Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

An Approach to Safely Evolve Preprocessor-Based C Program Families

Flávio Mota Medeiros

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Rohit Gheyi / Márcio Ribeiro
(Orientadores)

Campina Grande, Paraíba, Brasil
©Flávio Mota Medeiros, 02/02/2016

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M488a Medeiros, Flávio Mota.
An approach to safely evolve preprocessor - based C Program Families / Flávio Mota Medeiros. – Campina Grande, 2016.
178 f.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
"Orientação: Prof. Dr. Rohit Gheyi, Prof. Dr. Márcio Ribeiro".
Referências.

1. Pré-Processadores. 2. Sistemas Customizáveis. 3. Refatoramentos. I. Gheyi, Rohit. II. Ribeiro, Márcio. III. Título.

CDU 004.31(043)

Resumo

Desde os anos 70, o pré-processador C é amplamente utilizado na prática para adaptar sistemas para diferentes plataformas e cenários de aplicação. Na academia, no entanto, o pré-processador tem recebido fortes críticas desde o início dos anos 90. Os pesquisadores têm criticado a sua falta de modularidade, a sua propensão para introduzir erros sutis e sua ofuscação do código fonte. Para entender melhor os problemas de usar o pré-processador C, considerando a percepção dos desenvolvedores, realizamos 40 entrevistas e uma pesquisa entre 202 desenvolvedores. Descobrimos que os desenvolvedores lidam com três problemas comuns na prática: erros relacionados à configuração, testes combinatórios e compreensão do código. Os desenvolvedores agravam estes problemas ao usar diretivas não disciplinadas, as quais não respeitam a estrutura sintática do código. Para evoluir famílias de programas de forma segura, foram propostas duas estratégias para a detecção de erros relacionados à configuração e um conjunto de 14 refatoramentos para remover diretivas não disciplinadas. Para lidar melhor com a grande quantidade de configurações do código fonte, a primeira estratégia considera todo o conjunto de configurações do código fonte e a segunda estratégia utiliza amostragem. Para propor um algoritmo de amostragem adequado, foram comparados 10 algoritmos com relação ao esforço (número de configurações para testar) e capacidade de detecção de erros (número de erros detectados nas configurações da amostra). Com base nos resultados deste estudo, foi proposto um algoritmo de amostragem. Estudos empíricos foram realizados usando 40 sistemas C do mundo real. Detectamos 128 erros relacionados à configuração, enviamos 43 correções para erros ainda não corrigidos e os desenvolvedores aceitaram 65% das correções. Os resultados de nossa pesquisa mostram que a maioria dos desenvolvedores preferem usar a versão refatorada, ou seja, disciplinada do código fonte, ao invés do código original com as diretivas não disciplinadas. Além disso, os desenvolvedores aceitaram 21 (75%) das 28 sugestões enviadas para transformar diretivas não disciplinadas em disciplinadas. Nossa pesquisa apresenta resultados úteis para desenvolvedores de código C durante suas tarefas de desenvolvimento, contribuindo para minimizar o número de erros relacionados à configuração, melhorar a compreensão e a manutenção do código fonte e orientar os desenvolvedores para realizar testes combinatórios.

Abstract

Since the 70s, the C preprocessor is still widely used in practice in a numbers of projects, including *Apache*, *Linux*, and *Libssh*, to tailor systems to different platforms and application scenarios. In academia, however, the preprocessor has received strong criticism since at least the early 90s. Researchers have criticized its lack of separation of concerns, its proneness to introduce subtle errors, and its obfuscation of the source code. To better understand the problems of using the C preprocessor, taking the perception of developers into account, we conducted 40 interviews and a survey among 202 developers. We found that developers deal with three common problems in practice: configuration-related bugs, combinatorial testing, and code comprehension. Developers aggravate these problems when using undisciplined directives (i.e., bad smells regarding preprocessor use), which are preprocessor directives that do not respect the syntactic structure of the source code. To safely evolve preprocessor-based program families, we proposed strategies to detect configuration-related bugs and bad smells, and a set of 14 refactorings to remove bad smells. To better deal with exponential configuration spaces, our strategies uses variability-aware analysis that considers the entire set of possible configurations, and sampling, which allows to reuse C tools that consider only one configuration at a time to detect bugs. To propose a suitable sampling algorithm, we compared 10 algorithms with respect to effort (i.e., number of configurations to test) and bug-detection capabilities (i.e., number of bugs detected in the sampled configurations). Based on the results, we proposed a sampling algorithm with an useful balance between effort and bug-detection capability. We performed empirical studies using a corpus of 40 C real-world systems. We detected 128 configuration-related bugs, submitted 43 patches to fix bugs not fixed yet, and developers accepted 65% of the patches. The results of our survey show that most developers prefer to use the refactored (i.e., disciplined) version of the code instead of the original code with undisciplined directives. Furthermore, developers accepted 21 (75%) out of 28 patches submitted to refactor undisciplined into disciplined directives. Our work presents useful findings for C developers during their development tasks, contributing to minimize the chances of introducing configuration-related bugs and bad smells, improve code comprehension, and guide developers to perform combinatorial testing.

Contents

1 **Introduction** 1
1.1 Problem Statement 1
1.2 Motivating Examples 4
1.3 Solution 6
1.4 Evaluation 7
1.5 Summary of Contributions 9
1.6 Organization of this Thesis 10

2 **Background** 11
2.1 Program Families and Software Product Lines 11
2.2 The C Preprocessor 13
2.2.1 Configuration 13
2.2.2 Undisciplined and Disciplined Directives 15
2.2.3 Configuration-Related Bugs 16
2.3 Variability-Aware Analysis 17
2.4 Sampling Analysis 19
2.5 Static Analysis Tools for C 21
2.5.1 Cppcheck 23
2.6 Refactoring 24

3 **Problem Dimension** 27
3.1 Challenges Induced by the C Preprocessor 27
3.1.1 Readability and Separation of Concerns 29
3.1.2 Combinatorial Explosion and Parsing Unpreprocessed C Code . . . 29

3.1.3 Error Proneness and Guidelines 30
3.1.4 Difficulty to Develop Tool Support and Syntactic Preprocessors . . 31
3.2 Research Study 31
3.2.1 Overall Study Design 32
3.2.2 Results and Discussion 36
3.2.3 Threats to Validity 51

4 A Sampling-Based Strategy to Detect Configuration-Related Bugs 52
4.1 The Sampling-Based Strategy 52
4.2 Comparison of Sampling Algorithms 54
4.2.1 Overall Study Design 55
4.2.2 Results and Discussion 61
4.2.3 Threats to Validity 73
4.3 The Linear Sampling Algorithm 75
4.4 Research Study 76
4.4.1 Overall Study Design 76
4.4.2 Results and Discussion 78
4.4.3 Threats to Validity 89

5 A Variability-Aware Strategy to Detect Configuration-Related Bugs 91
5.1 The Variability-Aware Strategy 91
5.1.1 Stubs 92
5.1.2 Platform-Specific Headers 94
5.2 Research Study 96
5.2.1 Overall Study Design 96
5.2.2 Results and Discussion 98
5.2.3 Threats to Validity 109

6 Catalog of Refactorings 112
6.1 Refactorings 112
6.1.1 Single Statements 113
6.1.2 Conditions 114

6.1.3 Wrappers 114
6.1.4 Comma-Separated Elements 115
6.2 Evaluation 116
6.2.1 Overall Study Design 117
6.2.2 Results and Discussion 117
6.2.3 Threats to Validity 129

7 Tool Support: Colligens **131**
7.1 Macro Constraints Integration 131
7.2 Variability-Aware Analysis 132
7.3 Sampling-Based Analysis 134
7.4 Detecting and Removing Bad Smells 135

8 Related Work **136**
8.1 Analysis of C Preprocessor Usage 136
8.2 Static Analysis to Find Bugs 137
8.3 Variability-Aware Analysis 138
8.4 Sampling Analysis 139
8.5 Extracting Variability Information 140
8.6 Refactoring Program Families 141

9 Concluding Remarks **144**
9.1 Review of the Contributions 145
9.2 Future Work 148

A The Complete Catalog of Refactorings **168**

B The C Model **175**

List of Figures

1.1 Code snippet of *CVS* that causes a compilation error. 4
1.2 Code snippet of *Bash* with unexpected behavior. 5
1.3 Code snippet of *Xterm* with bad smells. 6

2.1 The activities of software product line engineering. 12
2.2 A program family with four configurations. 14
2.3 Code snippet of *Vim* with undisciplined directives. 15
2.4 Refactored code of *Vim* including disciplined directives only. 15
2.5 Code snippet of *Gawk* with a memory leak. 16
2.6 Code snippet of *Libpng* with a syntax error. 17
2.7 Performing partial preprocessing. 18
2.8 Abstract syntax tree enhanced with variability information. 18
2.9 Comparing the sampling algorithms by example. 20
2.10 Sample sets of *t-wise* sampling. 20
2.11 Code snippet to discuss strategies of static analysis tools. 23
2.12 Code snippet of *Vim* to show how *Cppcheck* selects configurations. 24
2.13 Wrong code transformation that introduces a compilation error. 25
2.14 Refactoring to remove undisciplined directives by cloning code. 26

3.1 The interacting tasks of the C preprocessor. 28
3.2 Real code snippets taken from *Vim* with undisciplined directives. 29
3.3 Using function-like macros to avoid code duplication. 39
3.4 Preprocessor directives or portability functions. 41
3.5 Conditional compilation or run-time checks. 43
3.6 Results of our survey to quantify some findings of our interviews. 45

3.7 Disciplining a preprocessor directive. 49
3.8 Avoiding code clone and compiler warnings. 51

4.1 A sampling-based strategy to detect configuration-related bugs. 53
4.2 Analyzing different versions of the projects. 54
4.3 Strategy used to compare the sampling algorithms. 58
4.4 Number of distinct preprocessor macros in files with bugs. 59
4.5 Number of bugs and samples per file for each algorithm. 63
4.6 Number of bugs and samples per file for the combination of algorithms. . . 64
4.7 Number of warnings reported and samples per file for each algorithm. . . . 71
4.8 Number of warnings and samples per file for the combinations of algorithms. 72
4.9 Selecting configurations systematically with *LSA*. 76
4.10 Types of configuration-related bugs. 80
4.11 An example of configuration-related uninitialized variable in *Bash*. 81
4.12 Number of days developers take to fix bugs that appear in all configurations. 82
4.13 Number of days developers take to fix configuration-related bugs. 83
4.14 Introducing configuration-related memory leaks. 84
4.15 Introducing configuration-related resource leaks. 85
4.16 Introducing configuration-related uninitialized variables. 85
4.17 Introducing configuration-related null pointer dereferences. 86
4.18 Introducing configuration-related buffer overflows. 87

5.1 Strategy to detect bugs using stubs. 93
5.2 Strategy to detect bugs using platform-specific headers. 95
5.3 Generating platform-specific headers for *Linux*. 95
5.4 Code snippet of *Vim* with a syntax error. 100
5.5 An undeclared variable in *Libpng*. 100
5.6 An unused variable in *Libssh*. 101
5.7 Introducing configuration-related syntax errors. 103
5.8 Introducing configuration-related undeclared functions. 105
5.9 Introducing configuration-related undeclared variables. 105
5.10 Introducing configuration-related unused functions. 106

5.11 Introducing configuration-related unused variables. 106

6.1 Duplicating tokens to discipline preprocessor directives. 120
6.2 Adding a local variable to discipline preprocessor directives. 121
6.3 Using macros to discipline preprocessor directives. 121
6.4 Applying regression testing to verify behavior preservation. 125
6.5 Examples of generated and refactored programs. 126
6.6 Undisciplined `if` condition that introduced behavioral changes. 127

7.1 Code snippet of *Libssh* and its macro constraints. 132
7.2 *Colligens* view to set configuration parameters. 133
7.3 *Colligens* view to present bugs detected by using variability-aware analysis. 134
7.4 *Colligens* view to present bugs detected by using sampling. 134
7.5 *Colligens* view to refactor undisciplined directives. 135

B.1 Program generated with possibility to apply our refactoring. 178

List of Tables

2.1 Tools to perform static analysis in C. 22

3.1 General information about projects repositories. 33

4.1 Configuration-related bugs considered in our first study. 59
4.2 Project characterization and the total number of known bugs. 60
4.3 Presence conditions of the configuration-related bugs. 61
4.4 Number of bugs, size of sample sets and ranking of algorithms. 74
4.5 Overview of the subject projects and the total number of bugs. 79
4.6 Occurrences of configuration-related memory leaks. 84
4.7 Occurrences of configuration-related resource leaks. 85
4.8 Occurrences of configuration-related uninitialized variables. 86
4.9 Occurrences of configuration-related null pointer dereferences. 86
4.10 Patches submitted to subject systems. 89

5.1 Subject characterization and number of bugs 99
5.2 Preprocessor macros involved in bugs. 102
5.3 Occurrences of configuration-related syntax errors. 104
5.4 Occurrences of configuration-related undeclared functions. 104
5.5 Time to fix configuration-related bugs. 108
5.6 Patches submitted to subject systems. 110

6.1 Subject characterization 113
6.2 Application possibilities in 63 C projects. 119
6.3 Patches accepted after minor changes. 123
6.4 Patches rejected. 123

6.5 Results of behavioral changes regarding the generated families. 126
6.6 Results of testing on *BusyBox*, *OpenSSL*, and *SQLite*. 128
6.7 Subject characterization 129

Chapter 1

Introduction

The C preprocessor is a simple, effective, and language independent tool to transform the source code before compilation, but it provides no perceptible form of modularity [1]. Developers frequently use the C preprocessor to develop infrastructure software like operating systems, e.g., *Linux* and *FreeBSD*, security protocols, such as *Libssh*, and web servers like *Apache* and *Cherokee*. Infrastructure software is critical, and requires configurability to run on different platforms and high quality software artifacts to minimize the chances of financial losses due to software bugs.

The preprocessor is still widely used in industry and practice to implement program families [2; 3; 4]. A program family is a set of programs whose commonality is so extensive that it is advantageous to study their common properties before analyzing individual programs [5]. In this context, developers use preprocessor conditional directives, such as `#ifdef`, `#else`, and `#endif`, to mark parts of the source code as optional, with the purpose of tailoring software systems to different hardware platforms, operating systems, and application scenarios. However, by coding with preprocessor directives, developers deal with two independent languages, which hinders code understanding, maintainability, and the development of tool support.

1.1 Problem Statement

Despite the widespread use of the C preprocessor, it has received strong criticism since at least the early 90s. Researchers have criticized its lack of separation of concerns [6; 7; 8; 9; 10], its proneness to introduce subtle errors [2; 11; 7; 4; 12; 13], and its obfuscation of the

source code [13; 3; 6; 14; 15]. Many studies have found bugs related to preprocessor use [16; 12; 17; 18; 19; 20]. Additionally, its complexity hinders tool support available in other languages, such as automated refactoring [21; 17; 22; 23; 24; 25].

The C preprocessor essentially has not changed since the 70s. Researchers have proposed several alternatives to preprocessor directives, e.g., syntactical preprocessors [26; 23; 27], aspect-oriented programming [15; 28], and various forms of metaprogramming. However, for the best of our knowledge, such alternatives have not been adopted in practice.

To better understand the C preprocessor challenges, and its widespread use in practice despite all criticism and alternatives, we conducted 40 interviews and a survey among 202 developers. We found that developers have a love/hate relationship with the C preprocessor and do not see any current technologies that can entirely replace the preprocessor [29]. Many developers see the preprocessor as an elegant solution to workaround portability problems. However, developers are aware that they must follow code guidelines strictly to avoid three common problems of the preprocessor: (1) configuration-related bugs, which are perceived as more critical than other bugs, (2) combinatorial testing, as conditional directives increase the number of configurations to check for quality-assurance, and (3) code comprehension, due to the cluttering of `#ifdefs` and C statements [29].

Developers aggravate these problems when using undisciplined directives that do not respect the syntactic structure of the source code, for example, wrapping a single bracket without its corresponding closing one [3; 13; 30; 4]. Undisciplined directives influence code understanding, maintainability, and error proneness negatively [29; 13; 3; 4]. Although some tools could enforce such guidelines [4; 31; 13; 20], research studies show that guidelines are not followed strictly in practice [3; 4; 29]. The guidelines on coding style of the *Linux Kernel*, for example, guide developers explicitly to avoid undisciplined directives, saying: "*prefer to compile out entire functions, rather than portions of functions or portions of expressions. Rather than putting an `#ifdef` in an expression, factor out part or all of the expression into a separate helper function and apply the conditional to that function.*" Some researchers have proposed refactorings to convert undisciplined into disciplined directives, however, these refactorings clone code [30; 32], which also impacts code quality negatively [33].

Besides, the vast majority of mature quality-assurance C development tools consider only a single configuration at a time. For example, state-of-the-art tools, such as *Gcc*, *Clang*,

Eclipse, *Xcode*, and *NetBeans*, operate typically on C code after the C preprocessor has resolved variability implemented through conditional compilation (e.g., implemented with `#ifdef` directives). To reuse these mature C development tools to detect configuration-related bugs, *sampling* is a viable alternative [34; 35; 36; 37; 20]. That is, instead of analyzing all configurations, one selects a subset of configurations to analyze individually. However, the effectiveness of sampling for detecting configuration-related bugs depends significantly on how samples are selected. In this context, there is a gap of studies comparing sampling algorithms with regards to their efficiency to detect bugs. In the research literature, there are some tools with support to deal with variability in C. For instance, *TypeChef* [17] and *SuperC* [38], variability-aware parsers for C code, which analyze complete configuration spaces. However, they require a time-consuming setup to analyze all dependencies defined through `#include` directives.

Due to the complexities of dealing with variability in C and without an appropriate tool support, developers have problems when evolving C program families, e.g., introducing bugs [13; 3; 12; 39] and bad smells [33; 29] related to preprocessor directives. Furthermore, developers introduce bugs and bad smells that appear in software repositories like *Git* [40], such as uninitialized variables, undefined functions, and other compilation errors.[1] This way, as these problems are difficult to detect due to variability [29], they also appear in the projects releases [12; 39], which may impact time-to-market, software quality, and lead to problems like financial losses.

In summary, we focus on the following three problems:

1. Configuration-related syntax errors, bugs, and warnings that we can detect by performing static analysis, such as undeclared and unused variables and functions, memory and resource leaks, dereference of null pointers, and uninitialized variables;

2. Code comprehension with regards to the use of undisciplined directives;

3. Combinatorial testing, as preprocessor conditional directives increase the number of configurations to check for quality-assurance.

[1] `https://bugzilla.gnome.org/show_bug.cgi?id=580750`, 445140, 309748, and 461011.

1.2 Motivating Examples

To clarify the problems we address in this work, we present some motivating examples in what follows. For instance, Figure 1.1 (a) presents a syntax error in the *CVS*[2] project when we enable, for example, macros SHUTDOWN, SOCKET, and POPEN. After preprocessing this code snippet, we generate an invalid program, see Figure 1.1 (b). When compiling this program, traditional C compilers (e.g., *Gcc* and *Clang*) report a compilation error, as we have an else if just after an if statement. However, when compiling the code snippet presented in Figure 1.1 (a), compilers report no syntax errors or warnings when we enable macros SHUTDOWN and POPEN, and disable SOCKET. Notice, though, that the syntax error actually exists, but in another configuration, as it is a configuration-related syntax error.

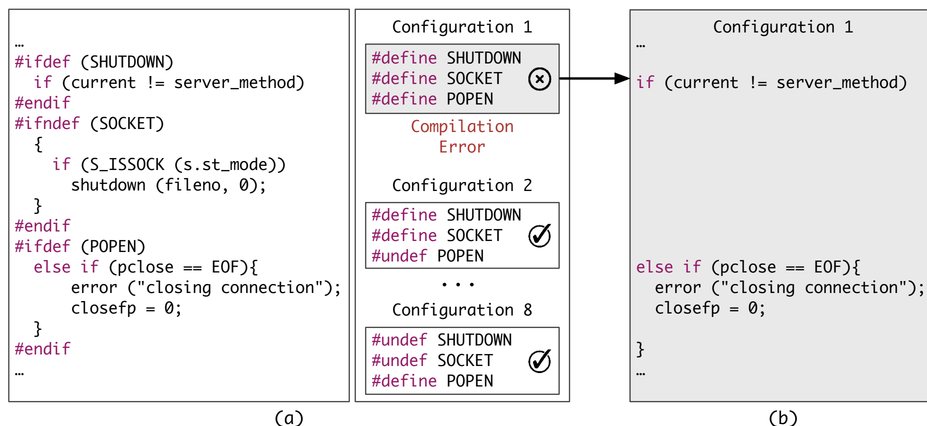


Figure 1.1: Code snippet of *CVS* that causes a compilation error.

As another example, Figure 1.2 (a) presents a code snippet of the *Bash*[3] project with unexpected behavior when developers disable macros TRACE and REGISTER, and enable macro WATCH. As we can see in Figure 1.2 (b), variable ubytes is not initialized, but it is used at Line 15. Technically, the value of an uninitialized, non-static, local variable is indeterminate in C, and accessing it leads to an undefined behavior [41]. Developers can use traditional C tools (e.g., *Gcc*) to detect this uninitialized variable, but it is not guaranteed. These tools preprocess the code to generate each configuration and check these configurations individually. So, these tools might not detect this uninitialized variable, because it

[2] http://www.nongnu.org/cvs/
[3] https://www.gnu.org/software/bash/

appears only in some configurations of the code. In this context, developers face a problem of selecting which configurations they check (i.e., combinatorial testing), specially because the space of possible configurations is exponential, in the worst case, and it is usually too large to explore exhaustively. Assuming *n* optional and independent configuration options, the number of configurations is 2^n. In the *Linux Kernel*, for example, there are more than 12K configuration options.

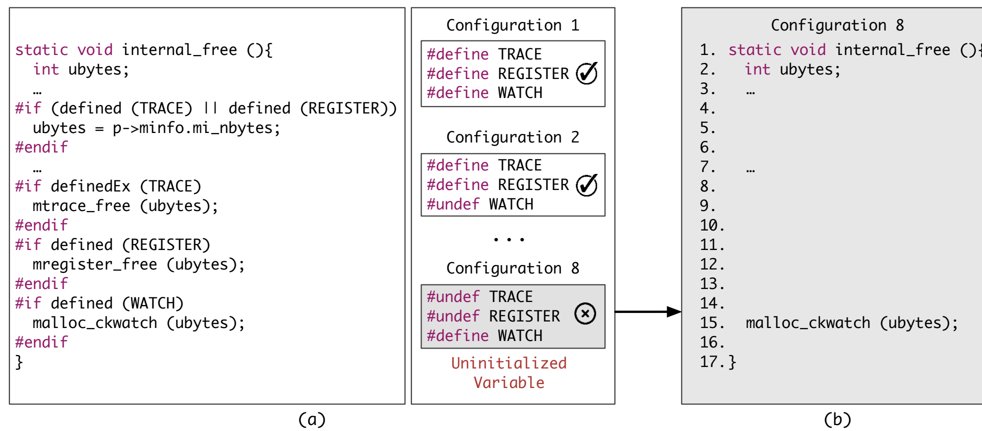


Figure 1.2: Code snippet of *Bash* with unexpected behavior.

Besides syntax errors and undefined behavior that appear only in some configurations of the source code, developers can also introduce bad smells. Figure 1.3 presents a code snippet of *Xterm*[4] that contains undisciplined directives. As we can see, the developers of *Xterm* encompass only a closing bracket with preprocessor directives (see Line 21). In this work, we consider undisciplined directives as bad smells [33] related to preprocessor directives, because undisciplined directives influence code quality negatively, making the tasks of reading and understanding the source code more difficult [13; 3; 4].

Developers may need more time to understand the code snippet of Figure 1.3 (a), e.g., to detect where `if` statements end, or to analyze whether opening and closing brackets match correctly. Furthermore, undisciplined directives leave the source code more conducive to introduce syntax errors [12]. In this code snippet, for example, there is a syntax problem but in invalid configurations, such as when we enable macros `GLIBC` and `PTSFLAG`, as presented in Figure 1.3 (b). By setting this configuration, developers introduce an extra bracket at Line 12. Thus, they may still need more time to detect that this configuration is invalid since the source code does not contain this information explicitly.

