP-annotated code): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the MANN-WHITNEY U TEST (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and most recent versions (FCS_∞) of formerly closed-source systems)

4.2 RH_2: Scattering of Configuration Constants within Files and `#ifdef`s (SD_{file}, $SD_{\#ifdef}$)

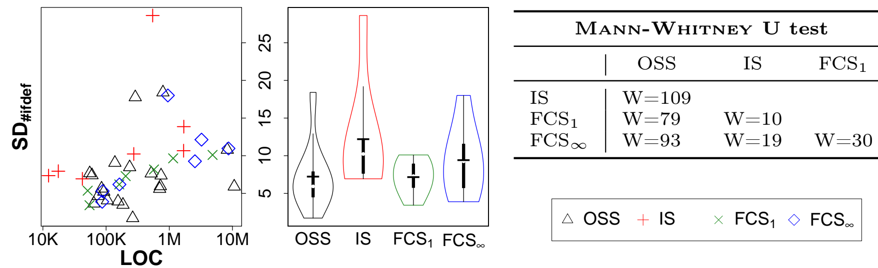
The open-source systems have a single configuration constant in, on average, 2.59 ± 1.37 files and 7.22 ± 4.30 `#ifdef`s. Most of the systems have an average between 1 and 3 files—only a few use configuration constants in up to 4 files—, and an $SD_{\#ifdef}$ value between 3 and 11.

The industrial systems have an average of 2.98 ± 1.42 files per configuration constant, whereas almost all systems have an average between 2 and 3 files. Furthermore, in industrial systems, the configuration constants also scatter extensively over `#ifdef`s (12.2 ± 76.1, on average).

The average scattering degree over files for the formerly closed-source systems is for both versions around 2.85; there is no significant evolution in any direction; the scattering degree regarding `#ifdef`s, however, increases from 7.17 ± 2.38 to 9.39 ± 4.85.

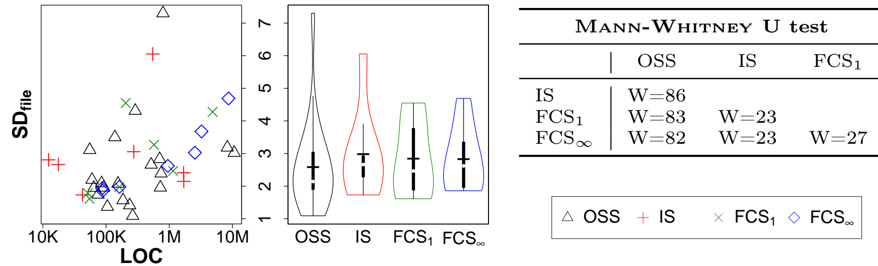
The results of our statistical tests (Figures 5 and 6) do not provide support for our hypothesis regarding the scattering degrees of different kinds of systems.

RH_2: Rejected.



MANN-WHITNEY U test			
	OSS	IS	FCS$_1$
IS	W=109		
FCS$_1$	W=79	W=10	
FCS$_\infty$	W=93	W=19	W=30

Fig. 5 Results for the metric $SD_{\#ifdef}$ (average number of `#ifdef`s per configuration constant): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the MANN-WHITNEY U TEST (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and current versions (FCS_∞) of formerly closed-source systems)



MANN-WHITNEY U test			
	OSS	IS	FCS$_1$
IS	W=86		
FCS$_1$	W=83	W=23	
FCS$_\infty$	W=82	W=23	W=27

Fig. 6 Results for the metric SD_{file} (average number of files per configuration constant): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the MANN-WHITNEY U TEST (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and current versions (FCS_∞) of formerly closed-source systems)

4.3 RH_3: Tangling of Configuration Constants within Files and `#ifdefs` (TD_{file}, $TD_{\texttt{\#ifdef}}$)

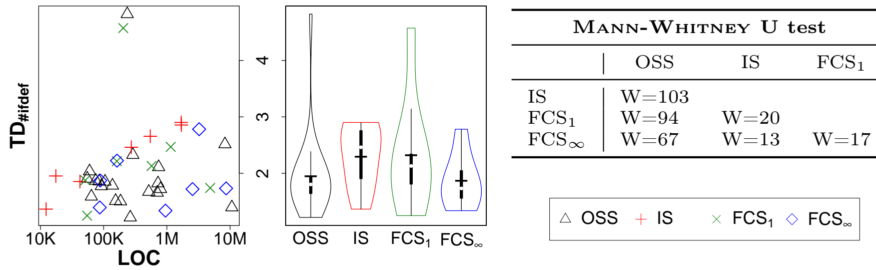
In the open-source systems, 1.95 ± 0.74 configuration constants are used, on average, per `#ifdef`, and 8.39 ± 7.05, on average, per file.

The industrial systems contain slightly, but not significantly, more configuration constants per `#ifdef` (2.29 ± 0.58) than the open-source systems. The tangling degree within a file is 4.27 ± 1.02, on average.

The formerly-closed-source systems have a $TD_{\texttt{\#ifdef}}$ value of 2.32 ± 1.07, while, on average, 4.42 ± 1.63 configuration constants are used per file. Over time, `#ifdef` tangling decreases to 1.87 ± 0.50 as well as the average number of constants within a single file decreases to 4.33 ± 1.17.

The results of our statistical tests (Figures 7 and 8) do not support our hypothesis of differing tangling degrees in the different kinds of systems.

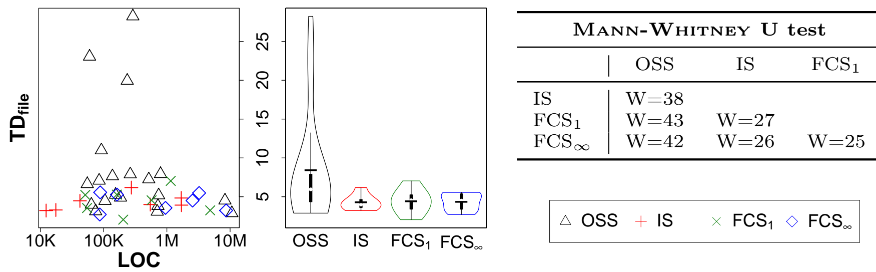
RH_3: Rejected.



Mann-Whitney U test

	OSS	IS	FCS$_1$
IS	W=103		
FCS$_1$	W=94	W=20	
FCS$_\infty$	W=67	W=13	W=17

Fig. 7 Results for the metric $\mathbf{TD}_{\texttt{\#ifdef}}$ (average number of configuration constants per `#ifdef`): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the Mann-Whitney U test (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and current versions (FCS_∞) of formerly closed-source systems)



Mann-Whitney U test

	OSS	IS	FCS$_1$
IS	W=38		
FCS$_1$	W=43	W=27	
FCS$_\infty$	W=42	W=26	W=25

Fig. 8 Results for the metric $\mathbf{TD}_{\mathbf{file}}$ (average number of configuration constants per file): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the Mann-Whitney U test (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and current versions (FCS_∞) of formerly closed-source systems)

4.4 RH_4: Nesting of `#ifdefs` (ND_{avg}, ND_{max})

The open-source systems have an average nesting degree of 1.17 ± 0.14. All systems have a value between 1.55 (GNUPLOT) and 1.04 (GNUMERIC).

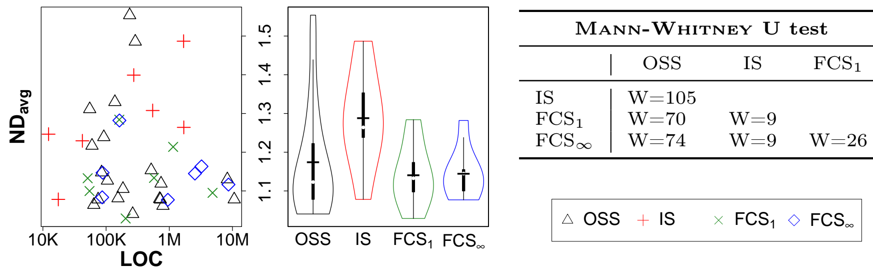
The industrial systems have a slightly higher nesting degree regarding their implemented `#ifdef` directives; the `#ifdefs` are nested between 1 and 2 times (average of 1.29 ± 0.13).

The formerly closed-source systems have an average nesting degree that is below the average of "pure" open-source systems, for both versions we considered (1.14). We did not observe substantial changes in their evolution (from FCS_1 to FCS_∞), except for BLENDER.

Regarding the maximum nesting degree, all systems have an ND_{max} value between 4 and 9, and all categories have about the same values. The outliers are the system B from the IS category, with a maximum depth of 12 and the operating system FREEBSD from the open-source category, which has an extraordinarily high ND_{max} value of 24 (cf. left plot in Figure 10).

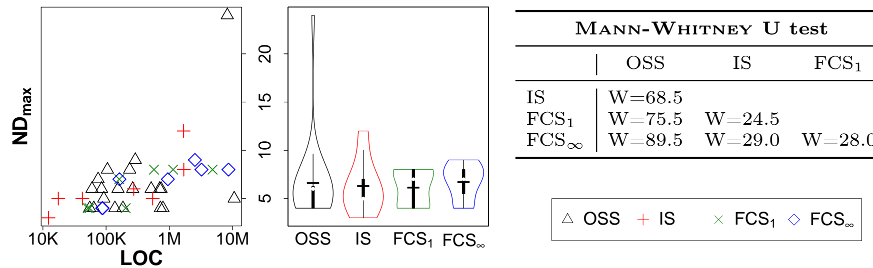
For both ND_{avg} and ND_{max}, there is no statistical evidence that there is a significant difference between the different kinds of systems, as also shown by the statistical analysis results in Figures 9 and 10.

RH_4: Rejected.



MANN-WHITNEY U test	OSS	IS	FCS_1
IS	W=105		
FCS_1	W=70	W=9	
FCS_∞	W=74	W=9	W=26

Fig. 9 Results for the metric $\mathbf{ND_{avg}}$ (average nesting depth of `#ifdefs`): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the MANN-WHITNEY U TEST (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and current versions (FCS_∞) of formerly closed-source systems)



MANN-WHITNEY U test	OSS	IS	FCS_1
IS	W=68.5		
FCS_1	W=75.5	W=24.5	
FCS_∞	W=89.5	W=29.0	W=28.0

Fig. 10 Results for the metric $\mathbf{ND_{max}}$ (maximum nesting depth of `#ifdefs`): Scatter plot of the raw data (*left*); violin plot for each system category (*middle*); results of the MANN-WHITNEY U TEST (*right*). (IS: industrial systems; OSS: open-source systems; first (FCS_1) and current versions (FCS_∞) of formerly closed-source systems)

5 Interpretation

In this section, we discuss the measurement results for each hypothesis, and we answer our research questions based on this discussion.