[4] http://invisible-island.net/xterm/

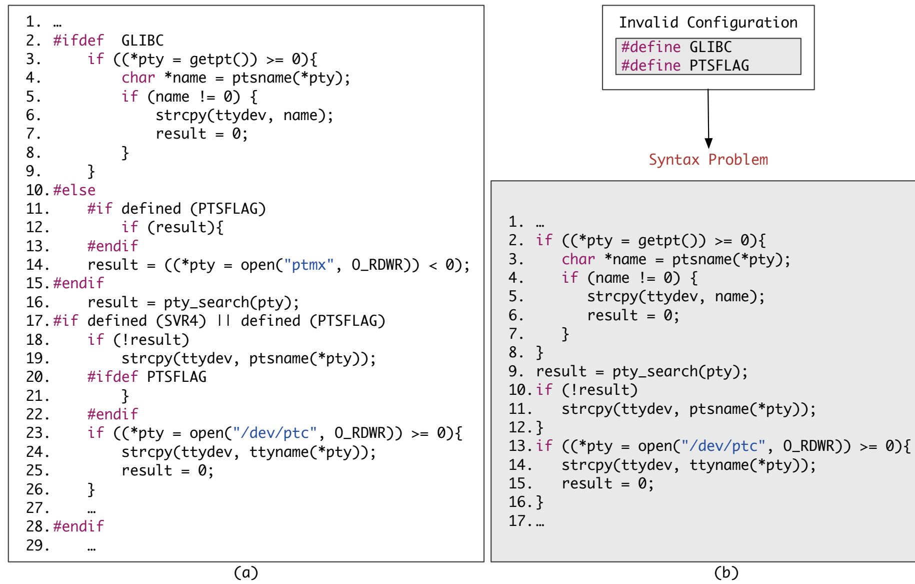


Figure 1.3: Code snippet of *Xterm* with bad smells.

Bugs like undefined behavior may cause security problems and financial losses. In this context, developers need better tool support to develop and evolve program families, and to minimize configuration-related bugs. We can support C developers in distinct ways, such as using a defective or corrective strategy, in which the main focus is finding bugs. Also, we can apply a perfective or preventive solution, which focuses on improving code quality with the purpose of avoiding bugs in the future, and making the tasks of reading and understanding the code faster [42]. In addition, studies to investigate the use of the C preprocessor in real projects are helpful to understand common problems that happen in practice, provide insights for better development processes, and minimize chances of introducing subtle bugs [13; 3; 12; 39; 29; 43].

1.3 Solution

To minimize the aforementioned problems, this study proposes an approach to safely evolve C program families. To support defective evolution, which focuses on detecting exiting problems [42], we defined two strategies to detect configuration-related bugs. These strategies consider different types of bugs, such as syntax errors, type errors, memory leaks, resource leaks, dereferences of null pointers, and uninitialized variables.

The first strategy applies variability-aware analysis. It uses a variability-aware parser to generate abstract syntax trees enhanced with variability information and performs static analysis to detect configuration-related bugs. Our first strategy uses some simplifications to avoid the time-consuming setup of variability-aware tools and to make the analysis of several projects feasible, such as the use of stubs to eliminate the complexities of dealing with `#include` directives.

The second strategy uses sampling, which allows us to reuse traditional C tools, such as *Gcc*, *Clang*, and *Cppcheck*, to check one configuration at a time. The efficiency of our second strategy depends significantly on how we select samples. This way, we performed a study to compare a number of sampling algorithms, guiding developers to perform exponential testing. Based on the results of this study, we propose the *Linear Sampling Algorithm (LSA)*, which provides an useful balance between effort (i.e., number of configurations to test) and bug-detection capability (i.e., number of bugs detected in the sampled configurations).

To support perfective evolution, which focuses on improving code quality [42], we defined a catalog of refactorings to make the source code less conducive to introduce bugs, and improve code readability. Our refactorings are transformation templates, and each refactoring is an unidirectional transformation satisfying specific preconditions. Furthermore, our catalog of refactorings removes undisciplined directives without cloning code, different from previous studies [30; 4; 32]. Thus, developers do not need to decide whether to keep undisciplined directives or to introduce code clone.

Finally, we developed a supporting tool named *Colligens* to implement the strategies to detect configuration-related bugs and to apply the catalog of refactorings automatically. By using *Colligens*, developers gain the benefits of an integrated, sampling-based, and variability-aware environment to develop program families in C.

1.4 Evaluation

To evaluate our strategies and the catalog of refactorings, we used a corpus of 63 C open-source projects. Our corpus includes projects of different sizes, ranging from 2 thousand to 7 million lines of code, including projects from different domains, such as web servers, databases, diagramming software, lexical analysers, text editors, and file compressors.

To evaluate our support for defective evolution, we instantiated the sampling-based and the variability-aware strategies. To select a suitable sampling algorithm, we conducted a comparative study to analyze sampling algorithms and understand the tradeoffs, especially with regard to effort and bug-detection capabilities. We analyzed 10 sampling algorithms and 35 combinations of these sampling algorithms in a study of 135 known configuration-related bugs in 24 projects of our corpus. The results motivated us to instantiate the sampling-based strategy using *LSA*. We also used *TypeChef* to generate abstract syntax trees with variability information, and *Cppcheck*, a static analysis tool that developers have been using in many popular projects to detect various kinds of bugs, including memory leaks, uninitialized variables, and dereference of null pointers. In addition, developers of *Cppcheck* claim to minimize false positives.

By applying the sampling-based strategy using *LSA* and *Cppcheck*, we detected 34 memory leaks, 12 uninitialized variables, 11 dereferences of null pointers, 6 resource leaks, and 2 buffer overflows. By using *TypeChef* in our variability-aware strategy, we detected 24 syntax errors, 14 undeclared functions, 2 undeclared variables, 7 unused functions, and 16 unused variables. Overall, we detected 128 configuration-related bugs, submitted 43 patches to fix the configuration-related bugs not fixed by developers, and 28 (65%) patches were accepted.

Our empirical study presents findings to aid developers during their development tasks, such as examples of common configuration-related bugs, and analyses of how developers introduce these bugs in practice. The results show that configuration-related bugs remain longer in the source code than bugs that appear in all configurations. The variability of program families hide configuration-related bugs, hindering the detection of such bugs. We found that the majority of configuration-related bugs involve two or less preprocessor macros, which support the effectiveness of sampling algorithms, such as *pair-wise* [44; 45], and *LSA*. Furthermore, the results show that configuration-related bugs appear as frequent as bugs that occur in all configurations of the source code, giving evidence that bugs are equality distributed across different configurations.

We evaluated our catalog of refactorings regarding frequency of application possibilities in practice, opinion of developers, behavior preservation, and quality of the refactored code. We found 5670 application possibilities for our refactorings in 63 real-world projects, showing many opportunities to apply the refactorings. With regards to the opinion of de-

velopers, we found by using our survey among 202 developers that most participants prefer to use the refactored (i.e., disciplined) version of the source code instead of the original source code with undisciplined directives. Furthermore, developers accepted 21 (75%) out of 28 patches that we submitted converting undisciplined into disciplined directives. To check that our refactorings are behavior preserving, we applied the refactorings to more than 36 thousand programs generated automatically using a formal model as well as in three real-world projects: *BusyBox*, *OpenSSL*, and *SQLite*. By using regression testing [46; 47], we detected and fixed a few behavioral changes introduced by our refactorings, the majority caused by unspecified behavior in the C language, but also problems in the implementation of the catalog of refactorings. Last, we removed 447 undisciplined preprocessor directives of 12 real-world systems, such as *Apache* and *Ghostscript*, without cloning code, different from previous work [32; 30; 4].

1.5 Summary of Contributions

In summary, the main contributions of this thesis are:

- An interview study to understand how developers perceive the C preprocessor and complimentary studies (literature review, online survey, and repository analysis) to cross-validate and to quantify the results [29];
- A comparison of sampling algorithms for program families with regards to effort and bug-detection capability. Based on the results of our comparative study, we proposed the Linear Sampling Algorithm (LSA) [43];
- An empirical study to investigate and to quantify configuration-related bugs using real C projects [12; 39; 48; 49];
- Two strategies to identify configuration-related bugs in C projects using sampling and variability-aware analysis [12; 39];
- A catalog of refactorings to remove bad smells in preprocessor directives [50];
- A supporting tool named *Colligens* that automatizes our strategies to detect configuration-related bugs and applies our catalog of refactorings automatically [51].

1.6 Organization of this Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we present background information about the main concepts used in this thesis. In Chapter 3, we describe the problem we address in this study. In Chapter 4, we present the sampling-based strategy to detect bugs, and in Chapter 5, we present the variability-aware strategy. Chapter 6 presents our catalog of refactorings to remove bad smells in preprocessor directives, and Chapter 7 presents our supporting tool. Finally, we discuss the related work in Chapter 8, and present the concluding remarks in Chapter 9.

Chapter 2

Background

In this chapter, we present a brief overview of the main concepts used in this thesis. In Section 2.1, we discuss program family and software product line concepts. Section 2.2 presents information about the C preprocessor, including the definition of configuration, configuration spaces, configuration-related bugs, and undisciplined directives. In Section 2.3, we present concepts of variability-aware analysis, and Section 2.4 discusses sampling-based analysis. Section 2.5 considers concepts and tools to perform static analysis, and Section 2.6 discusses refactorings in C program families.

2.1 Program Families and Software Product Lines

A program family is a set of programs whose commonality is so extensive that it is advantageous to study their common properties before analyzing individual family members [5]. In this context, individual family members may have different functionalities, or the same functionalities implemented differently according to specific operating systems and platform characteristics [2]. The concept of program families is similar to Software Product Lines (SPL) [52; 53]. However, the latter is more systematic and uses some concepts, theories, and artifacts that are not necessarily used in program families, e.g., feature model [54; 55; 56] and configuration knowledge [57; 58].

Software product line engineering has its principles based on automobile manufactures, which enable mass production cheaper than individual product creation. These manufactures use a common platform to derive products that can be customized to specific customers or

market segments needs [52]. In the context of software engineering, the combination of mass customization, large-scale production, and the use of a common platform to derive products results in the software product line engineering paradigm [53].

A product line is a set of similar software intensive systems that share a collection of common features satisfying the needs of specific customers or market segments. This set of systems are developed from a set of core assets, which are documents, specifications, components, and other software artifacts that naturally become highly reusable during the development of each specific system in the product line [59; 60; 61; 53].

In this sense, the software product line development paradigm uses a systematic and planned reuse strategy, which is presented in Figure 2.1 and explained in what follows [52]:

- **Core asset development:** In this activity, a set of core assets, a product line scope, and a production plan are produced. The core assets form the basis of the product line and its production capability;
- **The product development** activity receives as input the outputs of the core asset development, and a product-specific requirement. The product required is developed using the core assets developed previously;
- **Management** is necessary because core asset development and product development activities are iterative, and this iteration must be carefully managed.

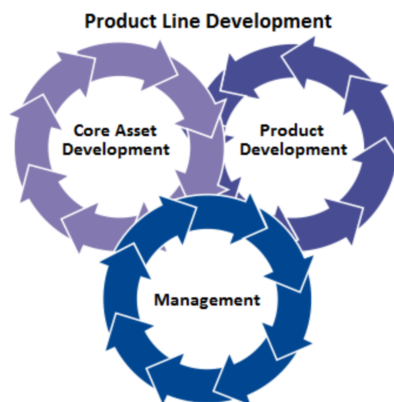


Figure 2.1: The activities of software product line engineering.

Regarding the C language, developers often use the C preprocessor to handle variability, solve portability problems, and implement individual family members [2], i.e., developers encompass C source code with preprocessor directives, such as `#ifdef`, `#else`, `#elif`, and `#endif`. However, real C program families do not necessarily use the concepts and artifacts of SPL, and their development is not always systematic. Thus, in this study, we use the term program family to reference projects, such as *Apache* and *Libssh*, which use the C preprocessor to handle variability and portability. The next section presents an overview about the C preprocessor.

2.2 The C Preprocessor

The C preprocessor is a language-independent tool for lightweight meta-programming that fills a need, among others, for portability and variability. The preprocessor is widely used in practice. It is essentially used in all projects written in C, including many well-known databases and operating systems. The C preprocessor essentially has not changed since the 70s and it is used automatically by C compilers to transform programs before compilation. The preprocessor is executed during the compilation process and performs three interacting tasks:

- It lexically includes files (`#include`);
- It expands macros (defined with `#define`); and
- It conditionally excludes part of the source code depending on which and how macros are defined (such as `#ifdef` and `#if`).

In this study, we focus on conditional compilation, because file inclusion and macro expansions are relatively well understood and there are mitigation strategies available in the literature [62; 63; 3; 64; 65].

2.2.1 Configuration

By using the C preprocessor, developers deal with a single code with different configurations. A configuration is an assignment of values `true` or `false` for all preprocessor macros

used in the source code. The `true` value means that the preprocessor macro is enabled, and `false` means disabled. For instance, Figure 2.2 presents a code snippet of a program family with four configurations: (1) macros `A` and `B` enabled, (2) macro `A` disabled and macro `B` enabled, (3) macro `A` enabled and macro `B` disabled, and (4) both macros disabled. By preprocessing this code snippet, developers can generate these four configurations. To set each specific configuration, we use `#define` and `#undef` directives to enable and disable macros respectively. These four configurations depicted in Figure 2.2 form the configuration space of the program family. In real-world projects, the configuration spaces are usually very large, which make the analysis of every individually configuration infeasible.

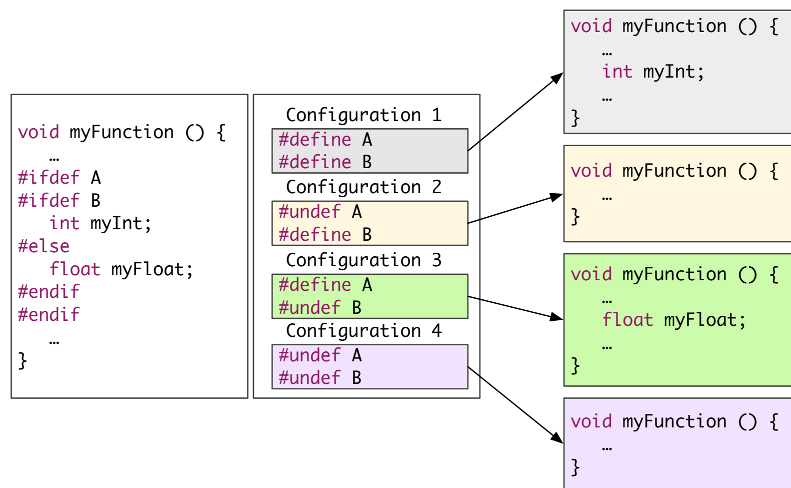


Figure 2.2: A program family with four configurations.

Despite being native of the C language, we can use the preprocessor to transform any text file. The preprocessor has no knowledge about the C language constructors, as it is a lexical preprocessor. For this reason, we can encompass any code with preprocessor directives, such as an opening bracket or a comma. This way, we can handle both fine-grained as well as coarse-grained variability with the C preprocessor [1; 66]. Regarding the way that developers encompass C source code with preprocessor directives, we can classify a preprocessor directive as disciplined or undisciplined [67; 17; 30], as presented in the next section.

2.2.2 Undisciplined and Disciplined Directives

Undisciplined preprocessor directives do not respect the syntactic structure of the source code (e.g., wrapping a single bracket without its correspondent closing one) [4]. For instance, Figure 2.3 presents part of the source code of *Vim* including undisciplined directives. Undisciplined directives split up part of C syntactical units, e.g., the `#ifdef` directive that starts at Line 2 and ends at Line 4 surrounds only part of the statement condition.

```
1. if (msec > 0
2. #ifdef USE_XSMP
3.   && xsmp_icefd != -1
4. #endif
5. ){
6.   // lines of code
7.   gettime(&start_tv);
8. }
```

Figure 2.3: Code snippet of *Vim* with undisciplined directives.

Disciplined directives encompass complete C syntactical units only, such as a function definition, variable declaration, and a function call [30; 67; 4]. Figure 2.4 presents a code snippet with disciplined directives only (an equivalent and disciplined version of the code presented in Figure 2.3). In Figure 2.4, the `#ifdef` directive that starts at Line 2 and ends at Line 4 is disciplined and it surrounds a complete variable attribution.

```
1. bool time = msec > 0;
2. #ifdef USE_XSMP
3.   time = time && xsmp_icefd != -1;
4. #endif
5. if (time){
6.   // lines of code
7.   gettime(&start_tv);
8. }
```

Figure 2.4: Refactored code of *Vim* including disciplined directives only.

The lexical operation mode, which allows developers to introduce undisciplined directives, is one of the most criticized aspects of the C preprocessor [3; 13; 30; 4; 29; 12; 39]. Prior studies criticise undisciplined directives due to its negative influence on code quality, maintainability, and error-proneness [3; 68; 4], we present more detail in Chapter 3. Thus, we consider that undisciplined directives are always bad smells with regards to preprocessor usage [33], which may lead to configuration-related bugs, as discussed next.

2.2.3 Configuration-Related Bugs

A configuration-related bug is an error related to the use of preprocessor directives. In this work, we consider an error as a result different from the expected, or an incorrect step, process or data definition that may lead to an error, e.g., a compilation error, or an out of memory error [69].

We define a configuration-related bug in the following way: A configuration-related bug is an error that does not occur in all configurations of the source code, that is, to decide whether a bug is a configuration-related bug, we need to identify at least one valid configuration where the bug appears and at least one valid configuration that the bug is not present. Valid configurations take the macro constraints into account.

To illustrate configuration-related bugs, Figure 2.5 presents a code snippet of the *Gawk*[1] source code related to its regular expression library. This code snippet contains a preprocessor macro that implements localization and language internationalization, i.e., `I18N`, which is responsible to adapt the software to specific regions or languages. By preprocessing the code snippet presented in Figure 2.5 with `I18N` enabled, developers generate a memory leak. We allocate memory to variable *mbcset* at Line 12, but we do not deallocate it when returning `NULL` at Line 21.

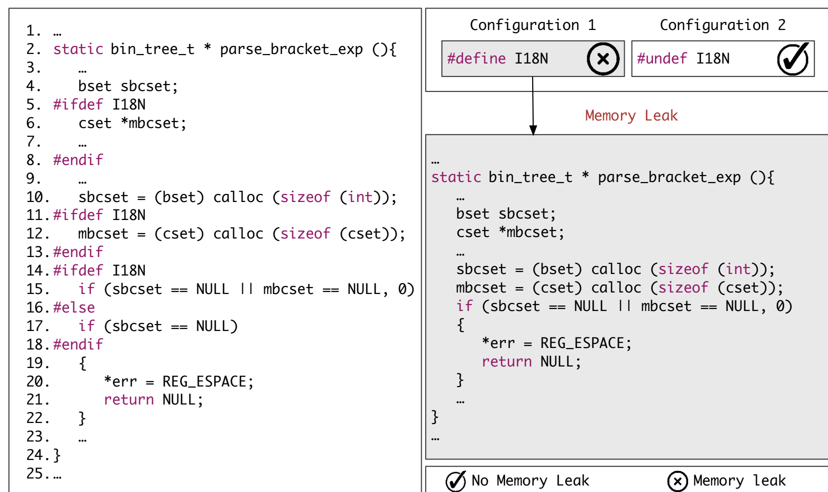


Figure 2.5: Code snippet of *Gawk* with a memory leak.

[1] http://www.gnu.org/software/gawk/

Presence Condition

By definition, configuration-related bugs are not present in all configurations of the source code. In this sense, a configuration-related bug occurs in a subset of valid configurations only. The presence condition of a configuration-related bug is a boolean expression that represents this subset of valid configurations. For instance, the configuration-related bug of *Gawk*, presented in Figure 2.5, occurs only when we enable macro `I18N`. This way, the presence condition of this bugs is `I18N`.

As another example, we use a configuration-related bug in *Libpng*,[2] as presented in Figure 2.6. By preprocessing this code snippet without macro `INTERLACING`, we generate an invalid program according to the C grammar. It contains a syntax error since it opens the `if` statement block at Line 4, but it does not close at Line 14. In contrast, if macro `INTERLACING` is enabled, there is no syntax error. This way, the presence condition of this configuration-related bug of *Libpng* is ¬`INTERLACING`.

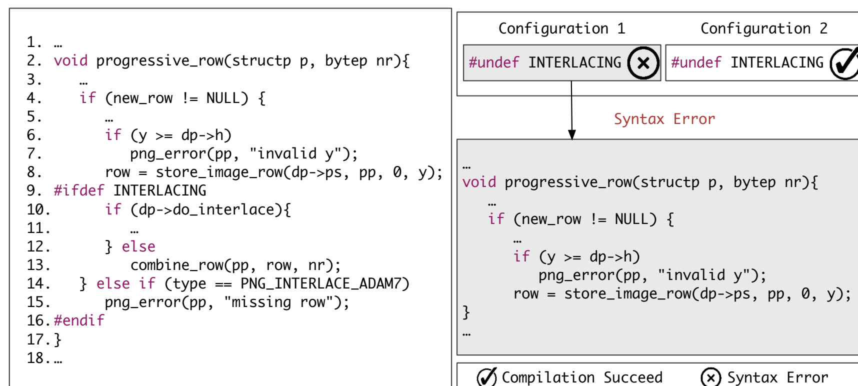


Figure 2.6: Code snippet of *Libpng* with a syntax error.

2.3 Variability-Aware Analysis

A variability-aware parser generates abstract syntax trees enhanced with variability information, guaranteeing the absence of syntax errors in all configurations. Variability-aware parsers, such as *TypeChef* [17] and *SuperC* [38], handle interactions of macros, file inclusion, and conditional compilation soundly. Instead of considering macro definitions, macro

[2] http://www.libpng.org

expansion, and file inclusion intertwined, variability-aware tools perform partial preprocessing, which preprocesses file inclusion and macro expansion, but retains variability information for further analysis [70]. In Figure 2.7, we present an example code snippet (left-hand side) and the results of performing partial preprocessing (right-hand side). As we can see, file inclusion and macro expansion have been performed at the right-hand side of Figure 2.7. Notice, though, that the preprocessor has not resolved conditional compilation and the resulting code, at the right-hand side, still contains the `#ifdef`, `#else`, and `#endif` directives.

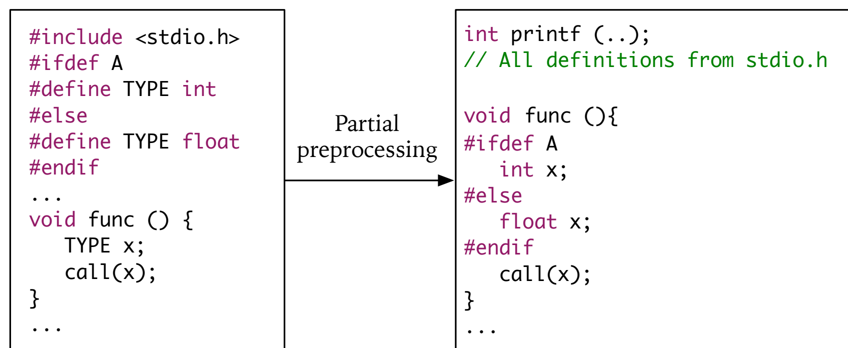


Figure 2.7: Performing partial preprocessing.

In Figure 2.8, we present an abstract syntax tree generated from an `if` statement with an undisciplined preprocessor directive. Notice that there is a choice node `A` that controls both configurations: (1) macro `A` enabled, and (2) macro `A` disabled. Therefore, by using the abstract syntax tree enhanced with variability information, we can search for configuration-related bugs in all configurations.

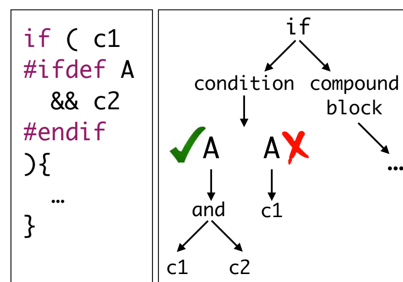


Figure 2.8: Abstract syntax tree enhanced with variability information.

2.4 Sampling Analysis

Although researchers have proposed approaches to analyze complete configuration spaces in a sound fashion for some classes of defects [71; 17; 72; 38; 18], as discussed, the vast majority of mature quality-assurance techniques consider only a single configuration at a time. For example, static-analysis tools operate typically on C code after the C preprocessor has resolved variability implemented through conditional compilation (e.g., implemented with `#ifdef` directives). To reuse state-of-the-art tools, such as *Gcc*, to detect configuration-related bugs, *sampling* is a viable alternative [34; 35; 36; 37; 20]. That is, instead of analyzing all configurations, one selects a subset of configurations to analyze individually. The effectiveness of sampling for detecting configuration-related bugs depends significantly on how samples are selected. Several sampling algorithms have been proposed in the literature. Next, we explain six state-of-the-art sampling algorithm using the example code snippet of Figure 2.9: *t-wise* [34; 35; 36; 37]; *statement-coverage* [73]; *random*; *one-disabled* [16]; *one-enabled*; and *most-enabled-disabled*.

The *t-wise* algorithm covers all combinations of t preprocessor macros: *pair-wise* checks all pairs of preprocessor macros ($t = 2$) [34; 35; 36; 37], and it selects four configurations regarding the example of Figure 2.9. Considering macros *A* and *B*, we can see that there is a configuration where both macros are disabled (*config-1*), two other configurations with only one of them enabled (*config-2* and *config-3*), and another configuration where both preprocessor macros are enabled (*config-4*). The same situation occurs for preprocessor macros *A* and *C*, and macros *B* and *C*. However, t can take integer values to check different combinations of macros, such as *three-wise* ($t = 3$), *four-wise* ($t = 4$), and *five-wise* ($t = 5$). As we increase t, the sizes of the sample sets also increase. Figure 2.10 presents the sample-set distributions of *three-wise*, *four-wise*, *five-wise*, and *six-wise* considering a file with a number of preprocessor macros ranging from zero to eighty. As we can see, *three-wise* and *four-wise* create small sample sets; *five-wise* and *six-wise* create larger sample sets.

The *statement-coverage* algorithm selects a set of configurations in which each block of optional code is enabled at least once [20]. As presented in Figure 2.9, by enabling preprocessor macros *A*, *B*, and *C*, the algorithm ensures that the optional code blocks `code`

```
#ifdef A
  // code 1
#endif

#ifdef B
  // code 2
#else
  // code 3
#endif

#ifdef C
  // code 4
#endif
```

pair-wise			
config-1:	!A	!B	C
config-2:	!A	B	!C
config-3:	A	!B	!C
config-4:	A	B	C

one-enabled			
config-1:	A	!B	!C
config-2:	!A	B	!C
config-3:	!A	!B	C

one-disabled			
config-1:	!A	B	C
config-2:	A	!B	C
config-3:	A	B	!C

most-enabled-disabled			
config-1:	A	B	C
config-2:	!A	!B	!C

statement-coverage			
config-1:	A	B	C
config-2:	A	!B	C

Figure 2.9: Comparing the sampling algorithms by example.

`1`, `code 2`, and `code 4` are enabled at least once. However, it needs another configuration (e.g., *A* and *C* enabled, and *B* disabled) to enable `code 3`. Note that including each block of optional code at least once does not guarantee that all possible combinations of individual blocks of optional code are considered.

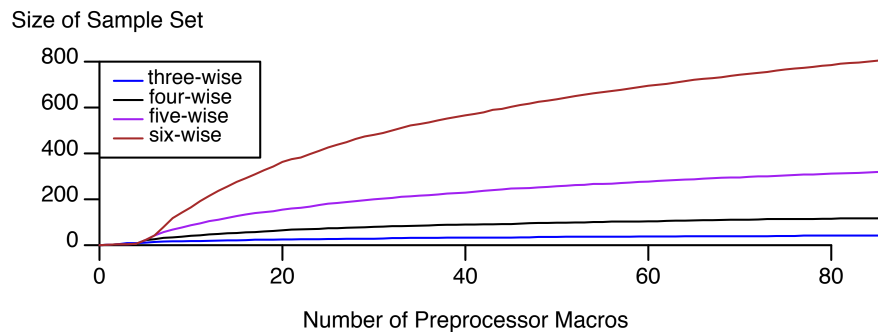


Figure 2.10: Sample sets of *t-wise* sampling.

The *most-enabled-disabled* algorithm checks two configurations independently of the number of preprocessor macros. When there is no constraints among preprocessor macros, it enables all macros (*config-1*), and then it disables all preprocessor macros (*config-2*). **One-disabled** is an algorithm suggested by Abal et al. [16] based on 42 bugs found in the *Linux Kernel*. It disables one preprocessor macro at a time. We can also see in Figure 2.9 that it disables preprocessor macro *A* in *config-1*, macro *B* in *config-2*, and macro *C* in *config-3*. In contrast, **one-enabled** enables one preprocessor macro at a time.

Finally, the *random* sampling algorithm, which receives as input the maximum number of configurations (c) to check per file. Then, it creates c distinct configurations with all preprocessor macros within the file and randomly assigns `true` or `false` for every macro of each configuration. For files which a *brute-force* algorithm requires fewer configurations than the maximum number of configurations (c) per file, random selects all configurations. For instance, *brute-force* selects 2^n configurations, where n is the number of distinct configuration options. Thus, for a given source file with 5 distinct configuration options, *brute-force* selects 32 configurations. Assuming that we are checking 40 configurations per-file using *random* (i.e., $c = 40$), it makes sense to check all 32 configurations selected by the *brute-force* algorithm.

2.5 Static Analysis Tools for C

In this section, we present concepts of static analysis and discuss some tools to perform this kind of analysis in C. Static analysis is a technique to analyze computer systems without actually executing them [74]. Developers can use static analysis tools as a writer use spell checkers, i.e., to avoid subtle mistakes. However, although poor developers, which do not program well, gain benefits from using a static analysis tool, it does not transform them into expert developers [75].

Static analysis tools can analyze the source code directly using, for example, abstract syntax trees, i.e., seeing the source code as the compiler sees it. However, it can bring ambiguity problems since the tool and the compiler may interpret the source code differently. In contrast, some static analysis tools may require compilation to analyze object code, i.e., seeing the source code as the runtime environment sees it. In the latter case, the compiler has already made its job, and the static analysis tool does not have to guess how the compiler interprets the code [75].

In the context of the C language, static analysis tools that use object code miss information about variability, i.e., the compiler has already preprocessed the source code, and the analysis considers only one configuration. Thus, to analyze program families completely, C static analysis tools should analyze the original source code, which contains all preprocessor directives and variability information.

There are several static analysis tools in the literature to analyze different aspects of the source code, such as type checking, style checking, program understanding, program verification, property checking, bug finding, and security reviews [75; 76; 77; 78; 79]. Table 2.1 shows some tools to perform static analysis in C. In this study, we focus on static analysis tools to detect bugs.

Table 2.1: Tools to perform static analysis in C.

Tool name	**Category**	**Input**
API Sanity Checker	bug finding	object code
Clang Analyzer	bug finding	object code
Cppcheck	bug finding	source code
FlawFinder	security review	source code
Nsiqcppstyle	style checking	source code
Splint	bug finding	source code
Valgrind	bug finding	object code
Vera++	style checking	source code
Coverity	bug finding	source code
Coccinelle	bug finding	source code
Lint	bug finding	source code
PVS-Studio	bug finding	source code

The most common complaint regarding static analysis tools is false positives, i.e., the tools report a problem in a program when no problem actually exists (*false alarm*). Developers may think that the tools do not work properly, and may, for example, spend time by looking for nonexistent bugs. However, it is worse if the problem is related to false negatives, in which the problem exists but the tools do not detect it [75]. False positives and negatives may happen depending on the techniques that the tools use, such as intra-procedural and inter-procedural analyses [74]. The former considers the analysis of functions in isolation and the latter analyzes interaction between calling and called functions, analyzing the whole program.

Intra-procedural and inter-procedural analyses get more complicated in C due to the presence of preprocessor directives. For instance, consider the example presented in Figure 2.11, in which developers allocate memory to variable `ptr` in file *Main.c* at Line 4. A static analysis tool that performs only intra-procedural analysis may detect a false positive, i.e., a memory leak in variable `ptr`. A tool that performs inter-procedural analysis may pro-

duce a better result, as it considers the `#include` directive in *Main.c* at Line 1 to recognize function `test` defined in *Memory.c* (which calls `free`). However, the tool should be variability-aware to detect that the memory leak happens only when macro `A` is disabled. For these reasons, false negatives and positives may occur in static analysis tools.

Main.c

```
1. #include "Memory.h"
2.
3. int main (int argc, char **argv){
4.     int *ptr = (int *) malloc (sizeof (int));
5.     test (ptr);
6. }
```

Memory.h

```
1. void test (int *x);
```

Memory.c

```
1. void test (int *x){
2.   #ifdef A
3.     free(x);
4.   #endif
5. }
```

Figure 2.11: Code snippet to discuss strategies of static analysis tools.

2.5.1 Cppcheck

Cppcheck is a static analysis tool that have been used in many open source projects and its developers claim to minimize false positives. The tool analyzes C source code and detects different types of bugs, such as memory and resource leaks, dereferences of null pointers, and uninitialized variables, using intra and inter-procedural analyses. *Cppcheck* provides support to analyze different configurations of the source code. It focuses on the identification of bugs that compilers normally do not detect, such as memory and resource leaks, uninitialized variables, and dereferences of null pointers. *Cppcheck* implements a sampling algorithm to detect bugs in different configurations. For instance, *Cppcheck* analyzes the following configurations when checking the code snippet of *Vim* presented in Figure 2.12. The tool checks the code with all macros disabled, then, it activates each macro separately, i.e., `CMDL`, `UNIX`, `BUFLIST`, `TITLE`, and `PERL`. Finally, it checks the source code with nested macros together, i.e., `UNIX` and `BUFLIST`.

```
// Code here..
#if defined(CMDL)
   static char_u *buflist_match __ARGS((regprog_T *prog, buf_T *buf));
#endif

// Code here..

#ifdef UNIX
   #ifdef BUFLIST
      static char_u *fname_match __ARGS((regprog_T *prog, char_u *name));
   #endif
   static int buf_same_ino __ARGS((buf_T *buf, struct stat *stp));
#else
   static int otherfile_buf __ARGS((buf_T *buf, char_u *ffname));
#endif

#ifdef TITLE
   static int ti_change __ARGS((char_u *str, char_u **last));
#elif defined (PERL)
   static void clear_wininfo __ARGS((buf_T *buf));
#endif
```

Figure 2.12: Code snippet of *Vim* to show how *Cppcheck* selects configurations.

2.6 Refactoring

Refactoring is the process of changing a software system with the purpose of improving its internal structure without modifying its external behaviour [80; 33]. With refactoring we can take a bad design and rework it to a well-designed code. To refactor a code, we perform a set of simple and small code transformations, e.g., move a field from one class to another and pull some code out of a function to make its own function, and the cumulative effect of these small changes can radically improve the design of the software system.

The design of a system decay due to changes performed to realize short-term goals during software evolution. Hence, developers perform refactorings to improve the design of software. Refactoring can also make the software easier to understand, help to find bugs, and aid developers to program faster [33]. The reason is that when refactoring a code, developers have a code that works but is not ideally structured. This way, a little time spent refactoring can make the source code simpler, and better organized and structured, which will help developers to better understand the source code, find bugs and make software development tasks faster [80; 33].

Before you start refactoring, it is important to have a solid suite of tests [33]. Thus, after identifying a refactory, i.e., a place where refactoring should be performed [80], we refactor the code, and check whether the test cases still pass in the refactored version. To identify

places to refactor, we can search for bad smells in the code, such as duplicated code, long methods, large classes, long parameter list, and undisciplined directives [33]. In summary, the steps to refactor a software system are:

1. Identify bad smells in the source code;
2. Check whether you have a solid suite of tests for that specify part of the code. Otherwise, create a solid test suite;
3. Apply the refactoring to improve the quality of the code;
4. Run the test suite again using the refactored code to check whether the tests still pass.

In the context of the C language, refactoring becomes a challenge because of the preprocessor directives [81; 82; 83]. To refactor program families we have to consider all valid configurations of the source code. Otherwise, we may introduce behavioral changes. For instance, Figure 2.13 presents a sample code of a C program family. In this context, macro `A` is disabled and a developer wants to rename variable `x` declared at Line 2 on the left hand side. However, variable `x` is used when macro `A` is disabled as well. Thus, a refactoring tool that is not variability-aware would rename variable `x` only in the active source code as we can see on the right hand side of Figure 2.13, leaving variable `x` unchanged at Line 4. Existing refactoring tools, such *Eclipse* and *Xcode*, perform wrong code transformations similar to this one because they consider only one configuration at a time.

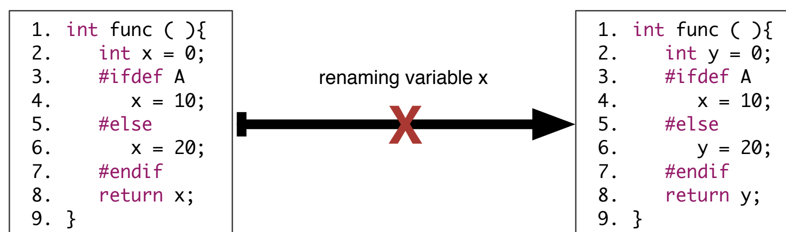


Figure 2.13: Wrong code transformation that introduces a compilation error.

In Section 2.2.2, we presented a bad smell (i.e., an undisciplined directive in *Vim*). In Figure 2.14, we present the refactoring used in previous studies to remove undisciplined directives by cloning the source code [4; 30; 32]. Notice that there are lines of duplicated

code at Lines 3 and 8, for example, in the refactored code, as we can see at the right-hand side of Figure 2.14. In Chapter 6, we proposed alternative refactorings to remove undisciplined directives without cloning the source code.

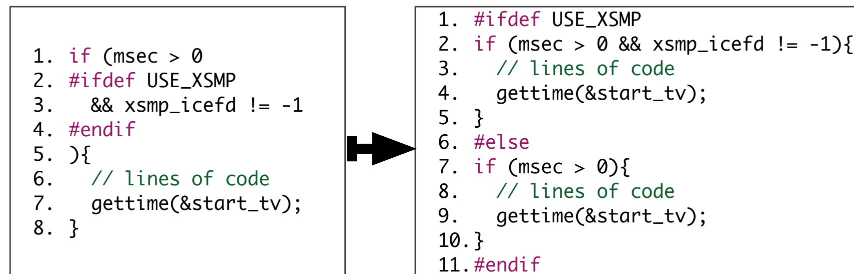


Figure 2.14: Refactoring to remove undisciplined directives by cloning code.

Chapter 3

Problem Dimension

In this chapter, we present a study performed to understand how developers perceive the C preprocessor. All prior studies [3; 4; 13] on the C preprocessor that we are aware of were based on conceptual arguments or evidence extracted from software repositories. Our study is designed to elicit the *perception* of developers by talking to them.

In Section 3.1, we discuss the challenges induced by the C preprocessor according to our literature review. This review of the state-of-the-art guided us in the design of our study to analyze whether and how the perception of developers differs from that in the research literature. Next, in Section 3.2, we present the settings of our study and the research methods we used to understand the perception of developers regarding the C preprocessor. Then, we discuss the results of our study in Sections 3.2.2–3.2.2, presenting the problems of using the C preprocessor in practice according to the perception of developers.

3.1 Challenges Induced by the C Preprocessor

The preprocessor is widely used in practice in almost all projects written in C. It is executed during the compilation process and performs three interacting tasks: lexical inclusion, lexical macros, and conditional compilation. In Figure 3.1, we present the original code on the left-hand side and the result of performing each task on the right-hand side.

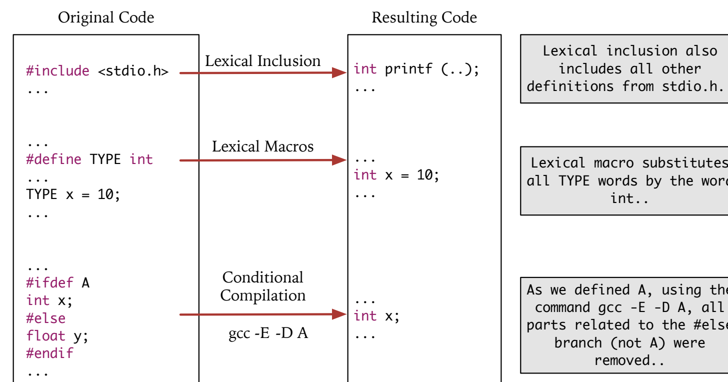


Figure 3.1: The interacting tasks of the C preprocessor.

According to our literature review, all three functions have been criticized:

- Lexical inclusion causes large amounts of I/O operations during compilation and slows down the build process. For example, an average file in the *Linux* kernel includes over 300 header files [17]. There is movement in the C community towards a proper build system to replace `#include` directives [84].
- Lexical macros allow all kinds of potential problems [3] since they have no notion of structure, hygiene, or capture avoidance that advanced macro systems support [85; 86; 87; 26; 23]. Developers avoid these problems by following certain patterns when defining macros [3], which are broadly adopted and also checked by a number of static analysis tools [31]. In addition, C++ introduced several language features to replace common uses of preprocessor macros [63; 88].
- Since conditional compilation removes code before compilation, it causes compilers and many other analysis tools to see only parts of the code. It has been criticized as limiting separation of concerns, as obfuscating the code, as being error prone, and as preventing tool support. Interactions of conditional compilation with lexical inclusion and macro expansion make it even harder to reason about the preprocessor execution.

Next, we discuss the challenges of using the C preprocessor in practice according to the research literature. In particular, we focus on conditional compilation, as file inclusion and macro expansions are relatively well understood and there are mitigation strategies available in the literature [62; 63; 3; 64; 65].

3.1.1 Readability and Separation of Concerns

Several studies criticized the C preprocessor regarding its limited separation of concerns and code obfuscation, which make maintenance and code comprehension difficult [13; 3; 30; 4; 17]. In particular, when conditional directives are used at fine granularity and are strongly scattered, it can be difficult to follow the control flow logic [6; 14]. Such source code is sometimes referred to as the "`#ifdef` hell" by developers [15]. Long and deeply nested conditional directives also can make it difficult to see when specific code fragments are included [6; 11; 9]. Many researchers have proposed aspect-oriented programming as an alternative [15; 28], where optional code would be separated into distinct code artifacts and woven together at compile time, but we are not aware of any adoption beyond some research projects.

A specific practice that has been discussed in detail is the use of *undisciplined directives:* conditional compilation directives that do not align with the code structure, as discussed in Chapter 2. In Figure 3.2, we illustrate some examples of undisciplined directives. Undisciplined directives are related to error proneness [3; 4; 12; 17], hindered code understanding and maintainability [13; 3], and limitations in tool support [81; 30; 13; 89]. An empirical study by Liebig et al. [4] revealed that most conditional compilation directives in 40 open source C projects are disciplined, but 15.6 % of all `#ifdef` blocks do not align with the code structure.

```
if (b_ffname != NULL
#ifdef FEAT_NETBEANS
  && netbeansReadFile
#endif
){
  // lines of code
}
```

```
mfp = open(mf_fname
#ifdef UNIX
 , (mode_t)0600
#endif
#if defined (MSDOS)
  , S_IREAD | S_IWRITE
#endif
);
```

```
#if defined (GUI_W32)
void msgNetbeansW32(
#else
void msgNetbeans(Xt client,
#endif
XtInputId *id){
  // lines of code..
}
```

Figure 3.2: Real code snippets taken from *Vim* with undisciplined directives.

3.1.2 Combinatorial Explosion and Parsing Unpreprocessed C Code

Conditional compilation decides which code fragments to include depending on the values of macros. The number of possible configurations explodes exponentially with the number

of preprocessor macros involved in `#ifdef` and similar directives. C projects often have a large number of conditional directives depending on many macros; for instance, which parts of the *Linux* kernel are compiled depends on more than 12 thousand macros [20; 67].

A separate analysis of every possible configuration simply does not scale in any but the smallest systems. A typical strategy to cope with the combinatorial explosion is through sampling, for example, by analyzing configurations with the majority of conditional code included. For more systematic sampling, researchers have proposed several combinatorial testing strategies [90; 36] and other strategies that maximize configuration coverage [20]. Sampling is inherently incomplete though and may not discover issues occurring only in few configurations due to interactions or complex `#if` conditions.

Some researchers have started to investigate tools that can parse unpreprocessed code and preserve all compile-time choices during the analysis. While earlier tools used unsound heuristics or supported only specific usage patterns of the C preprocessor (e.g., requiring disciplined directives) [13; 30; 89], more recent tools as *TypeChef* [17; 47] and *SuperC* [38] can accurately parse and analyze unpreprocessed C code, covering all configurations. In the product-line community, such analyses are called *family-based analyses* [71].

3.1.3 Error Proneness and Guidelines

Previous studies discussed the error-prone characteristics of the preprocessor [3; 2; 7] and found many bugs related to conditional compilation [16; 19; 20; 91; 12; 72; 3; 40], ranging from dead code to syntax and type errors and to behavioral issues and memory leaks. Spencer and Collyer [2] argue that many macro combinations are tested and often do not even make sense. Others argue that the simplicity of the C preprocessor enables developers to make ad-hoc extensions instead of restructuring the code, which leads to poor code quality and bugs related to preprocessor usage [7; 13].

Code guidelines have been developed to prevent certain problems, e.g., undisciplined directives or scope issues with macros [31; 4; 3]. Even though some of them can be enforced automatically by analysis tools [31; 4], research shows that such code guidelines are often but not strictly followed [3; 4].

3.1.4 Difficulty to Develop Tool Support and Syntactic Preprocessors

Finally, preprocessor directives also make the development of tool support more difficult [4; 17; 38]. Even simple tasks as removing obsolete macros or identifying dead code require sophisticated analyses [13; 91]. Developing refactoring engines for C code is extremely challenging due to the need to parse unpreprocessed code (possibly with many undisciplined directives) and the need to deal with macro expansion [81; 22; 21; 92]; it is challenging even when conditional compilation is not considered [64; 93].

Many academic proposals for preprocessor alternatives are driven by a desire to provide better tool support and analysis. For example, *ASTEC* is a syntactic preprocessor that enables precise refactoring [23]. Several other syntactic preprocessors or related environments have been proposed [26; 27; 87; 94; 1; 95]. Some researchers propose means to refactor existing C code to alternative implementations [23; 94; 28] or at least undisciplined to disciplined directives [30; 32; 50], as discussed in Chapter 2. We are not aware of any adoption of these alternatives in practice though.

3.2 Research Study

The goal of our research study is to analyze the common pitfalls of the C preprocessor, as perceived by C developers. We specifically collect information that cannot be observed by analyzing only artifacts as in previous studies [4; 17; 12; 16; 30; 47]. We performed this research study primarily by interviewing developers and cross-validating our results with a survey, other information from software repositories, and related studies. In this section, we give an overview of our research method. All the details about the research methods are available in the companion[1] appendix [96].

For this research, we combine several empirical research methods, including interviews, surveys, and mining software repositories. Empirical research methods allow researchers to investigate how human developers think and behave. We study not only the outcome of the development process, but assess also their opinions and perceptions. If not conducted carefully, empirical research can result in biased and superficial results. However, whole communities of researchers have investigated how to perform empirical studies that reduce

[1] http://drops.dagstuhl.de/opus/volltexte/2015/5516/

biases and enable reliable and reproducible despite potentially vague research materials. For example, by following strict protocols and documenting steps and research results when analyzing transcribed interviews can mitigate many biases that researchers might otherwise introduce. In addition, cross-validating results from different sources is essential. This way, results complement and confirm each other and form a more reliable bigger picture. In this study, we strictly followed established research methods and cross-validated our results across several sources and with prior research results, as we will explain.

3.2.1 Overall Study Design

The motivation for our study is based on the criticism that the C preprocessor has received from academics [13; 3; 30; 4; 17; 2], the number of alternatives proposed [26; 23; 27; 15; 28] that have not been adopted in practice, and the broad use of the preprocessor in several real-world projects.

Specifically, we raise the following research questions:

RQ1. Why is the C preprocessor still widely used in practice?

RQ2. What do developers consider as alternatives to the preprocessor?

RQ3. What are the common problems of using the preprocessor?

RQ4. Do developers care about the discipline of directives?

Research Strategy

We performed our research in three phases. In the first phase, we analyzed the literature and identified the research questions stated above (see also Section 3.1). In the second phase, we performed semi-structured interviews with 40 developers. In the third phase, we cross-validated our interview findings by conducting a survey among developers contributing to open source C projects, mining data from 24 software repositories, and comparing our results with prior research results.