5.1 RH_1: Fraction of CPP-Annotated Code

In RH_1 (Section 2.3), we formulated our expectation that industrial systems and early versions of formerly closed-source systems have a larger fraction of code lines that are annotated with CPP's conditional-compilation directives. Indeed, we found that industrial systems contain a higher fraction of CPP-annotated lines of code—often, more than half the code can be compiled conditionally—than open-source and and both versions of the formerly closed-source systems, typically with 10 to 30 percent of CPP-annotated lines of code. This difference is statistically significant (MANN-WHITNEY U with FDR control) and also substantial in terms of effect size (0.62 up to 0.88).

We see a reason for this difference in CPP-annotated code in the differences in the development process and the specifics of the domains. The industrial systems that we considered have been developed as product lines to some degree. Furthermore, all the systems are from the embedded-systems domain. Product-line development in this context encourages explicit variability management, which systematically flows into the code base. Moreover, there are strategic aspects that press product-line developers to increase variability and code reuse, to serve the target market segment properly, and to improve return on investment. Furthermore, the domain of the industrial systems—all are from the embedded-systems domain—implies more fine-grained configurability in the code due to extensive non-functional requirements, which relies even more on a more disciplined planning of variability in general. So, both development process and the systems' domain likely influence the fraction of annotated code directly due to the requirements that are necessary for the development process. This result is likely representative, as all systems under consideration have matured over several years—as shown in Section 2.4.

Although we assumed that formerly closed-source systems use CPP similarly to industrial systems directly after their public release, we found as a second key result that these first open-source versions differ significantly from the industrial systems in their relative amount of CPP-annotated lines of code. A possible reason are the specifics of the different domains of the systems we considered for each category. Moreover, the industrial development teams may have refactored their products before their open-source release with respect to general amount of CPP use (e.g., reduction of functionality), so that the last closed-source releases of the formerly closed-source systems may have exhibited a higher fraction of annotated code (in comparison to the first open-source release).

As another result, we found no significant change in the PLOF metric between the two versions of the formerly closed-source systems, indicating

that they have not changed with respect to that metric. We did not expect this, because the systems have evolved for several years during their open-source development (as shown in Section 2.4.2). We see a possible reason in the refactoring process we mentioned before, which means that the industrial developers might have already adopted the first public releases of the formerly closed-source systems with respect to the general use of CPP.

The processes of the open-source and formerly closed-source projects that we looked at differ considerably from classic product-line development, which is likely a reason for less variable code. Those systems often have to support many different hardware platforms and usage scenarios in a domain, but do not need to actively distinguish different products for different customers as a strategy for revenue. In addition, some of the open-source systems are known for relying, beside CPP, on other variability mechanisms. In particular, LINUX and FREEBSD, although being highly configurable, do not make particularly much use of CPP (9% and 14% of CPP-annotated lines of code), but additionally use other variability mechanisms, such as dynamically loadable modules, parameter-based configuration, and build-system variability [Tartler, 2013], which mostly interact with the preprocessor-based variability implementation. Nevertheless, there are some open-source systems that exhibit a substantial fraction of CPP-annotated code, for example, VIM, OPENVPN, and LIBXML2. Although not product lines in the classic sense, most open-source systems are highly configurable; for example, LIBXML2 can be configured with XML's extensive language facilities, such as schemas, compression, parsers, and dialects. However, the embedded-systems domain of the industrial systems restricts the development teams from extensively using other variability mechanisms, due to their possible overhead to runtime and binary size.

5.2 RH_2–RH_4: Scattering, Tangling, and Nesting

In Section 2.3, we expected less scattering for the industrial systems and the first public release of formerly-closed source systems due to a more planned development process, but we found similar scattering degrees regarding files and `#ifdef`s across all system categories. The observed differences are not statistically significant. An explanation is that scattering, in general, impairs maintenance and evolution. Being successful, developers need to resort to some kind of disciplined use of CPP, to handle the inherent system complexity. Coding conventions, such as in LINUX-kernel development[7], make this point explicit, and seem to be adopted more and more in open-source and also industrial development practice.

Much like for scattering, we found a similar situation for tangling and nesting across all categories, again without any statistically significant differences. Since scattering, tangling, and nesting are closely related, the reason for this situation is likely the same as for scattering.

[7] `https://www.kernel.org/doc/Documentation/CodingStyle`

5.3 Importance of the Results

With the empirical results described and discussed before, we now return to our research questions that we stated in Section 2.2.

5.3.1 RQ_1: Differences and Similarities Between Open-Source and Industrial Use of CPP

Except for the significantly larger fraction of variable code (PLOF) in industrial systems, we found no statistical evidence that open-source and industrial systems differ with respect to any CPP characteristic that we identified in Section 2.1. The different organizational processes and domains of the open-source and industrial development projects do not seem to affect the use of CPP in software systems with respect to scattering, tangling, and nesting: there are variations between projects, but no systematic variations among the different categories of systems; that is, CPP is used similarly in the open-source and industrial systems—except for the higher fraction of CPP-annotated code in the industrial systems.

Following these considerations, the similar use of CPP indicates that research on a sample of open-source systems can be likely transferred to industrial systems, and vice versa—at least, in certain limits. This has immediate effect on research regarding tools and methods that have been developed for both industrial and open-source systems, as research effort may not need to be specialized and results are transferable to some extent. This is especially important for researchers who are not always able to get hold of industrial case studies. However, our study is just a first—but promising—step toward the goal of identifying proper substitutes for industrial systems in academia.