Corpus

For this study, we use a corpus of 24 open source C systems. With the revision history of the systems in the corpus, we identified candidate interviewees and survey participants, and we studied technical aspects. We selected the systems in the corpus based on prior corpus studies

on the C preprocessor [3; 67], covering a range of different domains and sizes (2.6 thousand to 7.8 million lines of code). We selected only projects for which we could find developer contact information in commits. The corpus includes the projects listed in Table 3.1.

Table 3.1: General information about projects repositories.

Project	Domain	Number of Commits
apache	Web Server	25,615
bash	Interpreter	68
bison	Parser Generator	5,423
cherokee	Web Server	5,748
dia	Diagramming Software	5,634
flex	Lexical Analyzer	1,609
fvwm	Window Manager	5,439
gawk	Interpreter	1,345
gnuchess	Game	236
gnuplot	Plotting Tool	8,024
gzip	File Compressor	445
irssi	IRC Client	4,130
libpng	Image Library	2,188
libsoup	Web Service Library	2,005
libssh	Security Library	2,915
libxml2	XML Library	4,246
lighttpd	Web Server	1,470
linux	Operating System	445,169
lua	Programming Language	83
m4	Macro Expander	953
mpsolve	Mathematical Software	1,434
rcs	Revision Control System	915
sqlite	Database System	553
vim	Text Editor	5,720

Interviews

We started our empirical study by interviewing developers regarding how they perceive the C preprocessor. To reduce any potential bias and to make our study replicable, we followed the established exploratory research method *grounded theory* [97; 98]. We performed *semi-structured* interviews [99; 100], which are informal conversations where the interviewer lets the interviewees express their perception regarding specific topics. To elicit not only the

foreseen information, but also unexpected data, we avoided a high degree of structure and formality and, instead, used open-ended questions. To cover the topic broadly, our questions evolved during the interview process based on gained insights [97; 98]. We followed standard guidelines regarding how to perform interviews [99; 100]. For example, we explained the purpose of the interviews, we provided clear transitions between major topics, we did not allow interviewees to get off topic, we allowed interviewees to ask questions before starting the interview, and we scheduled the interviews beforehand.

The interviews were grounded in research questions RQ1–4. We started an interview by asking developers about their experience with the C preprocessor and then tried to cover 4-6 different topics. The topics evolved during the interviews, and we asked different topics to different developers based on their background and answers. This is a standard approach to cover a topic broadly and qualitatively. For example, questions included *'In which situations do developers use conditional directives?'*, *'How do developers test different combinations of macros in their code?'*, and *'What do developers think about directives that split up parts of C constructions?'*. In addition to these questions, we used code snippets to ask developers concrete questions about code to encourage them to give more concrete answers. For each interviewee, we searched through the code repositories and selected code snippets related to that specific developer. We sent such snippets by email before the scheduled interview.

We performed 10 phone and 30 email interviews. We initially contacted developers via email presenting some information about our project and asked them to participate. In this step, we encouraged developers to perform phone interviews, however, we also provided the alternative to answer our questions via email. When necessary, the emails interviews involved back and forth conversations (i.e., a dialogue between researcher and participant). We sent at least one additional email with further questions in 19 (63%) out of the 30 email interviews we conducted. This and the fact that we cover the same questions in both phone and email interviews allows us to discuss them together as interviews, and not separately as phone and email interviews. To analyze the interview transcripts, we again followed established research methods: coding the answers, analyzing keywords, organizing them into concepts and categories, and writing memos [97]. The two main researchers involved in this project met weekly to discuss the memos and noticed that interviewees progressively started to give similar answers, a situation called *saturation* [97]. At this point, we considered

the topic sufficiently clear and focused on other topics that needed further elaboration.

We selected participants for the interviews from active developers in the 24 projects of our corpus. By mining the repositories, we identified the top 10 % active developers in each project that regularly use conditional compilation (ranked by code churn). We sent emails to 213 open source developers, and 32 (15 %) participated in our interviews. Even though many open source contributors expressed that they primarily worked in industrial projects, we additionally explored whether interviewees from industrial projects would provide new insights. After reaching out to our contacts (convenience sampling), eight developers from Brazilian companies accepted to participate in our interviews. Most of our 40 interviewees self-identified as having at least 5 years of experience and many worked both within open source and industrial contexts. Our selection of developers is biased toward developers with experience with conditional compilation, which we counteracted however by cross-validating our results with a survey of a broader population. In our result presentation, we refer to individual anonymized participants as P1–P40.

Online Survey

Whereas our interviews are designed to elicit qualitative insights into practices and reasons, our survey is designed to collect quantitative data from a large population. We designed the survey after completing and evaluating the interviews. It is a standard approach to first perform qualitative investigations to identify relevant questions and subsequently perform a survey to explore them quantitatively in a larger population [101; 102].

With the survey, we explored topics that were unclear from the interviews or where we wanted additional quantitative data. We performed an online survey to reach more developers and again followed common guidelines for that research method [103]. For several questions, the survey included code snippets to make questions more concrete. We mention the survey questions while discussing our results in Sections 4–7.

To select participants for our online survey, we aimed at reaching a broader audience of developers with different levels of experience regarding conditional directives usage. We randomly sampled from all developers that contributed to the 24 projects in our corpus, excluding our interviewees. We sent emails to 3091 developers and 202 (6.5%) filled out our survey.

Mining Undisciplined Directives

To investigate the issue of undisciplined directives further, one of the most controversial and criticized issues in the literature, we mined software repositories to analyze different versions of the source code and statically detected undisciplined directives. Specifically, we analyzed each commit in 14 projects of our corpus. We considered only projects with at least two active developers to compare their programming style regarding undisciplined directives. An active developer has high code churn along the commit history. We used a modified version of Liebig's *Cppstats* tool [4]. With this tool, we identified all commits that introduced undisciplined directives, data analyzed grouped by developer. Then, we interviewed four developers regarding their reasons for introducing specific undisciplined directives.

3.2.2 Results and Discussion

We discuss the research questions next.

RQ1: Why is the C preprocessor still widely used in practice?

The C preprocessor has been heavily criticized in previous research, which raises the question of why it is still used in practice (RQ1). To fully answer this question, we need two pieces of information. The first is whether developers are actually aware of these (academic) criticisms, and the second is the set of scenarios in which developers find the C preprocessor useful. If developers are aware of the potential problems, but still use the C preprocessor, this suggests that there are cases in which using the C preprocessor is still the preferred or even the only available alternative. However, to identify such cases, we first need to understand the various situations in which the C preprocessor is used.

Developers Awareness of C Preprocessor Criticism

We found that developers are aware of the criticism the C preprocessor has received, but they still believe that it is an elegant solution to handle variability and overcome portability problems, if properly used *(P1-P3, P18, P22, P23, P26)*. As one developer (*P39*) explains: *"Every feature of any technology can be abused or misused. When used appropriately, the use of preprocessor directives is not a problem."* That said, many developers *(P1-P3,*

P5-P8, P19, P20, P22-P26, P30-P33) are aware that they must follow code guidelines to minimize problems related to code comprehension, maintainability, and error proneness (C preprocessor problems are discussed in more detail in Section 3.2.2).

Usage of the C Preprocessor

Our discussion with developers reveals the following fives cases in which they use the C preprocessor.

- *Portability.* Despite being from different domains, many of the systems we studied need to support many platforms and operating systems. The preprocessor is perceived as a convenient way to ensure the system's portability across different environments. For example, developers often use conditional directives to check settings of operating systems, platforms, compilers, and library versions *(P1-P3, P6, P17-P25, P27)*. Based on these settings, developers use certain macros, types, and header files that may only be available when using a specific operating system or compiler. For example, it is not possible to include *Windows* specific headers such as `windows.h` when compiling the source code on *Linux* or *Mac OS*. In addition to handling platform-specific header files, portability involves checking for specific system constraints as well as making use of platform-specific functionality during implementation *(P1, P3, P18, P19, P21, P24, P27)*. For example, in some operating systems, such as *GNU Hurd*, there is no imposed limit on overall file name length, as there is on *Windows*.

- *Variability.* Developers often use conditional directives to provide *optional features* or to *select between alternative implementations*. For example, participant (*P4*) describes his use of variability as follows: *"I use conditional directives to remove parts of the library I do not need, since it makes the binary code much smaller."* Reducing binary size may influence the decision in using macros to represent optional functionality (i.e., *features*). The `DEBUG` feature is one extreme example, which was mentioned frequently by developers. It is a common feature developers use to print messages along the source code to understand what is going on during execution *(P21, P22, P23)*. Since `DEBUG` may not be useful for end-users, developers guard debugging code with the corresponding macro such that end-users can exclude it from the binary code during compilation. Several developers also state that they commonly use conditional

directives to support alternate implementations (*P13, P27, P38-P40*). For example, in *Libssh*, developers can choose between different cryptographic libraries such as *Libcrypt* or *Libcrypto*, depending on the characteristics of the cryptographic algorithms they want to use. They find that the C preprocessor provides a convenient way to switch between such libraries at compile-time

- *Code Optimization.* Some developers explain that, apart from excluding unnecessary functionality, they also explicitly use conditional directives to optimize the code for performance or size *(P3, P4, P40)*. Interviewees explain that they often do not trust that all compilers will properly optimize their code. Thus, in some cases, developers take the task of optimizing the code into their own hands by implementing known code optimizations after checking for compiler name and version at compile-time using the C preprocessor. For example, the *Gcc* compiler offers some *GNU Extensions* such as type discovery and zero-length arrays. Developers explain that they want to make use of such optimizations if they are aware of their availability as this allows them to actively make the binary code smaller and faster.

- *Code Evolution.* A few developers state that they also often use conditional directives during the introduction of new code versions related to critical functionality *(P27, P28, P39)*. In this context, they introduce new implementations inside conditional directives, but they remove the previous version only when the new version is stable. They explain that by using conditional directives, they can switch between the old and new implementations for testing purposes.

- *Language Limitations.* Many developers mention using conditional directives because of the limitations of the C language *(P6, P14-P16, P20, P36-P38)*. For example, they use `#ifdef` checks to avoid multiple inclusion of header files. Such header guards (or include guards) are probably one of the few applications of the C preprocessor that is accepted by critics [2].

Some developers also mentioned using macros and function-like macros to avoid code duplication and to encapsulate frequently-changing code *(P20, P39, P40)*. In this context, developers need to only change the definition of macros instead of changing all occurrences

```c
#ifdef DEBUG
#define DEBUG_MSG printf
#else
#define DEBUG_MSG (format, args...) ((void)0)
#endif
// Developers do not need to check #ifdef DEBUG multiple times..
DEBUG_MSG ("message..");
```

Figure 3.3: Using function-like macros to avoid code duplication.

in the code. For example, Figure 3.3 shows how function-like macros can be used to define the behavior of `DEBUG_MSG`. While avoiding duplication and supporting encapsulation are not specific to the C language, using the C preprocessor is perceived as a convenient way to change function definitions at compile-time instead of at run-time. Previous studies [63; 62] considered the replacement of preprocessor macros with new features and idioms in the C++ programming language.

We observed that the answers in our interview data reached a saturation point that is why we did not include this research question in our survey. This is also supported by the fact that many of the cases of C preprocessor usage we find (apart from the rare case of supporting code evolution) align with those found in previous work. For example, Ernst et al. [3] observed that portability accounts for 37 % of the use of conditional directives in the systems they examined, while include guards account for 6.2 %. They also found frequent usage of inline functions or function-like macros. Ernst el al. also argue that in order to eliminate some of the preprocessor usage, developers must be confident that the compiler will perform the necessary code optimizations. Our interviews support this and further suggest that, even after more than a decade, developers still lack this confidence in compiler optimizations.

SUMMARY

Developers are aware of the criticism the C preprocessor receives, but still use it in the following situations: (1) supporting portability, (2) supporting variability, (3) providing code optimizations, (4) supporting code evolution, and (5) overcoming limitations of the C language.

Data Sources: Interviews and Prior studies [3; 4; 104; 2]

RQ2: What do developers consider as alternatives to the preprocessor?

We have seen that developers are aware of problems and risks of using the preprocessor but still have several use cases for which they need the C preprocessor's functionality. While researchers have proposed alternatives [26; 23; 27; 87], we wanted to see which alternatives developers are aware of or would recommend. We asked whether developers have thought about alternatives to the C preprocessor. We did not ask about specific alternatives or tools, because it was apparent that they usually would not be familiar with them. When we asked for preferences, we were only comparing different ways of using the C preprocessor. Our main goal with this question was to identify perceived alternatives.

We received three kinds of answers: suggestions to use the C preprocessor in specific ways (guidelines on how to structure the code), suggestions to use in language runtime mechanisms instead of compile-time mechanisms of the preprocessor, and arguments that the preprocessor cannot be replaced. Equally important is that none of our interviewees mentioned alternative preprocessors, aspect-oriented programming, or meta-programming solutions suggested by researchers. In the following, we discuss the three kinds of answers we received, cross-validated with survey findings.

Guidelines for Structuring Code

The first common suggestion to avoid using the C preprocessor is to separate alternative and optional code on the function, file and directory structure level *(P3, P8, P9, P14, P18, P24)*. For functions, the idea is to define alternative implementations of a function in separate files and to use the build system and the linker to choose the desired one. Figure 3.4 (b) shows an example of this. Similarly, grouping related files in the same directory can also move compilation control to the build system, i.e., the whole directory will be compiled or not. Such structuring of the code means that no preprocessing is necessary within files. Additionally, the code structure is portable and requires no special tools. Nonetheless, one developer *(P15)* cautions that structuring the code in this way may leave it more difficult to comprehend. It is also difficult to deal with similar functions and code duplication if developers do not use helper functions for the common code.

The answers we received align with previous academic discussions [105; 2], but did not reach a saturation point in our analysis. Therefore, we asked a larger population of

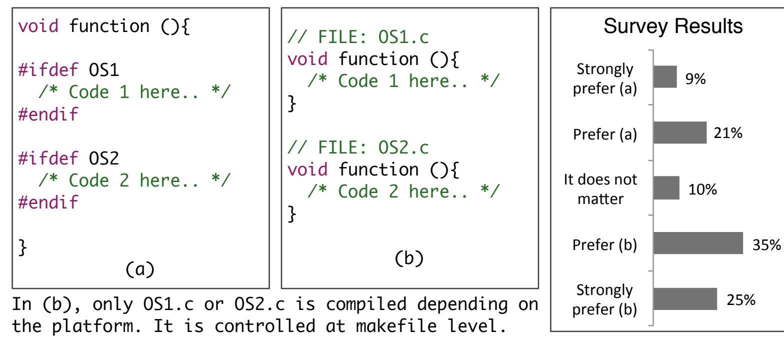


Figure 3.4: Preprocessor directives or portability functions.

developers whether they prefer this code structuring strategy. In the survey, we present the two equivalent code snippets shown in Figure 3.4, asking developers which one they prefer based on a five-point Likert scale. We find that 30 % prefer to use conditional directives inside function bodies, i.e., Figure 3.4 (a), while 60% prefer to use different functions to solve portability concerns, i.e., Figure 3.4 (b). The remaining 10% of respondents had no preference between both options.

Alternative In-language Runtime Mechanisms

Another alternative that was suggested frequently during our interviews is the use of run-time variability binding (i.e., C `if` Statements) instead of compile-time binding with the preprocessor (i.e., `#ifdef` directives). An example of this is shown in Figure 3.5 (b).[2] Many developers state that they prefer to solve variability at run-time, when possible, since it is more flexible *(P1-P4, P6, P23, P40)*. One developer (*P23*) illustrates on this, saying: *"If something can reasonably be done without the preprocessor, I choose [to do it] that way. [Once the binary is there,] it is much more flexible to enable functions at run-time or with a configuration file than having to recompile the project again."*

To achieve run-time variability, interviewees suggest to use variables and enumerators instead of macros with constant values. They also suggest using inline functions to optimize the source code instead of function-like macros. However, developers caution that runtime variability, e.g., the use of global variables and enumerators, is not thread-safe in C, and that using runtime variability is not possible in some cases. For instance, when runtime checks

[2]While the decision here is still made statically, it could also be loaded from command-line options.

are not feasible due to performance reasons. One developer clarifies that checks at runtime would cause performance overheads when checking for debugging mode, for example. This developer explains that when your goal is to process millions of I/O operations in the Linux kernel, for example, having runtime (i.e., C `if`) debug checks to verify certain assumptions would prevent you from scaling. However, developers still need a mechanism to easily verify assumptions when checking for code correctness, and they suggest that this can be cheaply achieved using the C preprocessor at compile-time.

Since developers' comments about run-time checks were not entirely consistent, we use the survey to see the preference of a broader population. This time, we present the two code snippets shown in Figure 3.5. In Figure 3.5 (a), we use conditional directives, while in Figure 3.5 (b), we use run-time variability with C `ifs`. We again ask survey participants to indicate which style they prefer using. Surprisingly, 75 % mentioned that they prefer to use conditional compilation directives, i.e., Figure 3.5 (a), while only 19 % prefer to use run-time variability, see Figure 3.5 (b). The remaining 6 % of developers did not have a preference.

Based on the results of our interviews, we expected a higher percentage of developers to prefer using run-time variability in the survey. Accordingly, we went back to our interview data to see if we can find supporting reasons for why this might be the case. We find that developers might be inclined to use `#ifdefs` instead of `if` checks because of the following reasons. First, as stated by developer *P1*, when using conditional directives, it is easier to see the optional code. In other words, it is clear that the block of code from lines 3 to 8 is optional in Figure 3.5 (a). Second, developers *P3* and *P4* mentioned that by using variables instead of macros, developers do not know whether the compiler will optimize the source code. For instance, if the developer does not define `PM3D` in Figure 3.5 (b), variable `PM3D_RT` is always zero, i.e., `false`, and the block of code from lines 7 to 10 becomes unreachable. Developers argue that compilers may perform optimizations to remove such dead code, but there are no guarantees (i.e., this is compiler specific and depends on the compiler settings). Additionally, a few developers mention that some compilers may issue warnings about such unreachable code or about cases where the `if` condition would always be true which they find annoying *(P3, P29)*.

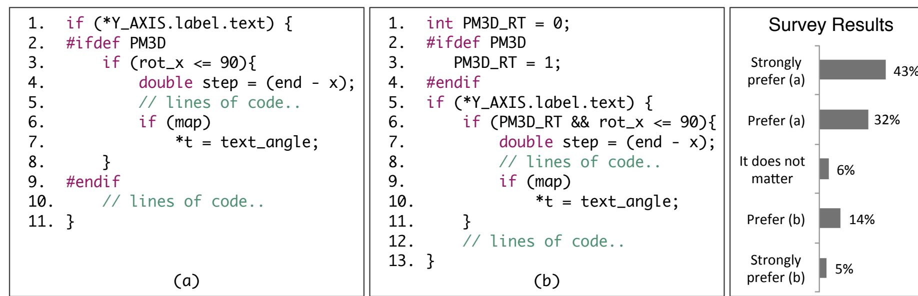


Figure 3.5: Conditional compilation or run-time checks.

No Perceived Alternatives

Several interviewees mentioned that they have not thought about alternatives to the C preprocessor *(P17, P19-P21, P25-P28)* and that they are comfortable with using it for the purposes previously discussed. One developer (*P40*) said: *"Preprocessor directives can be used to remove the most tedious and error-prone parts of programming. It [is] also the only C native way to conditionally compile the source code when runtime checks are unacceptable [due] to performance [overheads]. [Thus], there are no alternatives to the C preprocessor for this type of usage without using some tool outside the language."* Similarly, additional developers mentioned that in some cases, they really need to remove parts of the source code *(P1, P6, P23, P27)*. Otherwise, the code will not compile because of platform-specific parts that have not been removed. This leads them to argue that it does not matter which alternative one comes up with, one will need the C preprocessor at some point for such a platform-specific conditional compilation problem. Additionally, developers expressed concern that new technologies that replace the C preprocessor are likely not going to be present in all compilers *(P1, P6, P20)*. This shows hesitation to adopt third-party tools or alternate technologies (e.g., aspect-oriented programming [28; 15] or new macro languages such as ASTEC [23]) because of portability concerns.

SUMMARY

The developers interviewed do not see any current technologies that can entirely replace the C preprocessor. However, some developers routinely use alternate coding styles such as dividing functionality into separate files or functions (preferred by 60 %) and using run-time checks instead of `#ifdef` checks (preferred by 19 %).

Data Sources: Interviews, Survey, and Prior studies [105; 2]

RQ3: What are the common problems of using the preprocessor?

We now try to understand what problems, if any, do developers face while using the C preprocessor. We find that developers' comments generally align with the problems raised in the research literature. Specifically, developers mention the following problems: (1) dealing with configuration-related bugs, (2) testing an exponential number of configurations, and (3) difficulty with understanding code with too many `#ifdefs`.

Configuration-Related Bugs

Many developers confirm that bugs related to conditional compilation occur frequently *(P7, P18, P23, P35-P37)* or at least sometimes *(P8-P10, P13, P14, P27)*. Our interviewees list different types of bugs related to the use the C preprocessor, such as: incompatible macro selection *(P17)*; macros resulting in erroneous control flow *(P20, P22, P26)*; incorrect macro expansion *(P9)*; misspelled macro names *(P22, P23)*; missing variables and functions such as defining a variable in optional code and using it in mandatory code *(P13)*; type errors *(P8, P18, P23)*; syntax errors like missing control flow tokens, e.g., opening and closing brackets *(P24)*; linking problems *(P24)*; behavioral changes due to the interactions of macro *(P1, P9)*; memory and resource leaks, memory corruption, and race conditions *(P14)*; and incorrect use of `#else` and `#elif`. For instance, `#else` clause incorrectly treating some configurations, or use of `#elif` without an `#else` at the end to treat the default case *(P14)*.

Some interviewees *(P3, P20, P27)* argue that it is hard to deal with a high number of different macro combinations, which may ease the introduction of bugs. Developer *P1* points out that code that does not compile is easy to deal with, but the runtime bugs are the hardest ones to detect. Some developers *(P8, P10, P13, P20)* mention that even to detect simple compiler errors, someone has to compile the source code using the specific configuration that contains errors which is not that easy. The types of bugs developers mentioned align with various results from previous studies [79; 77; 78; 91; 18; 12]. Additionally, the fact that dealing with macro combinations is one of the sources of bugs is consistent with the findings of Iago et al. [16].

Since the data from our interviews is qualitative, we used our survey to quantify the frequency of C configuration-related bugs. We also asked developers about the difficulty of introducing configuration-related bugs when compared to other types of bugs as well as

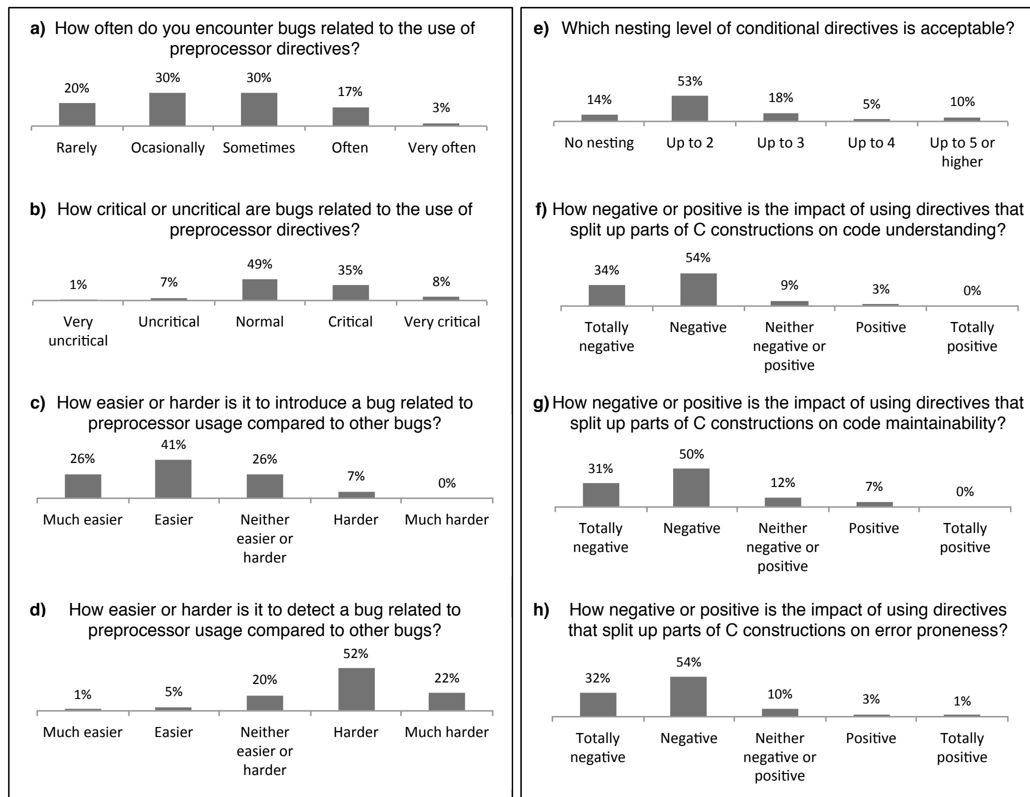


Figure 3.6: Results of our survey to quantify some findings of our interviews.

difficulty of detection. We present the results in Figure 3.6 (a-d), which can be summarized as follows. Even though developers believe that configuration-related bugs do not occur very frequently, see Figure 3.6 (a), they find that they are slightly more critical than the non-prerprocessor related bugs, see Figure 3.6 (b), and that they are easier to introduce, see Figure 3.6 (c) and harder to detect, see Figure 3.6 (d). Our survey findings are consistent with our prior work [17] that parsed all code of a *Linux* kernel (release `x86`, only) and did not find any syntax errors. On the other hand, when the same authors analyzed *BusyBox*, they only found a few type and linker errors that they reported to developers and which were fixed in subsequent releases [72]. This supports our findings that such types of errors may be rare, but are still important to fix nonetheless.

Both our interview and survey results suggest that, similar to researchers [17; 4; 16], practitioners perceive the C preprocessor as error-prone. However, developers did not mention having any tools that help them with avoiding such errors. Our findings suggest that we need further research on developing tool support to minimize bugs related to preprocessor usage and finding ways to make such tools attractive for developers to use.

Combinatorial Testing

Developers explain that another problem with using the C preprocessor is dealing with combinatorial explosions. As mentioned by developer *P19*, the use of conditional directives makes the code hard to test and debug since it increases the testing matrix. The number of configurations to test grows exponentially when developers add new preprocessor macros in `#ifdefs`. Assuming that there are n optional and independent macros, developers have 2^n different configurations. Furthermore, developers explain that they also need to consider different compilers, operating systems, and platforms, which is time-consuming and makes automation difficult. For these reasons, many developers *(P1-P6, P9, P17, P19, P20, P22, P23, P25)* mentioned that they normally do not test all different macro combinations due to time and resource constraints. They explain that they do not have an easy way to test all possible combinations.

Several developers *(P9, P17, P19, P20, P22, P23, P25)* mentioned that what happens in practice is that they check only a few configurations of the code. Moreover, some developers *(P11, P18, P24)* check only the default configuration on their own machine with all optional macros active. However, a few developers *(P1, P37)* mention that they additionally consider different platforms besides their own. They say that by compiling the source code with two or three different compilers and using 32 and 64-bit platforms, they are comfortable enough that the code is portable. Similarly, some developers *(P19, P20, P22)* said that they select specific configurations to test by setting different macro combinations manually.

This variation in testing style tells us that there is no systematic way to fully test such systems. We found that developers *(P2, P26)* often rely on end-users to test the source code using different platform configurations. Developer *P26* explains this as follows: *"I check whatever combinations I can, and some combinations can only be tested on systems to which I have no access. I rely on others to help out or just cross my fingers."* Developer *P2* echoes this, also stating that he heavily relies on his user base to report back errors. This result

aligns with a recent study on testing in the *Eclipse* platform by Greiler et al. [101]. Some developers *(P4, P10, P13)* realize that they use a narrow testing strategy and perceive it as a problem, expecting to find additional bugs if they are able to test more configurations. For example, one developer (*P26*) tell us: *"I do not find bugs related to preprocessor usage by running tests, but when running the tests with different combinations of macros."*

We found that testing in industry and in open source are different. While our open source interviewees repeatedly mention testing only a few configurations and relying on user testing, industry developers *(P31, P32, P38)* mention that they test the source code on all supported platforms with all macros active. Additionally, some industrial developers *(P33-P36)* state that they check all combinations and platforms supported. This difference can be explained by the lack of community involvement in the industry context and that the number of used configurations tends to be smaller (companies can restrict the supported configurations).

To overcome some of the challenges above, several developers *(P8-P12, P14, P31-P34)* mention that they use style checkers and static-analysis tools that often help them avoid bugs. This is especially true in industry projects. Our interviewees used the following tools: *Checkpath*, *Vera++*, *Coverity*, *Cppcheck*, *Valgrind*, *Coccinelle*, and *Lint*. Other developers *(P7, P13)* mentioned that they use at least *Gcc* with all warnings active. However, these tools consider only one configuration at a time, after preprocessing. Thus, these tools do not focus on bugs related to preprocessor usage. Some tools, e.g., *Cppcheck*, try to deal with many configurations by activating one macro at a time and performing the analysis to check the code several times, which is time-consuming. *Coccinelle* also handles variability to some extent by building a control-flow graph per function, where statement-level `#ifdefs` are taken into consideration. During the interviews, only one *Linux* developer (*P9*) mentioned a research tool, *Undertaker* [106], that can detect dead `#ifdef`-guarded blocks. However, developers did not mention any of the research tools that could analyze all configurations, such as *TypeChef* [17] or *SuperC* [38].

Code Comprehension

Our interviews suggest that many developers find it hard to understand code that is filled with `#ifdefs`. Developers *(P1, P5)* mentioned that the mixing of C code and directives interrupts the code logic since they are two independent languages. Developers *(P1, P5, P6, P19, P21, P25, P26)* believe that this mixing can obfuscate the source code making it harder

to read, comprehend, and maintain since it is difficult to determine which parts of the code are going to be compiled under which conditions. For example, some developers *(P1, P5, P23, P24)* complain about the use of fine-grained directives within function bodies. It requires the analysis of control flow structures (such as, `if`, `while`, `switch`, and `goto` statements) as well as `#ifdef`, `#ifndef`, `#if`, `#else`, and `#elif` directives. In addition, it becomes harder to understand the control flow, more difficult to check whether opening and closing brackets match, increases code complexity, and may lead to bugs. Additional developers *(P10, P18, P20, P24)* confirm this, saying that they avoid preprocessor directives because of readability problems. One developer (*P6*) specifically comments on this, saying *"My main problem is that [if] there are macros 7 layers deep[,] I don't understand them."*

We used our survey to gain further insight into the impact of C preprocessor directives on code comprehension and maintainability as shown in Figure 3.6 (e). We find that 14 % of our interviewees state that they prefer to avoid nesting preprocessor conditional directives altogether, 53 % do not mind using nesting up to level 2, and only 18 % find that three levels of nesting is still acceptable. Note that only a few developers find that deep nesting levels (i.e., those beyond three) are acceptable. Overall, implicitly, 85 % of the developers see that nesting levels beyond three should be avoided. This aligns with previous work that finds that the average nesting level across 40 analyzed C systems is approximately 1 [67].

SUMMARY

Developers face three configuration-related problems: (1) configuration-related bugs (do not appear often, but are perceived as more critical than other bugs), (2) combinatorial testing (conditional directives increase number of configurations to test), and (3) code comprehension (due to the cluttering of #ifdefs and C statements, and the deep nesting of #ifdefs).

Data Sources: Interviews, Survey, and Prior studies [67; 17; 4; 16; 79; 77; 78; 91; 18; 12]

RQ4: Do developers care about the discipline of directives?

The feasibility of introducing directives annotations is one of the most criticized aspects of the C preprocessor [3; 4; 12; 17; 13; 38; 81; 30], which is why we dedicate a separate

```
1. if (user_callbacks == NULL) {
2. #ifdef HAVE_PTHREAD
3.   callbacks=&ssh_pthread;
4. }
5. #else
6.   return SSH_ERROR;
7. }
8. #endif
```
(a)

```
1. if (user_callbacks == NULL) {
2. #ifdef HAVE_PTHREAD
3.   callbacks=&ssh_pthread;
4. #else
5.   return SSH_ERROR;
6. #endif
7. }
```
(b)

Figure 3.7: Disciplining a preprocessor directive.

research question for it. Our goal is to find out whether developers also view undisciplined annotations as problematic.

The majority of interviewees *(P1, P5, P17-P28, P32)* agree that the use of preprocessor directives to encompass individual tokens or parts of C syntactical units impacts the quality of code negatively. Developers emphasize that they would not use undisciplined annotations because they hinder source code readability *(P5, P17, P18, P22, P25, P26, P28)*, obfuscate control flow *(P1, P24)*, and make the code difficult to evolve and maintain *(P20, P22, P23)*. One developer (*P20*) elaborates on this, saying: *"I avoid this kind of directives, they make the source code hard to understand and maintain. My gut feeling keeps screaming possible bugs when I'm faced with a code like that."* Along the same lines, one of these developers recommends to discourage or disallow undisciplined annotations through code guidelines to avoid the aforementioned problems *(P26)*. Another developer *(P30)* stated that the use of code guidelines are important for the homogeneity of the project and that he often asks contributors to rewrite patches to follow the guidelines.

Despite the criticisms we received from most interviewees, some developers *(P4, P22, P31)* mention that they would still use undisciplined annotations in very specific cases. In such cases, they would also document the code to let others understand their reasoning. Furthermore, some developers (*P5, P7*) are reluctant to change undisciplined annotations once they exist. For instance, one developer (*P5*) states that: *"One thing is to not fix what is not broken. The problem is that to refactor a code, you have to understand [it]. If you do not understand [it], it is not easy to refactor. Many developers would say: I am not going to touch that."* Developer *P39* mentioned that while he believes it is good to fix undisciplined annotations, it has a very low priority. It is worth noting that none of the developers mentioned using tools to enforce disciplined annotations or identified a lack of tool support in general.

To generalize the findings of our study, we use the survey to quantify the influence of undisciplined directives on code understanding, maintainability, and error proneness as shown in Figure 3.6 (f-h). Our results show that developers generally agree that the use of undisciplined directives have a negative influence on source code understanding (88 %), maintainability (81 %), and error proneness (86 %). However, in a previous study, Schulze et al. [32] could not establish significant differences between disciplined and undisciplined directives from a program comprehension perspective in a controlled experiment. However, they observed that finding and correcting errors is a time-consuming and tedious task in the presence of preprocessor directives. Additionally, although several developers see the use of undisciplined directives negatively, other researchers [4] detected that almost 16 % of conditional directives are undisciplined directives.

To investigate this gap between developer preferences and perceptions and the reality in the code base, we performed an additional analysis of software repositories to identify the reasons why developers introduce undisciplined annotations. By analyzing 14 software repositories of our corpus, we detected that only 21 (7 %) out of 299 developers introduced almost 85 % of all undisciplined annotations we found in the software repositories. When we tried to contact these developers, some got defensive and excused their use of undisciplined annotations. For instance, one developer argues that, *"The code was actively rewritten at the time, and it often happens that first drafts of an idea ends up in poor code."*

Figure 3.7 (a) presents an example of an undisciplined annotation introduced in one of the C projects we examined. When we discussed this code fragment with its author, the author mentioned to prefer the equivalent code snippet in Figure 3.7 (b) as a replacement. Another developer who we contacted about undisciplined annotations stated to use such annotations to avoid cloning the source code as well as compiler warnings. Figure 3.8 (a) presents the undisciplined directive introduced by this developer. When discussing the code, the developer showed us the alternative in Figure 3.8 (b) that clones the source code (see lines 5 and 8) and another that generates compiler warnings as shown in Figure 3.8 (c), both of which seemed unacceptable to that developer. In this latter case, variable `failed` is always `true` when macro `USE_NTLM_AUTH` is not defined. In addition, this developer mentioned that since the undisciplined annotation in the original code was not repeated in many places, this minimizes potential problems. Such examples show that there may be situations where

```
1. #ifdef USE_NTLM_AUTH
2. if (priv->sso_available) {
3.    conn->state = SSO_FAILED;
4. } else {
5. #endif
6.    conn->state = NTLM_FAILED;
7. #ifdef USE_NTLM_AUTH
8. }
9. #endif
              (a)
```

```
1. #ifdef USE_NTLM_AUTH
2. if (priv->sso_available) {
3.    conn->state = SSO_FAILED;
4. } else {
5.    conn->state = NTLM_FAILED;
6. }
7. #else
8. conn->state = NTLM_FAILED;
9. #endif
              (b)
```

```
1. boolean failed = TRUE;
2. #ifdef USE_NTLM_AUTH
3. if (priv->sso_available) {
4.    conn->state = SSO_FAILED;
5.    failed = FALSE;
6. }
7. #endif
8. if (failed){
9.    conn->state = NTLM_FAILED;
10.}            (c)
```

Figure 3.8: Avoiding code clone and compiler warnings.

developers would prefer to use undisciplined annotations. In a previous study, We proposed alternatives to discipline annotations without cloning the source code [50]. However, they did not take compiler warnings into consideration. According to our data, compiler warnings seem to be a problem that may hinder the adoption of syntactical preprocessors despite of the existence of compiler attributes to ignore specific warnings.

SUMMARY

The majority of developers agree that the use of undisciplined directives influences code understanding, maintainability, and error proneness. However, there are cases where developers use undisciplined annotations to avoid code clones and compiler warnings.

Data Sources: Interviews, Survey, Mining Repositories, and Previous work [13; 12; 50; 32; 107; 30; 4]

3.2.3 Threats to Validity

We selected interviewees by sending email to developers and only those interested in the topic participated in our study. From 40 interviews, even though they cross 24 projects of different sizes and domains and 3 companies, it is difficult to generalize results. Nonetheless, we alleviated these threats by cross-validating with a survey from a larger population. Our survey could be filled out in around 10-15 minutes, which encouraged more developers to participate. Code snippets used in our survey might be misunderstood by developers or might conflate multiple issues; that is, related results can only be interpreted in the context of these snippets. To detect undisciplined directives, we used *Cppstats* [4], which is based on heuristics and may miss-classify a small number of directives, but we expect that this does not affect the bigger picture collected across multiple projects.

Chapter 4

A Sampling-Based Strategy to Detect Configuration-Related Bugs

In this chapter, we present the sampling-based strategy to detect configuration-related bugs with the purpose of better understanding this kind of bug. As the effectiveness of sampling depends significantly on how samples are selected, we propose and present the *Linear Sampling Algorithm (LSA)* to select configurations systematically. By using our strategy, we performed an empirical study on a corpus of 27 C real-world projects to investigate configuration-related bugs, including memory and resource leaks, dereferences of null pointers, and uninitialized variables.

We present the proposed strategy in Section 4.1. In Section 4.2, we describe a comparative study of 10 sampling algorithms proposed in previous studies. Based on the results of this study, we propose the *Linear Sampling Algorithm (LSA)*, as presented in Section 4.3. Last, in Section 4.4, we discuss an empirical study performed to investigate configuration-related bugs using our sampling-based strategy, *LSA*, and *Cppcheck*.

4.1 The Sampling-Based Strategy

To detect configuration-related bugs, we need to consider multiple configurations of the code. Checking every configuration individually is often infeasible, because real-world C projects have high numbers of preprocessor macros, leading to configuration spaces of exponential sizes. To tackle this scalability problem, we propose a sampling-based approach to select a subset of configurations to analyze. This way, we preprocess the code to systematically generate some configurations and analyze each selected configuration individually [47; 20].

The strategy receives as input the source code of the project, a sampling algorithm, the preprocessor macro constraints and build-system information available, and an analysis tool to check the source code. The constraints and build-system information are not required, though, as many C open-source projects do not have these pieces of information available. We illustrate our strategy in Figure 4.1.

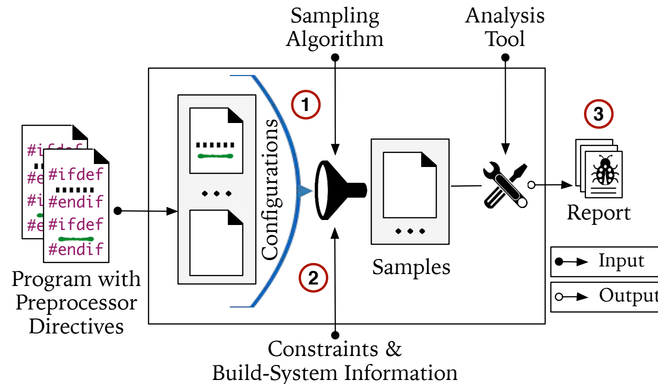


Figure 4.1: A sampling-based strategy to detect configuration-related bugs.

The strategy starts by selecting each source file of the project individually to perform a per-file analysis. We use a per-file analysis because a global analysis, considering the preprocessor macros of all source files, does not scale in real-world projects, as we discuss in Section 4.2.2. In this sense, the strategy considers the preprocessor macros of each source file separately. *Step 1* uses a sampling algorithm to select configurations systematically. In this step, our strategy can use different sampling algorithms.

Step 2 makes sure that we do not check invalid configurations according to constraints. For instance, the *Linux Kernel* uses two preprocessor macros (i.e., `X86_32` and `X86_64`) to represent 32 bit and 64 bit platforms respectively. There is a constraint that these macros are mutually exclusive, so that developers can select only one at a time. During this step, our strategy also receives build-system information, if available, to identify source files that are conditionally included depending on preprocessor macros.

Last, *Step 3* runs the analysis tool and presents a report. In this step, we run the tool for every source file of the project, once for each selected configuration. By definition, configuration-related bugs do not appear in all selected configurations. For bugs that appear in all selected configurations, we perform additional manual analysis to detect whether they are related to configurability or not.

To analyze the history of our subject projects, we extended our strategy to analyze several versions of the source code. For each set of files of a given commit, we apply our strategy to find configuration-related bugs. In the first commit, we analyze all files. In the following commits, we only consider the updated and added files. We avoid the overhead of analyzing files that have not been changed. Figure 4.2 illustrates this process. After selecting the source files to analyze (*Step A*), we use the strategy presented to find configuration-related bugs (*Step B*).

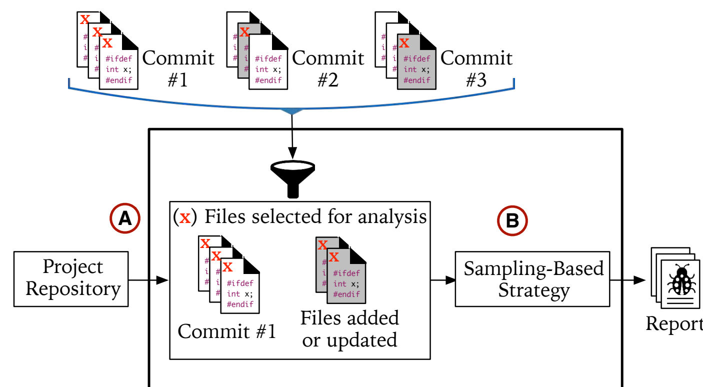


Figure 4.2: Analyzing different versions of the projects.

To select a suitable sampling algorithm for our sampling-based strategy, it is necessary to understand the tradeoffs, especially with regard to effort, i.e., how large are the sample sets, and bug-detection capabilities, i.e., how many bugs we find in the sampled configurations. This way, we present a comparison of 10 sampling algorithms and 35 combinations of these algorithms in the next section.

4.2 Comparison of Sampling Algorithms

Our overall goal is to compare state-of-the-art sampling algorithms regarding their capability to detect configuration-related bugs and the size of their sample sets. Furthermore, we study four assumptions of previous work, which often does not consider (1) constraints, (2) global analysis, (3) build-system information, and (4) header files.

In particular, we aim at answering the following research questions:

- **RQ1.** What is the number of configuration-related bugs detected by each sampling algorithm?
- **RQ2.** What is the size of the sample set selected by each sampling algorithm?
- **RQ3.** Which combinations of algorithms increase the number of bugs detected and minimize the number of configurations selected?
- **RQ4.** What is the influence of the four assumptions on the feasibility to perform the analysis for each sampling algorithm?
- **RQ5.** What is the influence of the four assumptions on the number of bugs detected by each sampling algorithm?
- **RQ6.** What is the influence of the four assumptions on the size of the sample set selected by each sampling algorithm?

4.2.1 Overall Study Design

At first glance, a study comparing sampling algorithms (RQ1–3) seems straightforward. We use a number of different sampling algorithms (independent variable) to measure how many of the bugs we can find with them in different software systems and how big the sample set is (dependent variables). However, there are several challenges to overcome in the design of such an experiment.

Sampling the configuration space needs to be combined with a technique to detect bugs in the respective selected configurations, such as inspection (which is unrealistically laborious), executing existing test suites (if available), automated test-case generation (look for crashing defects), or static analysis (prone to false positives). If not conduced carefully, we might be evaluating the bug-detection technique instead of the sampling algorithm. We address this potential bias by taking the bug-detection technique out of the loop and by using a corpus of previously found configuration-related bugs. For each known bug, we check whether the sampling algorithms select configurations in which the bug can be find, assuming a suitable bug-detection technique. By using a corpus of configuration-related bugs, we eliminate the

bug-detection technique as a confounding factor from our study setup. However, we actually do not know if the sampling algorithms actually discovered more or different bugs. We discuss this threat in Section 4.2.3, and an alternative study design in Section 4.2.2, which uses an analysis tool to detect new bugs.

A second design challenge is how to evaluate the influence of the assumptions (regarding global analysis, header files, constraints, and build-system information) on many sampling algorithms. As we will show, lifting these assumptions can make it infeasible to apply some of the algorithms to real-world software systems. Therefore, we decided to proceed in two steps: First, we study tradeoffs among algorithms (RQ1-3) under favorable conditions (i.e., fulfilling all assumptions). Subsequently, we investigate the influence of the assumptions (RQ4-6) on a smaller set of subject systems in a second study. The four assumptions are:

- **Constraints:** Constraints among macros may exclude certain configurations (e.g., macro X may only be selected if Y is selected) from the set of valid configurations. A sample set may contain configurations that violate constraints. Unfortunately, macro constraints are rarely documented explicitly. The *Linux Kernel* is an exception and has been studied therefore extensively [108; 91]. In the presence of macro constraints, sample sets are often larger to achieve the same code coverage, and highly optimized covering array tables[1] cannot be used. Since we do not know macro constraints for most of our subject systems, we exclude them entirely from the sampling process in our first study.
- **Global analysis:** We can select configurations per file or globally for the entire system. Even in systems with many preprocessor macros, individual files are usually affected only by few macros. Sampling over the global configuration space may detect inter-file bugs (e.g., linker issues), but this often creates huge sample sets, which hardly affect individual files. Thus, in the first study, we assume a per-file analysis.
- **Header files:** In C code, a significant amount of variability arises from header files. However, detecting all preprocessor macros from header files in a sound way is often

[1] A covering array is a mathematical object used for software testing, which ensures specific coverage criteria. For example, a *pair-wise* covering array ensures that all pairs of preprocessor macros are considered by the array [109; 36].

difficult and expensive, which requires some form of variability-aware analysis [110; 71; 17; 111]. It is necessary to resolve includes and macro expansions, but to keep the conditional directives. We therefore analyze only preprocessor macros within the source files in our first study.

- **Build system:** The build system may induce a significant amount of variability, such that certain files are not compiled in all configurations [108; 111]. Since build systems are inherently difficult to analyze [112], we do not use build-system information in the first study.

In both studies, we analyzed the same set of 10 sampling algorithms: five variations of *t-wise*, *statement-coverage*, *random*, *most-enabled-disabled*, *one-enabled*, and *one-disabled*, proposed in prior work [16; 73; 34; 35; 36; 37; 47], and their combinations. We explained these sampling algorithms in Section 2.4.

Detecting Bugs

In the first study, we compared the bug-detection capabilities and the sample sizes of the 10 sampling algorithms using a corpus of 135 known bugs of 24 systems to answer questions **RQ1–3**. As we explained, we performed this study under favorable assumptions, that is, without constraints, global analysis, build-system information and header files.

We proceeded in three steps, as illustrated in Figure 4.3. In *Step 1*, we select each source file of the subject system. *Step 2* applies each sampling algorithm to select the samples for every file. *Step 3* determines the number of configuration-related bugs detected (**RQ1**) and measures the size of the sample set (**RQ2**) for each sampling algorithm. The size of the sample set is the sum of the numbers of sampled configurations for every source file. To identify the algorithms that detect a bug, we consider the bug presence condition, which is a subset of system configurations in which the bug can be found [113] (see Section 2.2.3), assuming a suitable bug-detection technique. We checked whether we could find at least one configuration of this subset in the sampled configurations for each algorithm. Finally, we repeat the process for combinations of sampling algorithms (**RQ3**).