5.3.2 RQ_2: Similarity and Evolution of Formerly Closed-Source Systems

Referring to the discussion of the research hypotheses RH_1 to RH_4, we found an indication that the systems from the FCS category are, on the one hand, similar to the open-source systems with respect to some CPP characteristics ($SD_{\texttt{\#ifdef}}$, ND_{avg}), and, on the other hand, similar to the industrial systems with respect to other characteristics ($TD_{\texttt{\#ifdef}}$, TD_{file}), although there is no significant statistical evidence. Regarding the relative amount of CPP-annotated code, the formerly closed-source systems are more similar to the open-source systems (cf. the results in Figure 4), and less to the industrial systems (as supported by the statistical tests that found a significant difference between the industrial systems and all three other kinds of systems).

Moreover, changes in the metrics between the first and the current version of the formerly closed-source systems are only minor and do not follow a common trend (we found no statistically significant relation). In fact, there are systems for which we observed a decrease regarding some of the metrics and an increase regarding others.

We considered systems that are still developed by companies (e.g., mDSNresponder and Android) as well as community-driven projects (e.g., Blender and OpenSolaris), but there is no particular pattern. The dissimilarity to either open-source or industrial systems with regard to the use of cpp led us to rejecting our hypotheses about the increasing influence of the open-source community over time, or about effort of the industrial maintainers to adjust their systems to open-source standards and conventions. With the limited differences in cpp use overall, this is not surprising. A more detailed study on the evolution of the metrics might give more insight into the change the systems underwent by becoming open-source projects and being developed until today.

Overall, our findings—although only being a first step with a limited number of industrial case studies—indicate that formerly closed-source systems as well as open-source systems may be considered as substitutes for industrial case studies in the context of cpp research.

6 Threats to Validity

6.1 Internal Validity

In our analysis, we considered all cpp-annotated code except include guards. Include guards are used to prevent the multiple inclusion of header files (cf. Section 2.1) and would bias our results, because they are not a mechanism to implement variability. The remaining cpp code is treated equally, that is, we make no further distinction between `#ifdef`s implementing end-user visible functionality and `#ifdef`s used for portability means or hardware-specific code.

While developers could use other mechanisms to encode variability, we concentrated on the use of cpp, because it is a widely used tool in the development of highly-configurable systems and software product lines, both in the open-source and the industrial world.

Furthermore, we have not considered whether coding conventions concerning preprocessor usage have been enforced during the development of the subject systems. Such policies could have an effect on nesting depth or complexity of `#ifdef` annotations in the respective projects. However, the large number of projects we analyzed, across various domains, limits the influence of this confounding factor.

An additional threat to internal validity is the selection of metrics that we use for comparison. Some of the metrics that we used in previous work [Liebig et al., 2010, 2011] could not be applied to the industrial systems due to lack of source-code access (e.g., extension type and granularity metrics). This prevented us from performing an even more detailed analysis and limited us in the number of factors open to interpretation.

While conducting our experiments, less than 0.1% of all files and `#ifdef`s could not be processed by cppstats and had to be excluded, because either

src2srcml was not able to parse the file, or the `#ifdef` condition contained too complex expressions (e.g., due to extensive use of arithmetics). However, due to the small fraction, we do not expect this to affect the big picture of our study.

6.2 External Validity

A threat to external validity is the selection of subject systems, which applies especially to the selection of industrial and formerly closed-source systems, as they are hard to get hold of. Regarding the open-source and formerly closed-source projects, we were able to choose software systems from various domains, whereas the industrial case studies are all from the embedded-systems domain. Furthermore, the industrial systems have been developed as software product lines and the companies behind them seeked support, because they perceive a problem with their current practice. It is quite conceivable that some domains imply more platform-specific code than others, so the limitation to the embedded-systems domain threatens the generalizability of our results, as does the product-line-oriented development process. In future studies, we hope to get access to more industrial systems from further domains, to further increase external validity.

Also, the age of a system and the time of its peak development phase might threaten external validity, as coding styles, tools, and education of developers likely changed within a time span of 30 years. For example, a deeply nested `#ifdef` structure is probably easier to understand and maintain today with large screens and syntax highlighting available, than in the 1980s using terminals or monochrome monitors. We selected both old and young subject systems that have been developed for several years (for formerly closed-source systems both before and after their public release) and that are still developed actively by a company or community. Also, we considered some quite old projects (e.g., BerkeleyDB from the 1980s) and some younger projects (e.g., Android from 2005), so their main development time may imply differing coding styles present when starting development. Thus, we consider the subject systems as representatives for their corresponding system category as they come from different periods of time and all evolved over time. A further exploration of correlations between the metrics and the development time of the considered software systems is well beyond the scope of this study. We plan to analyze the successive evolution of the metrics over the whole time of development in future work.

Finally, we have analyzed only C code, whereas cpp itself can handle any textual artifacts (e.g., C++, Java, or just plain text). Thus, our results are applicable to languages related or similar to C. When a sample project used several languages, we ignored everything except C. However, we considered only projects with C as most-used language.

7 Related Work

Many researchers have stated that the use of CPP is likely to result in error-prone code that is difficult to understand and maintain [Spencer and Collyer, 1992; Favre, 1997; Lohmann et al., 2006; Schulze et al., 2013]. However, especially in the context of highly-configurable systems and software product lines, the preprocessor is still the standard tool [Jepsen and Beuche, 2009; Ganesan et al., 2009], and several researchers work on improved versions that avoid many of the pitfalls [Kästner et al., 2008b; Favre, 1997; Erwig and Walkingshaw, 2011; McCloskey and Brewer, 2005; Weise and Crew, 1993], on tools to migrate to alternative implementations [Adams et al., 2009; Kumar et al., 2012; Tomassetti and Ratiu, 2013; McCloskey and Brewer, 2005], and on tools to cope with existing CPP implementations [Vo and Chen, 1992; Krone and Snelting, 1994; Feigenspan et al., 2013; Kullbach and Riediger, 2001; Singh et al., 2007; Ribeiro et al., 2011].