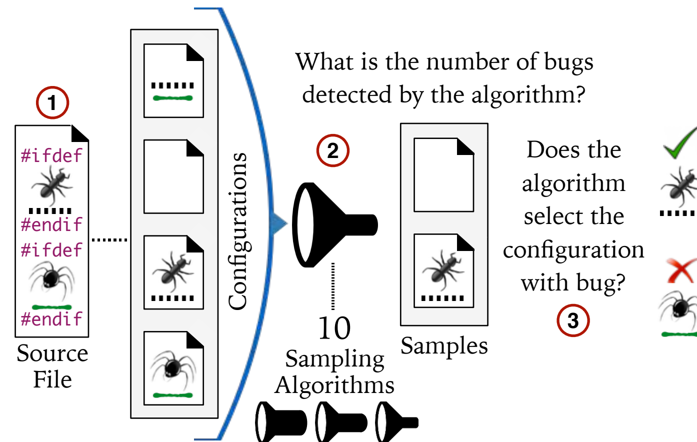


Figure 4.3: Strategy used to compare the sampling algorithms.

Corpus of Bugs

Using a corpus of configuration-related bugs in a study raises the question of how to acquire a proper corpus and whether it is a representative corpus of configuration-related bugs in real systems. Bugs identified with existing sampling algorithms will obviously bias results toward these specific algorithms. Instead, we assembled a corpus of bugs in which all bugs have been identified in one of two ways:

- Variability-aware analysis tools are able to identify certain kinds of bugs, i.e., mostly syntax and type errors, by covering the entire configuration space without sampling. Difficulties in setting up these tools and narrow classes of detectable bugs limit their applicability at this point. We collected only configuration-related bugs that have been reported by such tools, reported to the original developers, and confirmed or fixed by the system's developers [17; 12].

- We used configuration-related bugs that have been identified and fixed by developers. Bugs reported by users and fixed in the repository by the system's developers may be slightly biased toward more popular configurations, but are not systematically biased toward specific sampling algorithms. They represent configuration-related bugs that are routinely detected and fixed in real systems. We started with Abal's corpus of bugs of the *Linux Kernel* [16], and complemented it with bugs found in other studies [40; 19], and our own investigation of software repositories (see Table 4.1).

Table 4.1: Configuration-related bugs considered in our first study.

Source	**Bugs**	**Strategy**	**Subject System (number of bugs)**
[16]	30	Repository mining	*Linux* (30)
[17]	10	*TypeChef*	*BusyBox* (10)
[19]	5	Repository mining	*Gcc* (3), *Firefox* (2)
[40]	3	Repository mining	*Gnome-keyring* (1), *Gnome-vfs* (1), and *Totem* (1)
[12]	22	*TypeChef*	*Apache* (3), *Bash* (2), *Dia* (2), *Gnuplot* (5), *Libpng* (3), and *Libssh* (7)
-	65	Our repository mining	*Apache* (9), *Bison* (2), *Cherokee* (3), *Cvs* (1), *Dia* (1), *Fvwm* (10), *Gnuplot* (5), *Irssi* (4), *Libpng* (1), *Lua* (1), *Libssh* (10), *Linux* (7), *Libxml* (2), *Lighttpd* (1), *Vim* (5), *Xfig* (1), and *Xterm* (2)
Total	135		

Overall, the corpus of bugs used in our first study includes 135 configuration-related bugs from 24 subject systems of various sizes and from different domains, over 125 different files with distinct numbers of preprocessor macros (see Figure 4.4). Our corpus contains bugs of different kinds, including syntax errors (34%), memory leaks (22%), null-pointer dereferences (17%), uninitialized variables (13%), undeclared variables and functions (5%), resource leaks (3%), array and buffer overflows (3%), arithmetic problems (2%), and type errors (1%). Table 4.2 presents a characterization of the subject systems we used in the first study, listing the project name, application domain, lines of code, number of files, number of preprocessor macros and number of known bugs considered in our study.

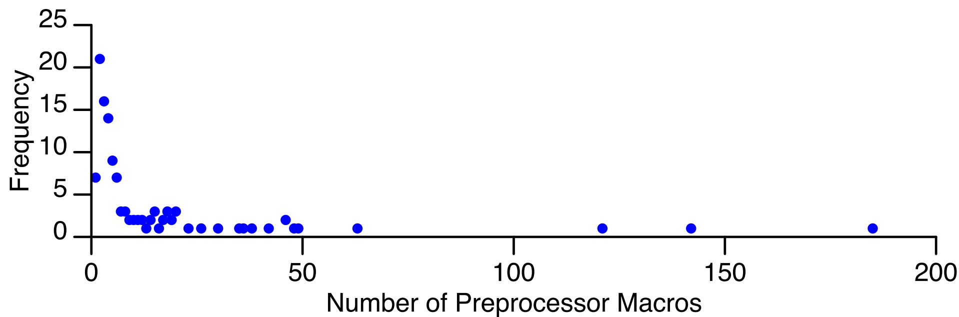


Figure 4.4: Number of distinct preprocessor macros in files with bugs.

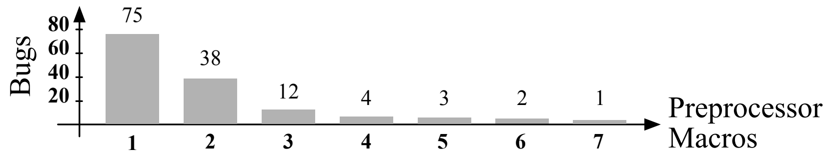
Table 4.2: Project characterization and the total number of known bugs.

Project	Domain	LOC	Files	Macros	Bugs
Apache	Web server	144 768	362	700	12
Bash	language interpreter	44 824	138	1427	2
Bison	parser generator	24 325	129	269	2
Busybox	UNIX utilities	189 722	805	1418	10
Cherokee	Web server	63 109	346	452	3
Cvs	version control system	76 125	236	628	1
Dia	diagramming software	28 074	132	307	3
Firefox	Web browser	6 017 673	22 423	17 415	2
Fvwm	windows manager	102 301	270	301	10
Gcc	C/C++ compiler	1 946 622	22 034	3825	3
Gnome-keyring	daemon application	76 525	376	213	1
Gnome-vfs	file system library	78 380	286	427	1
Gnuplot	plotting tool	79 557	152	500	10
Irssi	IRC client	51 356	308	157	4
Libpng	PNG library	44 828	61	327	4
Libssh	SSH library	28 015	125	115	17
Libxml	XML library	234 934	162	2126	2
Lighttpd	web server	38 847	132	215	1
Linux	operating system	12 594 584	37 520	26 427	37
Lua	language interpreter	14 503	59	145	1
Totem	movie player	31 596	135	84	1
Vim	text editor	288 654	178	942	5
Xfig	vector graphics editor	70 493	192	143	1
Xterm	terminal emulator	50 830	58	501	2
Total					**135**

Table 4.3 shows the presence conditions of the bugs and the number of preprocessor macros that we need to enable or disable to detect the configuration-related bugs we consider in the first study: for 78 bugs (58%), we need to enable some macros; for 27 bugs (20%), we need to disable some preprocessor macros; and for another 30 bugs (22%), we need to enable some macros and disable others. The majority of bugs (83%) are related to one or two preprocessor macros, while less than 5% are related to more than four preprocessor macros.

Table 4.3: Presence conditions of the configuration-related bugs.

Some preprocessor macros enabled	78 (58%)
a	59
$a \wedge b$	13
$a \wedge b \wedge c$	5
$a \wedge b \wedge c \wedge d \wedge e$	1
Some preprocessor macros disabled	27 (20%)
$!a$	16
$!a \wedge !b$	8
$!a \wedge !b \wedge !c$	1
$!a \wedge !b \wedge !c \wedge !d$	1
$!a \wedge !b \wedge !c \wedge !d \wedge !e \wedge !\text{f} \wedge !g$	1
Some macros enabled and some disabled	30 (22%)
$(!a \wedge b) \vee (a \wedge !b)$	17
$(a \wedge b \wedge !c) \vee (!a \wedge !b \wedge c)$	6
$(a \wedge b \wedge !c \wedge !d) \vee (a \wedge b \wedge c \wedge !d)$	3
$(a \wedge b \wedge c \wedge d \wedge !e) \vee (!a \wedge !b \wedge !c \wedge !d \wedge e)$	2
$a \wedge !b \wedge !c \wedge !d \wedge !e \wedge !\text{f}$	1
$a \wedge b \wedge !c \wedge !d \wedge !e \wedge !\text{f}$	1



We discarded seven bugs of the *Linux Kernel* from our corpus that span multiple files, because we performed a per-file analysis in our first study. We considered bugs that require inter-procedural analysis, as long as all procedures are defined in the same file.

4.2.2 Results and Discussion

For each sampling algorithm, we answered research questions **RQ1–2**. Figure 4.5 presents the number of bugs detected and the corresponding size of the sample set for each algorithm. However, note that detecting more bugs does not mean more efficiency, because there is a tradeoff between the number of bugs detected and size of sample set. We define an efficiency function in terms of *Efficiency*: $E = \mathit{SizeOfSampleSet} / \mathit{NumberOfBugs}$. This function returns the number of configurations that one needs to check per bug to be detected. In addition, we analyzed 35 combinations of algorithms to answer research question **RQ3**,

as illustrated in Figure 4.6. All data used in this study are available on our Website.[2] We discuss the results in terms of these research questions next.

RQ1. What is the number of configuration-related bugs detected by each algorithm?

Overall, we found that all algorithms detected at least more than 66% of all bugs of our corpus. *Statement-coverage* detected the lowest number of bugs, while *six-wise* detected the highest number. The majority of bugs in our corpus can be detected by enabling or disabling fewer than six preprocessor macros. In this way, *six-wise* is able to detect all these bugs. There is one bug for which developers need to disable seven preprocessor macros for triggering it, which *six-wise* detected by chance, as *six-wise* does not check all combinations between seven configuration options. *Statement-coverage* missed 45 bugs because they require developers to enable some macros and disable others (i.e., require specific combinations of multiple blocks of codes), whereas *statement-coverage* is only concerned with including each block of code at least once in a system configuration.

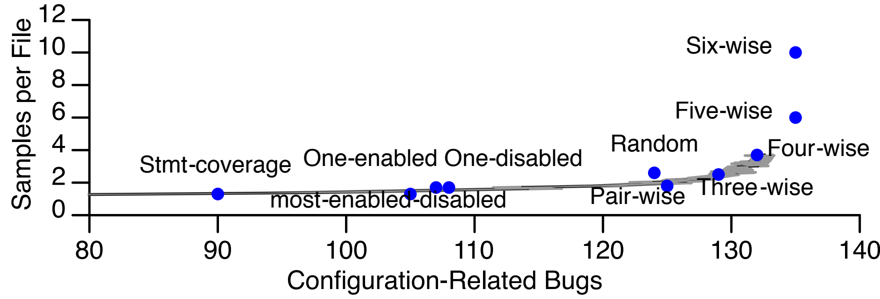
All *t-wise* algorithms detected more than 92% of the bugs. *Six-wise* and *five-wise* detected all bugs. *Most-enabled-disabled*, *one-enabled*, and *one-disabled* detected all between 78% to 80% of the bugs. Furthermore, we present the average values of random sampling with a 95% confidence interval (gray area) in Figure 4.5. We ran random sampling with the maximum number of configurations per file (n) ranging from 1 to 40, ten times for each value of n.[3] We report the mean of all runs; it detected 124 (92%) bugs.

RQ2. What is the size of the sample set selected by each sampling algorithm?

The sizes of the sample sets range from 1.3 to 10 configurations per file. The algorithm *most-enabled-disabled* selected the smallest sample set; *six-wise* required the largest sample set (with more than 500K sampled configurations considering all projects). The number of configurations to check influences the time of analysis. It is not feasible to use algorithms with large sample sets in all cases. Developers can use the results presented in Figure 4.5 to select the sampling algorithm that fits their needs best. For instance, during initial phases of a project, when developers are changing the source code frequently, they may prefer sampling algorithms with small sample sets to run the analysis fast. At some point, such as before a

[2] http://www.dsc.ufcg.edu.br/~spg/sampling/

[3] Random selects 2.6 samples per file, on average, because the subject systems of our corpus contain many source files without preprocessor macros, and others with only few macros.



Sampling Algorithm	Bugs	Samples	Sampling Algorithm	Bugs	Samples
Statement-coverage	90	1.3	Pair-wise	125	1.8
Most-enabled-disabled	105	1.3	Three-wise	129	2.5
One-enabled	107	1.7	Four-wise	132	3.7
One-disabled	108	1.7	Five-wise	135	6.0
Random	124	2.6	Six-wise	135	10.0

Figure 4.5: Number of bugs and samples per file for each algorithm.

release, developers might want to use algorithms with larger sample sets, to minimize the number of configuration-related bugs.

Based on our efficiency measure, we rank the sampling algorithms starting from the most efficient: *most-enabled-disabled*, *pair-wise*, *stmt-coverage*, *one-disabled*, *one-enabled*, *three-wise*, *random*, *four-wise*, *five-wise*, and *six-wise*.

RQ3. Which combinations of sampling algorithms increase the number of bugs detected and minimize the number of configurations selected?

In addition to the individual algorithms, we analyzed combinations (that is, the union of the sample sets of the algorithms) of two and three sampling algorithms, excluding *random*, *five-wise*, and *six-wise* algorithms. We excluded *five-wise* and *six-wise* because they already detected all 135 bugs, and we excluded *random* because it detects different numbers of bugs in different runs. Furthermore, we excluded combinations with more than three algorithms, because they resulted in inefficient combinations according to our efficiency function.

Figure 4.6 relates the number of bugs and the size of sample sets for all combinations of sampling algorithms. Based on the results, we determined the *Pareto Front* [114] to illustrate tradeoffs between number of bugs and size of the sample sets. Figure 4.6 also presents the

most four efficient combinations of sampling algorithms on the *Pareto Front*: *C1*, *C2*, *C3*, and *C4*. On the other hand, the combination of *one-enabled*, *one-disabled* and *four-wise* is the less efficient. The combination of sampling algorithms *C3* represents LSA algorithm, as presented in Section 4.3.

Sampling Algorithm		Bugs	Samples
C1	Pair-wise and one-disabled	131	3.5
C2	One-enabled, one-disabled, and statement-coverage	132	4.8
C3	One-enabled, one-disabled, and most-enabled-disabled	133	4.8
C4	One-enabled, one-disabled, and pair-wise	134	5.3

Figure 4.6: Number of bugs and samples per file for the combination of algorithms.

SUMMARY

All algorithms are able to detect at least 66% of the configuration-related bugs; most-enabled-disabled, pair-wise and statement-coverage are the most efficient algorithms; some combinations of algorithms provide an useful balance between sample size and bug-detection capabilities.

Checking Assumptions

In the first study, we made simplifying assumptions that have been made also in previous work on sampling: We ignored constraints, header files, and build-system information, and we did a per-file analysis only. In more realistic conditions, these assumptions often do not hold: For example, constraints often exist and ignoring them may lead to false positives, but constraints are rarely documented systematically and therefore easily ignored. Similarly information from the build system may increase precision but build systems are inherently

difficult to analyze [110; 111]. The simplifying assumptions allow one to apply sampling algorithms quickly to a large set of systems, as we did in our first study, but their influence on practicability and effectiveness is not well understood. Therefore, in a second study, we explore the effect of each assumption on the efficiency of the sampling algorithms.

In an exploratory setting, we replicate the first study for a subset of the corpus, with the purpose of investigating how the assumptions affect each sampling algorithm (**RQ4–6**). To increase internal validity, we considered each assumption separately as an independent variable that we manipulate to understand the influence of each assumption on sampling. We limit the second study to bugs of the *Linux Kernel* and *BusyBox* (47 bugs from the first study), because these subject systems are the only ones for which we have build-system and constraint information from the *LVAT* and *TypeChef* projects [115; 116; 117]. For the *Linux Kernel*, we consider additionally seven known bugs that cross files, which we excluded from our original corpus, as we discussed in Section 4.2.1.

Table 4.4 summarizes the number of configuration-related bugs detected, sizes of sample sets, and the ranking of sampling algorithms per lifted assumption.

Constraints

Constraints exclude certain combinations of preprocessor macros (e.g., macro X must be selected if macro Y is selected) from the set of valid configurations. Bugs identified in invalid configurations are considered false positives (which did not occur in the first study, because we consider only a corpus of true positives); hence sampling invalid configurations adds no value. The analyzed version of the *Linux Kernel* has 293,826 constraint clauses among its preprocessor macros; *BusyBox* has 615.

In the original sample sets of the first study, many sampled configurations are actually invalid in these highly constrained configuration spaces. For instance, in our study, *random* selects 24% of valid configurations and the percentage goes up to 43% when picking *most-enabled-disabled*. Sampling within such constrained spaces is more challenging for all sampling algorithms, as solvers or search-based strategies are needed. We incorporate constraints as follows:

- *Most-enabled-disabled:* We cannot simply enable all preprocessor macros if some of them are mutually exclusive. Instead, we use a solver to find two valid configurations with the maximum number of preprocessor macros enabled and disabled. If there are

multiple optimal solutions, we pick the first offered by the solver.

- *One-enabled-disabled:* Similarly, for each macro, we use a solver to identify the valid configuration that disables/enables the most other macros.

- *Random sampling:* We randomly assigned `true` or `false` for every preprocessor macro inside a file and discard invalid assignments until we find the desired number of configurations. Truly random sampling in large constrained spaces with many macros is still a research problem though, with recent progress in theory [118] and recent pragmatic search heuristics [119].

- *Statement-coverage:* To select a minimal set of covering configurations, we need to consider constraints. Conceptually we can use the original implementation of *statement-coverage*, as part of *Undertaker* [73], as in our first study, but the tool is not flexible to handle other projects than *Linux*. Thus, we used an implementation that we created in a previous work [47].

- *T-wise sampling:* The covering array tables used in the first study are precomputed, optimal solutions that, however, assume independence of all macros. Recent research investigated strategies to generate *t-wise* covering arrays for constrained configuration spaces, such as *SPLCATool* [36], *CASA* [120], and *ACTS* [121]. All tools use heuristics and may produce larger-than-optimal sampling sets and the sample sets produced may not actually achieve full t-wise coverage. To generate the *pair-wise* covering array, we used *SPLCATool*. We failed to generate *three-wise* or even higher covering arrays for the *Linux Kernel*: Even with 120 Gb RAM we ran out of memory; a developer from *CASA* estimated that the generation could take months and would require a 1.6 terabyte array to track the covered macros. Overall, we could not find an alternative to implement the *three-wise*, *four-wise*, *five-wise* and *six-wise* algorithms considering constraints; existing approaches are intractable for the size and complexity of *Linux*.

The changes in sampling algorithms to incorporate constraints changed the efficiency of the algorithms as summarized in Table 4.4. Most affected were t-wise strategies: *Pair-wise* required a larger sample set and detected fewer bugs (including bugs that *pair-wise* should have guarantee to find) from the *Linux Kernel*, because the used heuristics are unsound and

do not cover all valid pairs of preprocessor macros. *Three-wise* and beyond sampling was not tractable at all.

The time to compute sample sets increases significantly when adding constraints. Our use of a SAT solver required significant additional time and memory to generate the sample sets. On average, we created sample sets for each file in 0.04 seconds without constraints, while the analysis with constraints took 0.75 seconds per file, on average. This time represents an increase from 15 minutes to over 4 hours for the *Linux Kernel*. Regarding the ranking of algorithms, *most-enabled-disabled* and *statement-coverage* remain at top positions (see Table 4.4); the *t-wise* algorithms dropped significantly or were not feasible at all.

SUMMARY

When considering constraints, we substantially reduce false positives; but high costs for generating sample sets, which are often not optimal; it is infeasible for three-wise and higher at large scale.

Global Analysis

To perform global analysis, we created a single sample set across all files instead of a distinct set per file. Such global set allows us to perform cross-file analysis to find bugs that cannot be identified on a per-file basis, such as linking problems. However, for global analysis, a sampling algorithm needs to consider all macros in the system, not just the subset of macros used in each file.

We were not able to generate global sample sets with any *t-wise* algorithm at the scale of our subject systems. The largest precomputed tables we found covered up to 2000 macros (pair-wise) or 191 macros (six-wise). We are not aware of any tool that has the capability to generate covering arrays for such a large number of preprocessor macros, even without constraints. *Statement-coverage* also turns intractable, as it requires to solve the coverage problem considering all source files of the project (i.e., equivalent to concatenating all source code into a single file and finding a set of configurations that enabled all optional code blocks at least once). *One-enabled* and *one-disabled* require substantially larger sample sets as more macros are considered (from 1.7 to almost 8K). Also *random* requires larger sample sets, on average, because previously we could use smaller sample sets when the file had only few macros (see Section 4.2.1). *Most-enabled-disabled* is the only algorithm for which the size

of sample sets was not influenced, because it is not sensitive to the number of macros, that is, it always selects exactly two configurations.

To explore the ability of global analysis to identify non-modular bugs, we analyzed 7 known bugs of the *Linux Kernel* [16] that span multiple files, which we had to exclude from our first study. We detected all seven bugs by applying *one-enabled* and *one-disabled* with global analysis. *Most-enabled-disabled* detected five (71%) out of the seven bugs, and *random* detected four (57%) bugs. The other algorithms are not feasible when applying global analysis.

SUMMARY

Using a global analysis, we can potentially detect non-modular bugs that span multiple files; it causes an explosion in the number of considered preprocessor macros that leads to large sample sets; too large for t-wise and statement-coverage.

Header Files

In C source code, variability may be introduced by header files because macros used in `#ifdefs` can have non-local effect. If sampling is applied only to variability in the main source file, it may not detect bugs stemming from variability in header files. For example, a function may not be declared in all configurations of the header, a type name may be defined as either `int` or `long` depending on configuration decisions in the header, or a macro may be defined in the header only in some configurations. Precisely analyzing header variability is challenging, though, due to the interaction of file inclusion with conditional compilation and macros. Precise analyses exist [17; 38], but are challenging and time-consuming to use, because one needs to set up the environment with all header files used by the project.

Incorporating header files increases the number of macros per file significantly. Whereas the files of the *Linux Kernel* contain, on average, 3 distinct macros when ignoring variability from header files, headers add another 238 distinct preprocessor macros, on average. This increases the size of the sample set for all algorithms, except for *most-enabled-disabled*. For *statement-coverage*, *five-wise*, and *six-wise*, our subject systems reach configuration spaces for which these algorithms become intractable.

Since our corpus does not include bugs caused by misconfigurations from header files, most sampling algorithms detect the same bugs. The *one-enabled* algorithm detected more bugs, because including preprocessor macros from headers allowed it to disable more macros, while enabling one at a time.

SUMMARY

When incorporating header files, there is a potential to detect additional bugs from header files; but a difficult setup and much larger sample sets (if feasible at all), which lead to ranking changes.

Build-System Information

The build system controls which files are compiled and included. Files may be included only when specific preprocessor macros are selected or may be compiled with additional parameters. This is equivalent to wrapping an additional `#ifdef` around each source file or define additional macros in the beginning of a file. Like ignoring constraints, ignoring build-system information can lead to false positives where bugs are reported in configurations that are prevented in practice by the build system.

Build systems often have a strong influence on the configurability of a system; in the *Linux Kernel*, for example, 97% of source files are compiled only in certain configurations, and 80% in *BusyBox*. Still, extracting configuration knowledge from build systems is very difficult in the general case. While *Linux* and *Busybox* have been addressed with specialized parsers that recognize common patterns [116; 117], and more modern build systems use a more declarative style, which is easier to analyze [122], analyzing Make files in general is an open research problem with only few initial solutions [123; 124].

Considering build-system information, the presence conditions of bugs become more complex, because we include the condition when the file is compiled: Whereas without build system information 40 % of bugs in our corpus can be found by enabling or disabling a single preprocessor macro, only 17 % can be found the same way when considering build-system information. By requiring more macros to pinpoint bugs, incorporating build-system information decreases the efficiency of algorithms. *Pair-wise*, *three-wise*, *most-enabled-disabled*, and *one-enabled* detected fewer bugs than in the first study.

The sizes of the sample sets are slightly increased in all sampling algorithms (except *most-enabled-disabled*), as we consider additional preprocessor macros used in the build system. Time required to compute sample sets is increased only by a few milliseconds.

SUMMARY

When including build-system information, the analysis considers a few more macros, but no significant changes.