Ernst et al. [2002] have conducted a large-scale study of CPP in 26 open source systems. Their analysis was focused primarily on macro use and corresponding pitfalls, whereas we focus on conditional compilation. Several tools have been developed to analyze the structure of conditional-compilation directives [Krone and Snelting, 1994; Tartler et al., 2011; Sutton and Maletic, 2007; Pearse and Oman, 1997], while ignoring the underlying source code. Only few tools analyze how conditional compilation affects the underlying source code [Adams et al., 2009; Liebig et al., 2010, 2011, 2013; Ribeiro et al., 2011; Kästner et al., 2011, 2012; Queiroz et al., 2014], but not considering the difference between open-source and industrial systems.

The largest family of studies in this context centers around a corpus of 40 open-source systems collected by Liebig et al. [2010]. This corpus has been studied using various metrics regarding conditional compilation and its implications [Liebig et al., 2010, 2011; Ribeiro et al., 2011]. We selected a subset of the open-source projects from this corpus and use several metrics proposed by Liebig et al.

For few systems, especially the Linux kernel, variability has been studied in depth [Lotufo et al., 2010; Tartler et al., 2011; Passos et al., 2012]. Baxter and Mehlich [2001] as well as Sutton and Maletic [2007] informally report on conditional-compilation practices from few commercial projects. However, in contrast to all prior work, our study is the first to compare the use of conditional compilation between open-source and industrial projects.

Closest to our work, Spinellis [2008] has investigated the difference among four operating-system kernels, two open source (FREEBSD, LINUX), a formerly closed-source (OPENSOLARIS), and one proprietary (WINDOWS RESEARCH KERNEL). Spinellis found a lower preprocessor usage in SOLARIS and the WINDOWS RESEARCH KERNEL than in LINUX and FREEBSD. He conjectured that companies such as SUN or MICROSOFT are discouraging the use of CPP. We cannot confirm this observation in our experiments for a larger sample with industrial systems from the embedded-systems domain.

8 Conclusion

The C preprocessor (CPP) is widely used both in open-source and industrial systems. Being one of the most successful tools for implementing variable and configurable systems, including software product lines, it has gained considerable attention in academia, especially, in the recent years. However, most scientific studies on the use of CPP concentrate on open-source systems. The main reason is that researchers often do not have access to systems developed in industry. This circumstance raises the issue of whether the insights and conclusions drawn from studying open-source systems can be transferred to industrial systems—at least, in certain limits. The importance of this issue cannot be underestimated, as it has an immediate influence on the relevance of methods and tools developed in academia and on where further research should go.

By means of an empirical study on a substantial set of open-source and industrial systems, we aim at shedding light at this issue. Specifically, we compared open-source and industrial systems using a set of well-established size, scattering, tangling, and nesting metrics that characterize the use of CPP for variability implementation. The key finding is that open-source systems and industrial systems are very similar in this respect (except for that industrial systems have larger fractions of variable code, but with similar complexity), which we did not expect due to the different development processes and largely differing domains of the selected subject systems.

In addition, we analyzed open-source systems that have been developed closed-source until some point in their history. For each system, we examined the first open-source version and the most current version, and we compared them to the "pure" open-source and industrial systems. We analyzed the evolution of these formerly closed-source systems by means of searching for changes regarding our metrics over their lifetime. We found no significant difference between these systems and the "pure" industrial and open-source systems either, except that they also exhibit a significantly lower fraction of variable code compared to the industrial systems.

What one can learn from our study is that research based on open-source systems, such as developing tools on top of CPP, is likely to be applicable to systems developed in industry. Although, our findings are just a first—but promising—step toward the goal of comparing industrial and open-source systems regarding their use of CPP, and have to be verified with industrial systems from other domains than the embedded-systems domain, there is a good indication that open-source and formerly closed-source systems can be considered as substitutes for industrial case studies.

References

Adams B, De Meuter W, Tromp H, Hassan AE (2009) Can We Refactor Conditional Compilation into Aspects? In: Proc. Int. Conf. Aspect-Oriented

Software Development (AOSD), ACM, pp 243–254
Anderson TW, Finn JD (1996) The New Statistical Analysis of Data. Springer
Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines: Concepts and Implementation. Springer
Basili V, Caldiera G, Rombach H (1994) Goal Question Metrics Paradigm. Encyclopedia of Software Engineering pp 528–532
Baxter I, Mehlich M (2001) Preprocessor Conditional Removal by Simple Partial Evaluation. In: Proc. Working Conference on Reverse Engineering (WCRE), IEEE, pp 281–290
Benjamini Y, Hochberg Y (1995) Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. Journal of the Royal Statistical Society Series B (Methodological) 57(1):289–300
Berger T, She S, Lotufo R, Wąsowski A, Czarnecki K (2010) Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In: Proc. Int. Conf. Automated Software Engineering (ASE), ACM, pp 73–82
Clements PC, Northrop L (2001) Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley
Cliff N (1996) Ordinal Methods for Behavioral Data Analysis. Erlbaum
Conway ME (1968) How Do Committees Invent? Datamation 14(5):28–31
Cowles M, Davis C (1982) On the Origins of the .05 Level of Statistical Significance. American Psychologist 37:553–558
Czarnecki K, Eisenecker UW (2000) Generative Programming – Methods, Tools and Applications. Addison-Wesley
Ernst MD, Badros GJ, Notkin D (2002) An Empirical Analysis of C Preprocessor Use. IEEE Transactions on Software Engineering (TSE) 28(12):1146–1170
Erwig M, Walkingshaw E (2011) The Choice Calculus: A Representation for Software Variation. ACM Transactions on Software Engineering and Methodology (TOSEM) 21(1):6:1–6:27
Favre JM (1996) Preprocessors from an Abstract Point of View. In: Proc. Int. Conf. Software Maintenance (ICSM), IEEE, pp 329–339
Favre JM (1997) Understanding-In-The-Large. In: Proc. Int. Workshop on Program Comprehension (WPC), IEEE, pp 29–38
Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2013) Do Background Colors Improve Program Comprehension in the #ifdef Hell? Empirical Software Engineering 18(4):699–745
Fitzgerald B (2006) The Transformation of Open Source Software. MIS Quarterly 30(3):587–598
Ganesan D, Lindvall M, Ackermann C, McComas D, Bartholomew M (2009) Verifying Architectural Design Rules of the Flight Software Product Line. In: Proc. Int. Software Product Line Conference (SPLC), ACM, pp 161–170
Godfrey MW, Germán DM (2014) On the Evolution of Lehman's Laws. Journal of Software: Evolution and Process 26(7):613–619
Jepsen HP, Beuche D (2009) Running a Software Product Line – Standing Still is Going Backwards. In: Proc. Int. Software Product Line Conference

(SPLC), ACM, pp 101–110

Kang K, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute

Kästner C (2010) Virtual Separation of Concerns: Toward Preprocessors 2.0. Logos Verlag Berlin

Kästner C, Apel S, Kuhlemann M (2008a) Granularity in Software Product Lines. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 311–320

Kästner C, Trujillo S, Apel S (2008b) Visualizing Software Product Line Variabilities in Source Code. In: Proc. Int. SPLC Workshop Visualisation in Software Product Line Engineering (ViSPLE), Lero Int. Science Centre, University of Limerick, Ireland, pp 303–312

Kästner C, Giarrusso PG, Rendel T, Erdweg S, Ostermann K, Berger T (2011) Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, pp 805–824

Kästner C, Ostermann K, Erdweg S (2012) A Variability-Aware Module System. In: Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, pp 773–792

Kernighan BW, Ritchie D (1988) The C Programming Language, Second Edition. Prentice-Hall

Krone M, Snelting G (1994) On the Inference of Configuration Structures from Source Code. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 49–57

Kullbach B, Riediger V (2001) Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In: Proc. Working Conference on Reverse Engineering (WCRE), IEEE, pp 3–12

Kumar A, Sutton A, Stroustrup B (2012) Rejuvenating C++ Programs Through Demacrofication. In: Proc. Int. Conf. Software Maintenance (ICSM), IEEE, pp 98–107

Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 105–114

Liebig J, Kästner C, Apel S (2011) Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In: Proc. Int. Conf. Aspect-Oriented Software Development (AOSD), ACM, pp 191–202

Liebig J, von Rhein A, Kästner C, Apel S, Dörre J, Lengauer C (2013) Scalable Analysis of Variable Software. In: Proc. Europ. Software Engineering Conference and the Int. Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 81–91

Lohmann D, Scheler F, Tartler R, Spinczyk O, Schröder-Preikschat W (2006) A Quantitative Analysis of Aspects in the eCos Kernel. In: Proc. Int. EuroSys Conference (EuroSys), ACM, pp 191–204

Lotufo R, She S, Berger T, Czarnecki K, Wasowski A (2010) Evolution of the Linux Kernel Variability Model. In: Proc. Int. Software Product Line Conference (SPLC), Springer, pp 136–150

Mauerer W, Jaeger MC (2013) Open Source Engineering Processes. it – Information Technology 55(5):196–203

McCloskey B, Brewer E (2005) ASTEC: A New Approach to Refactoring C. In: Proc. Europ. Software Engineering Conference and the Int. Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 21–30

Passos LT, Czarnecki K, Wasowski A (2012) Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In: Proc. Int. Workshop on Feature-Oriented Software Development (FOSD), ACM, pp 62–69

Pearse TT, Oman PW (1997) Experiences Developing and Maintaining Software in a Multi-Platform Environment. In: Proc. Int. Conf. Software Maintenance (ICSM), IEEE, pp 270–277

Pech D, Knodel J, Carbon R, Schitter C, Hein D (2009) Variability Management in Small Development Organizations – Experiences and Lessons Learned from a Case Study. In: Proc. Int. Software Product Line Conference (SPLC), ACM, pp 285–294

Pohl K, Böckle G, van der Linden F (2005) Software Product Line Engineering – Foundations, Principles, and Techniques. Springer

Queiroz R, Passos LT, Valente MT, Apel S, Czarnecki K (2014) Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Software Families. In: Proc. Int. Workshop on Feature-Oriented Software Development (FOSD), ACM, pp 23–29

Ribeiro M, Queiroz F, Borba P, Tolêdo T, Brabrand C, Soares S (2011) On the Impact of Feature Dependencies When Maintaining Preprocessor-Based Software Product Lines. In: Proc. Int. Conf. Generative Programming and Component Engineering (GPCE), ACM, pp 23–32