Experiment: Cppcheck Warnings

The goal of this experiment is to compare the sampling algorithms (**RQ1–3**) by using a different perspective. Instead of measuring bug-detection capabilities in terms of a corpus of known configuration-related bugs, we use a static-analysis tool (i.e., *Cppcheck*) as our automated bug-detection mechanism.

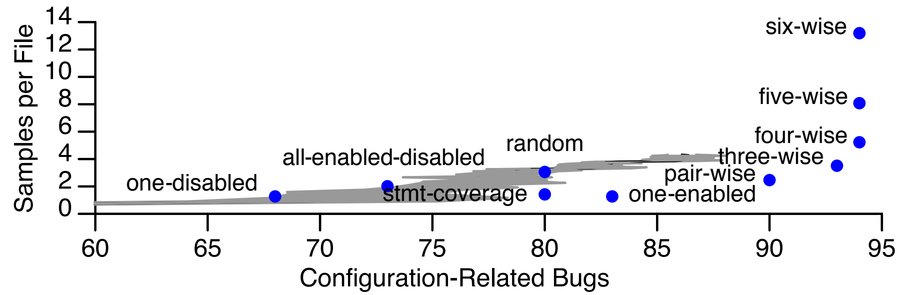
They key difference to *Study 1* is how we operationalize the dependent variable with regards to bug-detection capabilities. Unfortunately, there is no tool that would produce a reliable ground truth.[4] We run *Cppcheck* on each sampled configuration of each file and count all reported warnings. We discard warnings that occur in all configurations, because they are not configuration related. Although a warning does not necessarily correspond to a bug, it provides a rough estimate of the number of issues a developer needs to investigate, and *Cppcheck* also claims to minimize false positives. We assume that the distribution of warnings throughout the code is roughly similar to the distribution of real bugs in C code and can hence serve as a proxy to measure how configuration-related bugs are distributed over the configuration space.

We performed this experiment on a fresh set of subject systems, that does not overlap with the corpus of *Study 1*: *expat*, *flex*, *gimp*, *gnumeric*, *gzip*, *kindb*, *mplayer*, *mpsolve*, *mptris*, *openldap*, *parrot*, *prc-tools*, *privoxy*, *sylpheed*, *tk*, *xine-lib*. We selected these systems guided by previous work [4; 67], which studied projects statically configurable with the preprocessor.

Overall, *Cppcheck* reported 96 warnings that appear only in specific configurations of the code over 77 distinct files. All 10 sampling algorithms reported more than 70% of the

[4] Variability-aware analysis tools, such as *TypeChef* [17; 72] and *SuperC* [38], could soundly cover all configurations regarding syntax or type errors, but would require a time-consuming initial setup that would make our study infeasible.

96 configuration-related warnings, and no sampling algorithm reported all 96 warnings. We summarize the results of this experiment in Figure 4.7. *Five-wise* and *six-wise* reported the highest number of warnings again. *One-disabled* and *statement-coverage* reported the lowest number of warnings. There is a warning for *Xine-lib*, where developers need to disable eight distinct macros to make *Cppcheck* report it. *Six-wise* misses this warning. However, other sampling algorithms, such as *most-enabled-disabled* and *one-enabled*, reported the warning for *Xine-lib*. Furthermore, we computed the number of warnings reported for the combinations of sampling algorithms and found combinations that reported all 96 warnings (e.g., *C2* and *C3*), as depicted in Figure 4.8.



Sampling Algorithm	Bugs	Samples	Sampling Algorithm	Bugs	Samples
One-disabled	68	1.3	Pair-wise	90	2.5
Most-enabled-disabled	73	2.0	Three-wise	93	3.5
Statement-coverage	80	1.4	Four-wise	94	5.2
Random	80	3.1	Five-wise	94	8.1
One-enabled	83	1.3	Six-wise	94	13.2

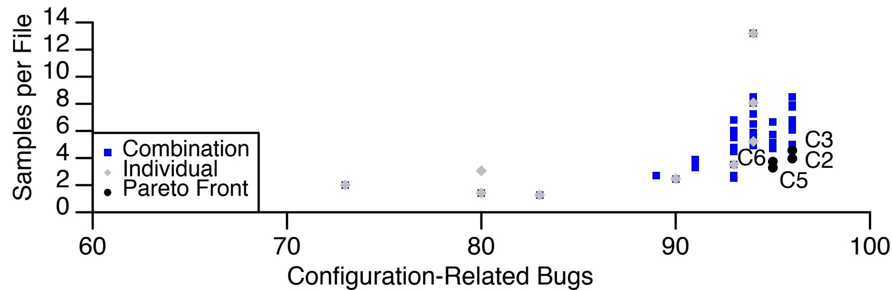
Figure 4.7: Number of warnings reported and samples per file for each algorithm.

The sizes of sample sets range from 1.3 to 13.2 configurations per file. Again, *six-wise* selected the highest number of configurations (more than 100K across all projects), while *one-enabled* and *one-disabled* selected the lowest number of configurations. The majority of the combinations of algorithms created a very large sample set. Figure 4.8 presents four combinations of sampling algorithms on the *Pareto Front*: *C2*, *C3*, *C5*, and *C6*.

We computed the ranking of sampling algorithms considering the efficiency function of Section 4.2.2. The algorithms, starting from the most efficient, are: *one-enabled*, *stmt-coverage*, *one-disabled*, *pair-wise*, *most-enabled-disabled*, *three-wise*, *random*, *four-wise*,

five-wise, and *six-wise*. Overall, the ranking is stable when compared to *Study 1* and there were only minor changes: *most-enabled-disabled* and *pair-wise* are less efficient here, while *one-enabled*, *one-disabled*, and *statement-coverage* are more efficient. These changes can be explained by analyzing the number of files with only one preprocessor macro, which is higher in our experiment than in *Study 1*. *Most-enabled-disabled* requires two configurations for each file with one preprocessor macro; *one-enabled* and *one-disabled* require only one configuration per file. It makes *one-enabled* and *one-disabled* more efficient and impacts the ranking. Regarding the five least efficient algorithms, the ranking is exactly the same as in the first study on our corpus.

Study 1 and this experiment complement and confirm each other, as we obtain essentially the same results regarding the bug-detection capabilities of the sampling algorithms by using different perspectives: known bugs reported in previous studies (*Study 1*) and *Cppcheck* as our bug-detected mechanism. We found two combinations of sampling algorithms (*C2*, and *C3*) that are on the *Pareto Front* of *Study 1* as well, which support them as efficient combinations. By triangulating the results, we gain confidence in the findings.



Sampling Algorithm		Bugs	Samples
C2	One-enabled, one-disabled and statement-coverage	96	4.0
C3	One-enabled, one-disabled and most-enabled-disabled	96	4.6
C5	One-enabled, and most-enabled-disabled	95	3.3
C6	Pair-wise and one-enabled	95	3.7

Figure 4.8: Number of warnings and samples per file for the combinations of algorithms.

SUMMARY

This experiment yielded comparable results and complements and confirms the first study on our corpus of 135 known bugs, as we obtain essentially the same results regarding the bug-detection capabilities of the sampling algorithms by using different perspectives: known bugs reported in previous studies and Cppcheck as our bug-detected mechanism.

4.2.3 Threats to Validity

In this section, we discuss a number of threats to validity that are crucial for our study.

Internal validity

Regarding internal validity, the corpus of bugs is critical for our research study. Creating a representative corpus is difficult, primarily because we have no means of knowing all bugs in the system, because we do not have a comprehensive quality assurance strategy in the first place. We address this threat with two strategies:

- We avoided biasing our corpus to any specific sampling algorithm. As the corpus has been partially mined from software repositories, it might be biased towards more popular system configurations. Still, our corpus is the most comprehensive corpus of configuration-related bugs we are aware of.

- We conducted a complimentary experiment using an automated bug finding technique instead of a corpus of known bugs, as presented in Section 4.2.2. This experiment yielded comparable results, complements, and confirms the first study on our corpus. In a nutshell, we measured which sampling algorithm the bug finding technique, static analysis with *Cppcheck*, would expose the most warnings per sampled configuration. This experiment introduces a different threat in terms of false positives, however, by triangulating the results across both setups with orthogonal threats to validity increases confidence in our findings.

External validity

Regarding external validity, we studied only subject systems that implement variability with conditional compilation and cannot generalize to subject systems that use other mechanisms to implement variability.

Table 4.4: Number of bugs, size of sample sets and ranking of algorithms.

Algorithms	**Constraints**			**Global Analysis**		
	Bugs	Configs	Rank	Bugs	Configs	Rank
Pair-wise	33 ↓	30 ⇑	5	–	–	–
Three-wise	–	–	–	–	–	–
Four-wise	–	–	–	–	–	–
Five-wise	–	–	–	–	–	–
Six-wise	–	–	–	–	–	–
Most-enabled-disabled	23 ↓	1.4 =	1	27 =	1.4 =	1
One-enabled	30 ↑	1.1 ↓	3	31 ↑	7943 ⇑	3
One-disabled	38 ↓	1.1 ↓	4	39 =	7943 ⇑	2
Random	39 ↓	4.1 =	6	29 ⇓	8123 ⇑	4
Stmt-coverage	32 ↑	4.1 ↑	2	–	–	–
Algorithms	**Header Files**			**Build System**		
	Bugs	Configs	Rank	Bugs	Configs	Rank
Pair-wise	39 =	936 ⇑	4	33 ↓	2.8 ↑	4
Three-wise	43 =	1218 ⇑	5	42 ↓	3.9 ↑	5
Four-wise	45 =	1639 ⇑	7	45 =	5.7 ↑	8
Five-wise	–	–	–	47 =	8.3 ↑	9
Six-wise	–	–	–	47 =	12 ↑	10
Most-enabled-disabled	27 =	1.4 =	1	26 ↓	1.4 ↑	2
One-enabled	31 ↑	890 ⇑	6	20 ↓	2.3 ↑	7
One-disabled	39 =	890 ⇑	3	39 =	2.3 ↑	3
Random	40 ↓	17.2 ⇑	2	41 =	4.2 ↑	6
Stmt-coverage	–	–	–	25 =	1.3 ↑	1

Some algorithms do not scale, indicated using dashes (–). We use ↑ and ↓ to represent small changes in the number of bugs and size of sample set, as compared to our first study. Furthermore, we use ⇑ and ⇓ to represent larger changes.

4.3 The Linear Sampling Algorithm

To select configurations systematically, we propose the *Linear Sampling Algorithm (LSA)*, which combines the following sampling algorithms:

- *One-disabled*: Abal et al. [16] suggested this algorithm based on 42 configuration-related bugs analyzed in the *Linux Kernel*. It deactivates one preprocessor macro at a time; it requires *n* configurations per file, where *n* is the number of preprocessor macros in each source file.
- *One-enabled*: the algorithm is similar to *one-disabled*, but *one-enabled* activates one preprocessor macro at a time. *One-enabled* also requires *n* configurations per file.
- *Most-enabled-disabled*: this algorithm consists of activating all preprocessor macros and then deactivating all macros, which require two configurations per source file.

The *LSA* sampling algorithm selects configurations linearly, that is, it selects $2n + 2$ configurations, in which *n* is the number of configuration options.

According to our comparison of sampling algorithms, *LSA* increases the number of configuration-related bugs detected without the need of selecting a large sample set. In Figure 4.9, we show how *LSA* selects configurations using an example source file with four preprocessor macros: *A*, *B*, *C*, and *D*. When considering constraints among preprocessor macros, *LSA* selects configurations slightly differently: For instance, assuming that *A* and *B* are mutually exclusive, *most-enabled-disabled* cannot activate all macros (i.e., *configuration 9* is invalid). In this case, *LSA* activates the highest number of macros possible using a SAT solver, that is, only three macros will be active (*A* or *B*, *C*, and *D*). The same situation occurs for *one-enabled* and *one-disabled*, when we need to consider constraints.

Configuration Options	A	B	C	D	
Configuration 1	✓	✗	✗	✗	One-enabled
Configuration 2	✗	✓	✗	✗	
Configuration 3	✗	✗	✓	✗	
Configuration 4	✗	✗	✗	✓	
Configuration 5	✗	✓	✓	✓	One-disabled
Configuration 6	✓	✗	✓	✓	
Configuration 7	✓	✓	✗	✓	
Configuration 8	✓	✓	✓	✗	
Configuration 9	✓	✓	✓	✓	Most-enabled-disabled
Configuration 10	✗	✗	✗	✗	

✓ Option is enabled ✗ Option is disabled

Figure 4.9: Selecting configurations systematically with *LSA*.

Notice that *LSA* ensures, in the absence of constraints, *pair-wise* coverage because it selects configurations that analyze all pairs of preprocessor macros. By considering macros *A* and *B* in Figure 4.9, we can see that there is a configuration where *A* and *B* are enabled (*configuration 7*); another configuration in which both *A* and *B* are disabled (*configuration 3*); and other configurations where only *A* or *B* is enabled (for example, *configurations 1* and *2*). The same situation occurs for preprocessor macros *A* and *C*, *A* and *D*, *B* and *C*, and macros *B* and *D*. *LSA* also ensures *3-way* combinatorial interaction testing (*three-wise* coverage) without considering constraints. We can use the same rationale to see that *LSA* covers all combinations of three preprocessor macros.

4.4 Research Study

In this section, we present the setup of an empirical study performed to better understand configuration-related bugs. To perform this empirical study, we instantiated the sampling-based strategy using the *LSA* sampling algorithm and *Cppcheck*.

4.4.1 Overall Study Design

In particular, we address the following research questions:

- **RQ7.** How frequent are configuration-related bugs when compared to bugs that appear in all configurations?

- **RQ8.** Do configuration-related bugs remain longer in the source code than bugs that appear in all configurations?
- **RQ9.** How do developers introduce configuration-related bugs?
- **RQ10.** Does undisciplined preprocessor usage influence developers to introduce configuration-related bugs?

Before answering research questions **RQ7** and **RQ8**, we confirmed each bug of both sets by searching for fixes in the corresponding software repositories (i.e., made by actual developers of the project) and by submitting patches to developers. We excluded from the analysis bugs that we could not confirm as real bugs. To answer **RQ7** and **RQ8**, we collected two sets of bugs: (1) a set with bugs that appear in all configurations; and (2) another set with configuration-related bugs that occur only in certain configurations.

To answer **RQ7**, we counted the number of bugs that appear in all configurations and the number of configuration-related bugs. Answering **RQ7** is important to identify how is the distribution and frequency of configuration-related bugs.

Regarding **RQ8**, we analyzed each bug of both sets to analyze the dates that developers introduce and fix the bug to measure the time in between. Then, we compared the numbers of days to fix bugs that appear in all configurations and to fix configuration-related bugs. This research question (**RQ8**) provides insights on how variability hinders the detection of configuration-related bugs. Our hypothesis is that configuration-related bugs remain in the source code longer than bugs that appear in all configurations because variability hinders the detection of configuration-related bugs.

In **RQ9**, we analyzed how developers introduce the configuration-related bugs we found. Then, we classified and quantified the different ways of introducing bugs. Notice that **RQ9** considers only the set of configuration-related bugs and might provide useful information for developers of variability-aware analysis tools [71].

In **RQ10**, we used the definition of Liebig at el. [67] of undisciplined preprocessor use, that is, conditional directives that do not align with the underlying syntactic structure of the source code (as discussed in Section 2.2.2). We counted the number of configuration-related bugs that appear in disciplined and undisciplined preprocessor usage. Again, notice that **RQ10** considers only the set of configuration-related bugs. Research question **RQ10** might

influence developers to avoid certain types of preprocessor usage to minimize configuration-related bugs in practice.

Subjects Selection

Overall, we analyzed 27 systems written in C ranging from 20 to 2126 preprocessor macros. These projects are from different domains, such as Web servers, text editors, databases, and games. We selected these projects guided by previous work [4; 67], which studied C projects that are statically configurable with the C preprocessor (i.e., projects that use preprocessor conditional directives). We present details of each project in Table 4.5, listing the project name, application domain, lines of code, number of files, number of preprocessor macros, number of developers, number of distinct code versions, and the number of configuration-related bugs detected in our empirical study.

Conduct and Instrumentation

For the purpose of our study, we selected the current stable release of each of our 27 projects. Furthermore, we considered previous versions of the code using the *Git* repositories of the projects. After selecting configurations for each source file, we used *Cppcheck* version 1.67 to detect bugs in each selected configuration on a per-file basis. We also used *Git* version 2.3.2 to get information about the repositories. We counted the lines of code and the number of files using the *Count Lines of Code (CLOC)* tool version 1.56. Finally, we used *Cppstats* 0.7 version to quantify the number of occurrences of undisciplined preprocessor usage.

4.4.2 Results and Discussion

Overall, we found 65 configuration-related bugs in 18 out of our 27 subject projects (67%), as we present in Table 4.5. We counted in our statistics only bugs that developers fixed or bugs for which we submitted patches that developers accepted. We found that developers had already fixed 53 bugs (81%) detected in our study, and they accepted our patches to fix 12 (19%) additional bugs.

We found different types of configuration-related bugs, as we present in Figure 4.10: 34 memory leaks (52%), 12 uninitialized variables (19%), 11 dereferences of null pointers (17%), 6 resource leaks (9%), and 2 bugs related to buffer overflows (3%).

Table 4.5: Overview of the subject projects and the total number of bugs.

Project	Domain	LOC	Macros	Versions	Optional (%)	Bugs
Apache	Web server	144 768	700	25 615	10	5
Bash	Language interpreter	44 824	1427	68	37	2
Bison	Parser generator	24 325	269	5423	13	1
Cherokee	Web server	63 109	452	5748	11	3
Clamav	Antivirus	107 548	1632	13 457	17	1
Dia	Diagramming software	28 074	307	5634	4	4
Expat	XML library	17 103	84	47	20	0
Flex	Lexical analyzer	16 501	130	1607	7	0
Fvwm	Window manager	102 301	301	5439	8	6
Gawk	GAWK interpreter	43 070	714	1345	27	1
Gnuchess	Chess player	9293	39	236	9	0
Gnuplot	Plotting tool	79 557	500	8024	26	3
Gzip	File compressor	5809	141	445	21	1
Irssi	IRC client	51 356	157	4130	3	5
Libpng	PNG library	44 828	327	2188	81	3
Libsoup	SOUP library	40 061	92	2005	1	0
Libssh	SSH library	28 015	115	2915	35	13
Libxml2	XML library	234 934	2126	4246	71	0
Lighttpd	Web server	38 847	215	1470	24	3
Lua	Language interpreter	14 503	145	83	3	1
M4	Macro expander	10 469	106	953	29	1
Mpsolve	Mathematical software	10 278	20	1434	2	0
Privoxy	Proxy server	29 021	158	63	36	0
Rcs	Revision control system	11 916	97	915	15	0
Sqlite	Database system	94 113	467	553	57	0
Sylpheed	E-mail client	83 528	286	2733	15	3
Vim	Text editor	288 654	942	5720	63	9
Total		**1 666 805**				**65**

Lines of code; number of source files; number of compile-time preprocessor macros; number of developers contributing to the project; number of versions analyzed; percentage of optional code; and number of configuration-related bugs detected.

We further determined the number of preprocessor macros involved in each configuration-related bug. The majority of bugs involves one preprocessor macro: 58 bugs (89% of the preprocessor-related bugs considered in our study). Furthermore, we found three bugs that involve 2 preprocessor macros; two bugs that relate to 3 macros; one bug that involves 5 preprocessor macros; and one bug that relates to 7 macros. Previous studies reported similar results [19; 16; 12; 39].

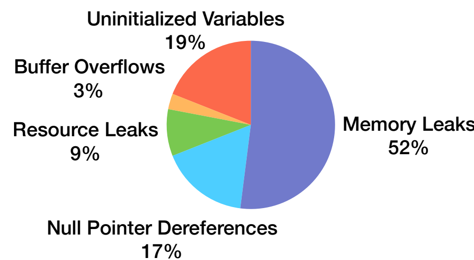


Figure 4.10: Types of configuration-related bugs.

Here, we present an example of configuration-related uninitialized variable in Figure 4.11 for illustration. We found this uninitialized variable in *Bash* and this bug occurs only when we disable macros `TRACE` and `REGISTER`, and enable macro `WATCH`. In this configuration, variable `ubytes` is not initialized at Line 5, but used at Line 15. Technically, the value of an uninitialized non-static, local variable is indeterminate in C, and accessing it leads to an undefined behavior.

Next, we answer the research questions. Then, we present the patches submitted to the projects in Section 4.4.2, and discuss the threats to validity in Section 4.4.3.

RQ7. How frequent are configuration-related bugs when compared to bugs that appear in all configurations?

To answer this research question, we counted the Lines of Code (LOC) of each subject system and measured the percentage of code related to preprocessor macros. In Table 4.5, we present this percentage and LOC for each subject system. Then, we computed the average of configuration-related code regarding all subject systems (i.e., we summed the percentage of configuration-related code of all systems and divided by the number of systems). As a result, on the average, 24% of the source code is related to configurability.

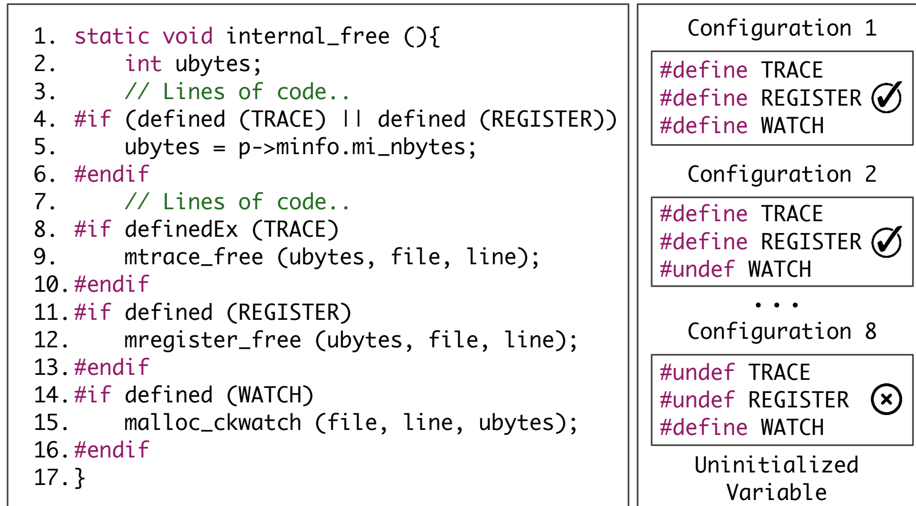


Figure 4.11: An example of configuration-related uninitialized variable in *Bash*.

We also counted the number of configuration-related bugs and the number of bugs that appear in all configurations. Overall, *Cppcheck* reported 368 warnings, but we classified 136 (37%) warnings as false positives, that is, bugs that appear in invalid configurations and warnings that are not real bugs. Regarding the other 232 (63%) warnings: 65 (28%) are configuration-related bugs and 167 (72%) appear in all configurations.

Our results show that bugs are similarly distributed across all source code. We found that 28% of the bugs are related to configurability and that they are distributed across 24% of the source code, which is related to preprocessor macros. Thus, our findings indicate that the frequency of configuration-related bugs is fairly similar to the frequency of bugs that appear in all configurations. By dividing the LOC related to configurability by the number of configuration-related bugs, we found $400\,033/65 = 0.16$ configuration-related bugs per thousand lines of code, while the frequency of bugs that appear in all configuration is: $1\,266\,771/167 = 0.13$ bugs per thousand lines of code.

SUMMARY

Developers face configuration-related bugs in practice as frequent as they face bugs that appear in all configurations. The frequency of bugs detectable by Cppcheck is 0.16 bugs per thousand lines of code for configuration-related bugs, and 0.13 bugs per thousand lines for bugs that appear in all configurations.

RQ8. Do configuration-related bugs remain longer in the source code than bugs that appear in all configurations?

To detect configuration-related bugs, developers need to analyze multiple configurations. They find a specific configuration-related bug only when checking a configuration, in which such bug occurs. Thus, the variability of configurable systems can hinder the detection of configuration-related bugs. In our study, we found that the time to fix bugs varies from days to years. However, we also found a number of bugs appearing in all configurations that developers took years to fix. Figure 4.12 shows the times to fix bugs that appear in all valid configurations, while Figure 4.13 presents the times to fix configuration-related bugs. We can see that developers usually need a long time to fix configuration-related bugs and bugs that appear in all configurations. We did not consider the configuration-related bugs that developers fixed using our patches, because this could influence the results. In addition, we considered only bugs that we know exactly when developers introduce and fix the bug. Overall, we considered 49 configuration-related bugs and 110 bugs that appear in all configurations.