Schulze S, Liebig J, Siegmund J, Apel S (2013) Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. In: Proc. Int. Conf. Generative Programming and Component Engineering (GPCE), ACM, pp 65–74

Singh N, Gibbs C, Coady Y (2007) C-CLR: A Tool for Navigating Highly Configurable System Software. In: Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), ACM, 6 pages

Spencer H, Collyer G (1992) #ifdef Considered Harmful, or Portability Experience With C News. In: USENIX Summer Technical Conference, USENIX Association, pp 185–197

Spinellis D (2008) A Tale of Four Kernels. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 381–390

Sutton A, Maletic JI (2007) How We Manage Portability and Configuration with the C Preprocessor. In: Proc. Int. Conf. Software Maintenance (ICSM), IEEE, pp 275–284

Tartler R (2013) Mastering Variability Challenges in Linux and Related Highly-Configurable System Software. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg

Tartler R, Lohmann D, Sincero J, Schröder-Preikschat W (2011) Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In: Proc. Int. EuroSys Conference (EuroSys),

ACM, pp 47–60

Tomassetti F, Ratiu D (2013) Extracting Variability from C and Lifting it to mbeddr. In: Int. Workshop on Reverse Variability Engineering (REVE), pp 9–16

Vo K, Chen Y (1992) Incl: A Tool to Analyze Include Files. In: Proc. USENIX Conference, USENIX Association, pp 199–208

Weise D, Crew R (1993) Programmable Syntax Macros. In: Proc. Int. Conf. Programming Languages Design and Implementation (PLDI), ACM, pp 156–165

Zhang B, Becker M, Patzke T, Sierszecki K, Savolainen JE (2013) Variability Evolution and Erosion in Industrial Product Lines: A Case Study. In: Proc. Int. Software Product Line Conference (SPLC), ACM, pp 168–177

Table 4 Collected raw data of the subject systems

Subject system	LOC	LOF	PLOF	VP	CC	$SD_{\#ifdef}$	SD_{file}	$TD_{\#ifdef}$	TD_{file}	ND_{avg}	ND_{max}
Open-source systems (OSS)											
Apache	154 471	15 394	10	2 176	624	3.9 ± 9.1	2.0 ± 3.8	1.5 ± 1.2	5.2 ± 7.6	1.0 ± 0.0	6
BerkeleyDB	517 423	170 610	33	12 346	1 681	7.6 ± 22.8	2.6 ± 4.9	1.7 ± 1.1	7.2 ± 29.9	1.1 ± 0.1	6
BusyBox	184 992	41 248	22	5 131	1 376	3.4 ± 5.5	1.5 ± 2.6	1.5 ± 1.3	4.9 ± 8.8	1.1 ± 0.0	4
Cherokee	63 920	9 098	14	1 115	366	3.6 ± 7.6	1.9 ± 6.0	1.6 ± 1.1	3.9 ± 5.8	1.0 ± 0.0	4
FreeBSD	8 292 355	1 174 593	14	125 948	17 907	10.7 ± 501.8	3.1 ± 21.6	2.5 ± 1.5	4.5 ± 11.3	1.1 ± 0.0	24
GIMP	704 286	28 223	4	2 749	590	5.6 ± 14.4	2.8 ± 7.7	1.7 ± 1.5	3.0 ± 5.2	1.0 ± 0.0	6
Gnumeric	263 583	12 220	5	1 603	738	1.7 ± 3.1	1.0 ± 0.6	1.2 ± 1.2	7.8 ± 53.5	1.0 ± 0.0	6
gnuplot	84 882	17 964	21	1 904	420	5.6 ± 14.9	2.0 ± 4.2	1.9 ± 1.5	7.0 ± 12.6	1.1 ± 0.1	6
libxml2	236 117	151 866	64	11 820	2 073	8.4 ± 89.8	1.4 ± 2.6	4.8 ± 3.0	19.9 ± 133.5	1.5 ± 0.3	8
Linux	10 664 898	968 673	9	81 365	12 189	5.9 ± 42.9	3.0 ± 18.6	1.4 ± 1.2	2.8 ± 5.6	1.0 ± 0.0	5
OpenVPN	55 075	37 282	68	1 894	318	7.6 ± 20.2	3.1 ± 7.1	1.9 ± 1.4	6.6 ± 11.7	1.3 ± 0.2	4
Parrot	104 994	5 810	6	1 064	343	4.0 ± 6.4	1.3 ± 1.1	1.9 ± 1.6	4.4 ± 9.3	1.1 ± 0.0	8
PostgreSQL	741 076	39 017	5	6 548	1 278	7.3 ± 33.3	2.3 ± 6.1	2.1 ± 1.7	5.1 ± 23.2	1.1 ± 0.0	6
QEMU	722 944	86 873	12	8 062	1 393	5.9 ± 23.2	1.9 ± 5.1	1.8 ± 2.2	3.8 ± 12.4	1.0 ± 0.0	4
Sendmail	92 000	32 497	35	3 659	887	5.2 ± 10.7	1.9 ± 2.1	1.8 ± 1.1	10.9 ± 35.1	1.2 ± 0.1	5
SQLite	137 799	74 886	54	3 007	395	9.0 ± 19.5	3.5 ± 5.4	1.8 ± 1.0	7.6 ± 14.7	1.3 ± 0.2	4
Subversion	794 101	22 141	3	6 476	365	18.4 ± 47.5	7.3 ± 12.2	1.7 ± 1.2	7.9 ± 21.6	1.0 ± 0.0	4
Vim	289 567	169 475	59	14 524	969	17.7 ± 83.0	4.3 ± 7.4	2.3 ± 1.4	28.2 ± 36.6	1.4 ± 0.3	9
xfig	74 818	5 374	7	465	108	4.6 ± 6.6	1.7 ± 2.4	1.9 ± 2.0	3.1 ± 7.5	1.0 ± 0.0	7
xterm	59 856	22 572	38	2 658	462	7.4 ± 15.7	2.1 ± 2.5	2.0 ± 1.7	23.0 ± 41.5	1.2 ± 0.2	6
Formerly closed-source systems (FCS); first (FCS_1) and current version (FCS_∞)											
Android Core 1.0	54 702	6 839	13	964	297	3.4 ± 7.3	1.6 ± 2.8	1.3 ± 0.7	3.5 ± 6.6	1.1 ± 0.0	4
Android Core 4.4	86 857	9 380	11	1 199	300	3.8 ± 8.2	1.9 ± 4.3	1.4 ± 1.0	2.7 ± 5.1	1.0 ± 0.0	4
Blender 2.26	204 201	12 591	6	1 953	237	7.3 ± 27.7	4.5 ± 22.1	4.6 ± 8.2	2.0 ± 2.2	1.0 ± 0.0	4
Blender 2.69	950 360	153 514	16	12 289	1 983	18.0 ± 162.4	2.6 ± 10.8	1.3 ± 1.1	3.5 ± 26.5	1.0 ± 0.0	7
KornShell 12-02-29	161 758	39 004	24	5 762	1 526	6.1 ± 20.8	1.9 ± 5.8	2.2 ± 2.1	5.2 ± 9.4	1.2 ± 0.1	7
KornShell 12-08-01	162 657	39 220	24	5 804	1 537	6.1 ± 20.8	1.9 ± 5.9	2.2 ± 2.1	5.2 ± 9.4	1.2 ± 0.1	7
mDNSResponder 22	51 175	6 319	12	1 205	264	5.3 ± 9.0	1.7 ± 2.4	1.9 ± 1.7	5.2 ± 11.2	1.1 ± 0.1	4
mDNSResponder 541	88 311	26 881	30	1 758	362	5.3 ± 8.8	1.8 ± 2.4	1.9 ± 1.4	5.5 ± 11.1	1.1 ± 0.1	4
OpenSolaris 1.0	573 385	135 542	24	11 457	1 481	8.1 ± 30.6	3.2 ± 11.1	2.1 ± 2.2	4.5 ± 6.9	1.1 ± 0.0	8
OpenSolaris 13-10-28	2 547 718	505 522	20	48 193	7 628	9.2 ± 39.1	3.0 ± 12.1	1.7 ± 1.4	4.5 ± 13.9	1.1 ± 0.0	9
SeaMonkey 98-3-31	4 845 665	523 492	11	71 663	7 335	10.1 ± 71.9	4.2 ± 48.9	1.7 ± 1.4	3.2 ± 10.7	1.1 ± 0.0	8
SeaMonkey 2.23	8 659 794	1 500 270	17	93 484	8 992	10.9 ± 103.3	4.6 ± 63.3	1.7 ± 1.3	3.2 ± 9.0	1.1 ± 0.1	8
VirtualBox 1.6.0	1 151 572	390 732	34	39 506	6 094	9.6 ± 62.8	2.4 ± 6.4	2.5 ± 2.7	7.0 ± 37.4	1.2 ± 0.1	8
VirtualBox 4.3.2	3 239 837	980 027	30	125 952	9 275	12.1 ± 77.3	3.6 ± 17.3	2.8 ± 2.7	5.4 ± 24.5	1.1 ± 0.0	8
Industrial systems (IS)											
A	1 698 018	1 032 263	61	51 892	6 502	13.8 ± 69.4	2.1 ± 6.2	2.8 ± 2.0	3.9 ± 13.1	1.2 ± 0.2	8
B	1 686 414	1 101 868	65	42 370	7 761	10.6 ± 57.2	2.4 ± 10.6	2.9 ± 3.8	4.8 ± 12.6	1.4 ± 0.2	12
C	12 296	6 137	50	340	31	7.3 ± 14.6	2.8 ± 3.4	1.3 ± 0.5	3.2 ± 2.4	1.2 ± 0.1	3
D	42 151	15 965	38	2 728	413	6.9 ± 17.8	1.7 ± 2.4	1.8 ± 0.9	4.4 ± 7.3	1.2 ± 0.1	5
E	546 326	283 804	52	18 307	719	28.6 ± 307.9	6.0 ± 23.6	2.6 ± 0.7	3.9 ± 3.3	1.3 ± 0.1	5
F	274 196	171 225	62	12 285	1 454	10.2 ± 19.3	3.0 ± 4.3	2.4 ± 1.8	6.1 ± 11.0	1.4 ± 0.2	6
G	17 531	3 239	18	546	104	7.9 ± 14.6	2.6 ± 4.5	1.9 ± 0.9	3.3 ± 3.9	1.0 ± 0.0	5

LOC: lines of normalized code; LOF: lines of normalized CPP-annotated code; PLOF: fraction of CPP-annotated code (LOF/LOC); VP: number of variation points (`#ifdef` blocks); CC: number of configuration constants; $SD_{\#ifdef}$: average number of `#ifdef`s per CC; SD_{file}: average number of files per CC; $TD_{\#ifdef}$: average number of CCs per `#ifdef`; TD_{file}: average number of CCs per file; ND_{avg}: average nesting depth of `#ifdef`s; ND_{max}: maximum nesting depth of `#ifdef`s