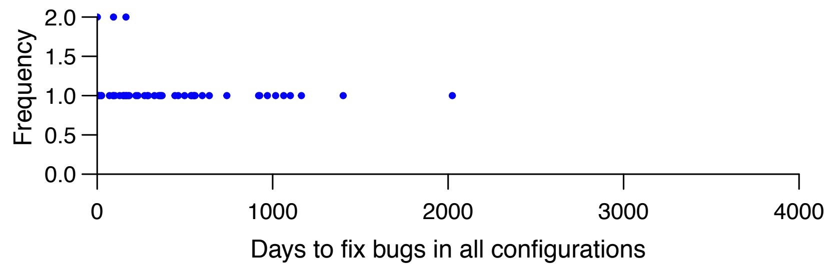


Figure 4.12: Number of days developers take to fix bugs that appear in all configurations.

To compare the time to fix bugs, we calculated the averages of the number of days to fix bugs: 397 days to fix bugs that appear in all configurations; and 1143 for configuration-related bugs. We ran an unpaired *t-test* and checked the null hypothesis that there is no difference between the averages. We obtained a *p-value* $= 1.27 \times 10^{-4}$, so we rejected the null hypothesis. So, our findings provide statistical evidence that configuration-related bugs remain longer in the code than bugs that appear in all configurations, considering a 95% confidence interval.

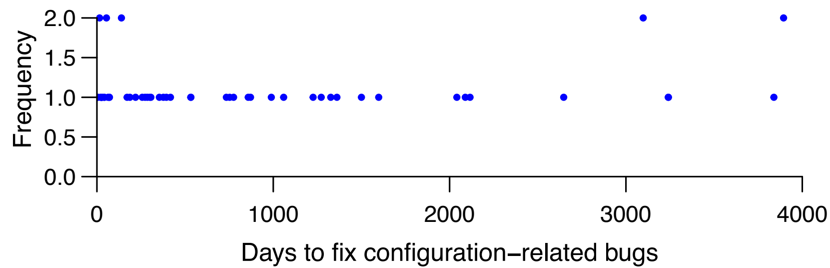


Figure 4.13: Number of days developers take to fix configuration-related bugs.

SUMMARY

Configuration-related bugs remain almost three times longer in the source code, on the average, than bugs that appear in all configurations.

RQ9. How do developers introduce configuration-related bugs?

We investigated how developers introduce the configuration-related bugs found in our study. Our goal is to identify whether developers introduce more bugs when implementing new functionalities or fixing other bugs in the source code. Our analysis reveal that developers introduce configuration-related memory and resource leaks, dereferences of null pointers, and uninitialized variables when modifying code (51%) and introducing new functionalities (49%), such as a new source file or function. We present the detailed results regarding bugs in the following order: memory leaks, resource leaks, uninitialized variables, dereferences null pointers, and buffer overflows.

We found developers introducing configuration-related memory leaks in seven distinct ways: **(I)** and **(II)**, introducing a new function that allocates memory without freeing it; **(III)** and **(IV)**, modifying an existing function by allocating memory without freeing it; **(V)**, optionally calling a function that returns an allocated memory; **(VI)**, conditionally passing a pointer to a function that allocates memory to this pointer; and **(VII)**, introducing an error handler that terminates the execution of a function without freeing the allocated memory. We show these seven cases in Figure 4.14. In the first two cases (**I** and **II**), developers introduce new functionalities. In the other cases (**III–VII**), they introduce memory leaks by modifying existing code. Overall, we found configuration-related memory leaks in the following projects, as presented in Table 4.6.

Table 4.6: Occurrences of configuration-related memory leaks.

Case	**Occurrences**
I	*Apache* (1), *Cherokee* (2), *Irssi* (1), *Libpng* (1), *Libssh* (3), and *Vim* (4).
II	*Fvwm* (3), *Gawk* (1), *Irssi* (1), and *Sylpheed* (1).
III	*Fvwm* (1), *Gnuplot* (1), *Irssi* (1), and *Vim* (2).
IV	*Libpng* (1), *Libssh* (1), *Lighttpd* (1), and *M4* (1).
V	*Libssh* (1), and *Vim* (1).
VI	*Dia* (1).
VII	*Libpng* (1), *Libssh* (1), *Lighttpd* (1), and *Vim* (1).

I

```
+ #ifdef A
+ void f1(){
+   …
+   int *p1;
+   p1 = malloc(…);
+   …
+ }
+ #endif
```

II

```
+ void f2(){
+   …
+ #ifdef A
+   int *p2;
+   p2 = malloc(…);
+ #endif
+   …
+ }
```

III

```
  void f3(){
    …
+ #ifdef A
+   int *p3;
+ #endif
    …
+ #ifdef A
+   p3 = malloc(…);
+ #endif
    …
  }
```

IV

```
  #ifdef A
  void f4(){
    …
+   int *p4;
+   p4 = malloc(…);

    …
  }
  #endif
```

V

```
  int f5(){
    int *p5;
    p5 = malloc(…);
    …
    return *p5;
  }

  int f6(){
    …
+ #ifdef A
+   int p6 = f5();
+ #endif
  }
```

VI

```
  void f7(int **p7){
    *p7 = malloc(…);
    …
  }

  void f8(){
    …
+ #ifdef A
+   int *p8;
+   f7(&p8);
+ #endif
    …
  }
```

VII

```
  type f9(){
    int *p9;
    p9 = malloc(…);
    …
+ #ifdef A
+   if (fail)
+     return NULL;
+ #endif
    free(p9);
    …
  }
```

`+` Including line

Figure 4.14: Introducing configuration-related memory leaks.

Developers introduce configuration-related resource leaks in four ways: **(I)** altering a function by conditionally opening a file without closing it after use; **(II)** altering a function by conditionally adding an error handler that does not close an opening file; **(III)** and **(IV)**, adding a new function that conditionally opens a file without closing it after use. Cases **I–II** modify existing code and cases **III–IV** introduce new functionalities. Figure 4.15 shows these four cases. We found resource leaks in the six projects, as presented in Table 4.7.

I
```
void f1(){
  …
+ #ifdef A
+   FILE *p1;
+   p1 = fopen(…);
+ #endif
  …
}
```
II
```
int f3(){
  FILE *p3;
  p3 = fopen(…);
  …
+ #ifdef A
+   if (fail)
+     return NULL;
+ #endif
  close(p3);
}
```
III
```
+ void f2(){
+   …
+ #ifdef A
+   FILE *p2;
+   p2 = fopen(…);
+ #endif
+   …
+ }
```
IV
```
+ #ifdef A
+ int f4(){
+   FILE *p4;
+   p4 = fopen(…);
+   …
+   if (fail)
+     return NULL;
+   close(p4);
+ }
+ #endif
```
\+ Including line

Figure 4.15: Introducing configuration-related resource leaks.

Table 4.7: Occurrences of configuration-related resource leaks.

Case	Occurrences
I	*Lua* (1).
II	*Libssh* (1), and *Lighttpd* (1).
III	*Clamav* (1).
IV	*Libssh* (1), and *Sylpheed* (1).

I
```
+ #ifdef A
+ void f1(){
+   …
+   int p1;
+   f2(p1);
+   …
+ }
+ #endif
```
II
```
+ void f5(){
+   int p3;
+   …
+ #ifdef A
+   …
+   p3 = 10;
+ #endif
+   f6 (p3);
+   …
+ }
```
III
```
void f3(){
  …
+ #ifdef A
+   int p2;
+   f4(p2);
+ #endif
  …
}
```
IV
```
void f7(){
  int p4;
  …
  #ifdef A
    p4 = 10;
  #endif
+ #ifdef B
+   f8(p4);
+ #endif
}
```
\+ Including line

Figure 4.16: Introducing configuration-related uninitialized variables.

Developers introduce uninitialized variables in four ways: **(I)** and **(II)**, introducing a new function that declares a variable and use it without initialization; **(III)**, modifying an existing function by declaring a variable and using this variable without initializing it; and **(IV)**, using a conditionally defined variable in a new code with different condition. Figure 4.16 presents these cases. We found configuration-related uninitialized variables in the following projects, as presented in Table 4.8.

Table 4.8: Occurrences of configuration-related uninitialized variables.

Case	Occurrences
I	*Apache* (1), *Cherokee* (1), *Dia* (2), *Libssh* (1), and *Sylpheed* (1).
II	*Bash* (1).
III	*Apache* (1), *Fvwm* (1), and *Gzip* (1).
IV	*Bash* (1).

We found developers introducing null pointer dereferences in three ways: **(I)**, adding a new variable, allocating memory and using it without checking whether the allocation succeeds; **(II)**, introducing a new function that receives a pointer and uses it without checking whether it is `NULL`; and **(III)**, using memory allocated previously without checking if the allocation succeeds. Figure 4.17 presents all cases. We found the following null dereferences, as presented in Table 4.9.

Table 4.9: Occurrences of configuration-related null pointer dereferences.

Case	Occurrences
I	*Gnuplot* (1), *Irssi* (1), and *Libssh* (3).
II	*Dia* (1), *Irssi* (1), and *Libssh* (1).
III	*Apache* (1), *Fvwm* (1), and *Gnuplot* (1).

I

```
  #ifdef A
  void f1(){
   …
+  int *p1;
+  p1 = malloc(…);
+  f2(p1);
   …
  }
  #endif
```

II

```
+ #ifdef A
+ void f2(int *p2){
+  …
+  f3(p2);
+  …
+ }
+ #endif
```

III

```
 void f4(){
  …
  int *p2;
  p2 = malloc(…);
+ #ifdef A
+  f3(p2);
+ #endif
  …
 }
```

+ Including line

Figure 4.17: Introducing configuration-related null pointer dereferences.

Developers introduce buffer overflows in two ways: **(I)**, introducing a buffer and trying to access a position outside its boundaries; and **(II)**, introducing a function that receives a buffer as parameter and use it without checking its boundaries. We present these two cases

in Figure 4.18. We found only two occurrences of configuration-related buffer overflows, one follows case **I**: *bison* (1); and another case **II**: *Vim* (1).

```
I
  #ifdef A
  void f1(){
    …
+   char *p1 [2];
+   int p2 = 3;
+   f2(p1[p2]);
    …
  }
  #endif

II
+ #ifdef A
+ void f3(char *p2){
+   …
+   f4(p2[3]);
+   …
+ }
+ #endif

+ Including line
```

Figure 4.18: Introducing configuration-related buffer overflows.

SUMMARY

Developers introduce configuration-related bugs when modifying existing code (51%) and also introducing new functionalities (49%).

RQ10. Does undisciplined preprocessor usage influence developers to introduce configuration-related bugs?

The C preprocessor is flexible enough to allow developers to embrace any code fragment with preprocessor conditional directives, even a single token such as an opening bracket. This way, developers can introduce undisciplined directives that do not align with the underlying syntactic structure of the source code [67; 17]. Undisciplined preprocessor use may influence the code quality negatively [3; 13; 29], and might ease the introduction of configuration-related bugs in practice, as discussed in Section 2.2.2.

By considering the 27 projects of our study: 15% of the directives are undisciplined. However, only 8% of the optional lines of code are surrounded by these directives. By analyzing the 65 configuration-related bugs considered in our study, we found that 6 configuration-related bugs (9%) occur in undisciplined preprocessor directives. In this way, 9% of the configuration-related bugs are distributed across 8% of undisciplined directives, which does not support our hypothesis that configuration-related bugs are more frequent in undisciplined directives. Overall, our results do not suggest that undisciplined directives usage may influence the introduction of configuration-related memory and resource leaks,

dereferences of null pointers, and uninitialized variables, as such bugs are similarly distributed across disciplined and undisciplined preprocessor directives. This result is not in line with configuration-related syntax errors, which we will present in Chapter 5.

SUMMARY

The results refuted the hypothesis that undisciplined directives influence developers to introduce configuration-related memory and resource leaks, dereferences of null pointers, and uninitialized variables.

Submitting Patches to Fix the Bugs

We submitted 20 patches—for each bug not already fixed by developers—to 11 subject projects: *Apache* (4), *Bash* (2), *Clamav* (1), *Dia* (1), *Gawk* (2), *Gnuplot* (2), *Gzip* (1), *Lighttpd* (2), *Libxml2* (1), *Sqlite* (3), and *Sylpheed* (1).

We consider that developers accept a patch when they mention that it is a bug, or keep the patch open after updating some patch information, such as its priority. Conversely, we consider that developers reject the patch when they mention it is not a bug, or update this information on the patch. Overall, developers accepted 12 (60%) out of 20 patches we submitted to the C projects. We present information about the patches in Table 4.10, listing the name of the project, file name with the bug, the type of bug, and the patch status.

Developers already fixed 9 bugs out of 12 patches accepted. Regarding one patch we submitted to *Dia* and two patches to *Bash*, developers accepted them, but they have not fixed the bugs yet. Regarding the 8 rejected patches, the *Apache* developers rejected four patches: two bugs that arise in invalid configurations and two false positives. The *Sqlite* Developers also rejected two patches as they were false positives. Moreover, the *Gnuplot* developers rejected two patches arguing that resource leaks in the main function are not problematic, as the operating system closes all resources when killing the process, but developers consider it as bad style: "*There are some fclose() statements missing at the end of main(). While this is certainly not good style, it is also not a problem since the C runtime or the operating system will close any open files on leaving main().*"

Table 4.10: Patches submitted to subject systems.

Project	File	Problem	Status
Apache	ssl_util.c	Null dereference	Rejected
Apache	mpm_winnt.c	Memory leak	Rejected
Apache	ap_regkey.c	Uninitialized variable	Rejected
Apache	ap_regkey.c	Uninitialized variable	Rejected
Bash	input.c	Uninitialized variable	Accepted
Bash	malloc.c	Uninitialized variable	Accepted
Clamav	clamconf.c	Resource leak	Fixed
Dia	test-boundingbox.c	Uninit variable	Accepted
Gawk	popen.c	Memory leak	Fixed
Gawk	regcomp.c	Memory leak	Fixed
Gnuplot	doc2html.c	Resource leak	Rejected
Gnuplot	doc2html.c	Resource leak	Rejected
Gzip	bits.c	Uninit variable	Fixed
Lighttpd	mod_dirlisting.c	Memory leak	Fixed
Lighttpd	mod_dirlisting.c	Resource leak	Fixed
Libxml2	catalog.c	Resource leak	Fixed
Sqlite	test_quota.c	Uninitialized variable	Fixed
Sqlite	os_win.c	Uninitialized variable	Rejected
Sqlite	test_intarray.c	Memory leak	Rejected
Sylpheed	jpilot.c	Resource leak	Fixed

4.4.3 Threats to Validity

In this section, we discuss a number of threats to validity that are crucial for our empirical study.

Construct validity

The issue of whether the configuration-related bugs detected are real bugs or false positives threatens construct validity. We addressed this threat by getting feedback from developers to confirm each bug reported in our statistics. For each bugs, we checked it in two ways: (1) finding a fix in newer versions of the code; and (2) submitting patches to the projects. Developers accepted 12 out of the 20 configuration-related bugs reported, and we confirmed 53 bugs that developers fixed in newer versions of the source code.

Internal validity

We used *Cppcheck* to detect bugs, thus limiting our study to bugs that this tool can detect, such as memory leaks, uninitialized variables, and null pointer dereferences. *Cppcheck* may also report false positives (more than 30% according to our study). However, we verified false positives by asking the actual developers and did not count them in our statistics.

Our strategy analyzes one file at a time, which does not find bugs that span multiple files. So, we may miss some bugs (false negatives). Furthermore, our strategy does not check all possible configurations, as we used sampling which checks only a subset of configurations. So, we might miss some bugs in configurations that we do not analyze. To minimize this threat, we used a sampling algorithm that we defined based on a comparative study of 10 sampling algorithms, aiming at maximizing the number of detected bugs [43].

Our strategy to find configuration-related bugs in project repositories considered only updated and added files, from the second to the last commit, as described in Section 4.1. However, this approach may lead to false negatives. For instance, developers may update a file A, which leads to bugs in a different file B. In our strategy, because only A has been modified, we only analyze A. However, later, if developers modify B, our strategy potentially catches the bugs.

During our study, we analyzed how developers introduce configuration-related bugs, such as introducing new functionalities or changing existing code. However, we performed this analysis manually, which may introduce errors. For instance, a developers might remove code from a source file and introduce the code removed in a later commit. To minimize this threat, we used the Source Tree[5] tool that highlights removed and added code to make the analysis less error prone.

External validity

We analyzed 27 projects of different domains, sizes, numbers of preprocessor macros, and numbers of developers. We selected mature C projects used in industrial practice, but we also selected some younger projects with a few developers to consider a broader range of project's characteristics. The corresponding communities exist for years and are very active. This way, we alleviated threats related to external validity.

[5] https://www.sourcetreeapp.com/

Chapter 5

A Variability-Aware Strategy to Detect Configuration-Related Bugs

In this chapter, we present our strategy to detect configuration-related bugs based on variability-aware analysis. The sampling-based strategy presented in Chapter 4 is incomplete, that is, it checks only a subset of configurations. So, variability-aware analysis is a complimentary strategy to detect additional configuration-related bugs. The strategy uses a variability-aware parser to create abstract syntax trees enhanced with variability information and applies a number of bug checkers to detect different types of configuration-related bugs, including syntax errors, undeclared variables, and unused functions.

In Section 5.1, we present the variability-aware strategy, explaining two simplifications to make the analysis scalable in detail: (1) the use of stubs to eliminate external dependencies in Section 5.1.1, and (2) platform-specific headers to reduce configurations in Section 5.1.2. Last, Section 5.2 discussed an empirical study we performed to evaluate the proposed variability-aware strategy using a corpus of 40 C real-world systems.

5.1 The Variability-Aware Strategy

Variability-aware tools, such as *TypeChef* [17] and *SuperC* [38], analyze complete configuration spaces, considering file inclusion (`#include` directives) and macro expansions (`#define` directives). Instead of considering macro definitions, macro expansion, and file inclusion intertwined, these tools perform partial preprocessing, which preprocesses file inclusion and macro expansion, but retains variability information for further analysis [70].

This way, variability-aware tools generate abstract syntax trees enhanced with all variability information.

Existing variability-aware tools consider file inclusion of many supported platforms and application scenarios. For example, *Libssh* uses two alternative cryptography routines: *libcrypto* and *libgcrypt*. When using variability-aware analysis, we need to consider the `#include` directives of both libraries. Likewise, for other optional functionalities. It causes large amounts of I/O operations during compilation, which slows down the compilation process and needs a time-consuming setup. An average file in the *Linux Kernel*, for example, includes over 300 header files [17]. Furthermore, incorporating header files increases the number of preprocessor macros per file significantly. The files of the *Linux Kernel* contain, on average, 3 distinct macros when ignoring variability from header files, headers add another 238 distinct preprocessor macros, on average. This way, the time-consuming setup of variability-aware tools hinders the analysis of several projects. Next, we present two simplifications of our strategy to make the analysis scalable.

5.1.1 Stubs

In this section, we present our strategy to detect configuration-related bugs using stubs, we refer to Figure 5.1, and detail its four steps in what follows.

The goal of the first step is to enable us to analyze several projects. In this step, we exclude all external libraries from the project by eliminating `#include` directives. We still consider the header files of the projects, but exclude the external ones. For example, the C file used as input in Figure 5.1 includes the `stdio.h` library, which is not part of the project code. Other external dependencies may be specific for an operating system, e.g., we cannot include the external *windows.h* in *Linux*. Because finding and downloading the correct library version is a manual and time consuming task, considering these external libraries of all supported platforms would hinders our analysis. In this way, we only focus on the project code.

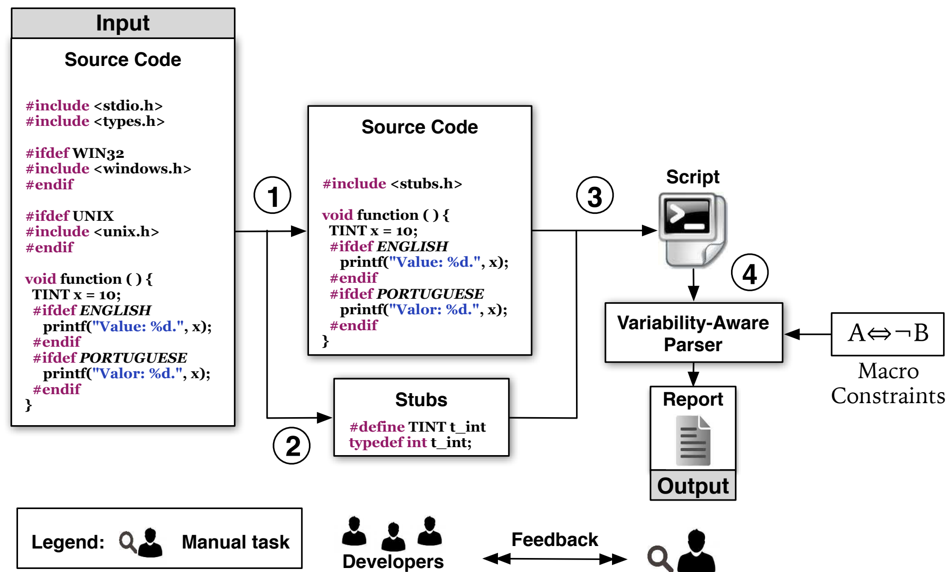


Figure 5.1: Strategy to detect bugs using stubs.

By excluding `#include` directives, Step 1 may leave some types and macros undefined. We generate stubs using C/C++ Development Tooling (CDT) with the default configuration to replace the original types and macros. Then, we create a `stubs.h` file to contain these stubs (Step 2). We use the CDT parser to generate an abstract syntax tree for each source code file. Then, we navigate through the abstract syntax tree, get the types and macros that CDT identifies, and add them to the `stubs.h` file. We include this file into the project source code and now the variability-aware parser is able to parse the source code.

Step 3 generates a shell script that calls the variability-aware parser for each source file. We built an Eclipse *plug-in* that automates Steps 1-3 (see Chapter 7). Finally, we run the script our strategy generates in Step 4. When the variability-aware parser reports a bug, we perform a manual check to verify whether the bug is a configuration-related bug, that is, a bug that appears only in some configurations of the source code. After fixing the bug, we may continue to analyze the project source file, i.e., depending on the bug, we add a missing bracket, or remove an additional comma, and so on. This way, the variability-aware parser continues to analyze the file. In case of a configuration-related bug, we create a bug report with information like the problematic configuration and code snippet with the bug. In this step, we get feedback from the actual projects developers to confirm the bugs.

When generating the abstract syntax trees using the variability-aware parser, we detect configuration-related syntax errors. During this step, our strategy may receive any known constraints to eliminate invalid configurations (for example, preprocessor macros `A` and `B` are mutually exclusive). We pass this information to the variability-aware tool, which then ignores the invalid configurations. Next, by using the abstract syntax trees, we are able to implement different bug checkers. Notice, though, that we cannot verify type errors as we substituted the external libraries, which may define types. Furthermore, it is important to mention that our strategy to detect bugs using stubs may generate false positives and negatives. For example, CDT may not identify all types and macros. Additionally, external libraries defining macros may influence the program family code. Section 5.2.3 discusses these topics in detail.

5.1.2 Platform-Specific Headers

Our second simplification preprocesses header files to generate platform-specific headers. This way, our strategy parses the system source code (C files only) without preprocessing and generates an abstract syntax tree enhanced with variability information for each source file. Figure 5.2 illustrates the three steps of our strategy, detailed in what follows.

The goal of *Step 1* is to enable us to analyze several C systems. A common difficulty in performing variability-aware analysis is that many preprocessor macros are related to platform-specific definitions and libraries. Hence, our strategy preprocesses the included header files and generates platform-specific versions of these files. Despite focusing only on one platform at a time, the strategy enables us to analyze several software systems in such a platform. To generate platform-specific headers, we remove the conditional directives (such as `#ifdef` and `#endif`) of the header files, according to the characteristics of a specific platform. For instance, Figure 5.3 presents how we generate platform-specific headers for the *Linux* platform using *Gcc*. We use the argument *-U* to disable macro `WIN32` and the argument *-D* to enable macro `LINUX`. After preprocessing the source code, the C preprocessor removes the conditional codes associated with the `WIN32` preprocessor macro, and resolves the includes. Thus, our strategy considers only one configuration of each header file. To instantiate our strategy for different platforms, one needs to generate platform-specific header files for each different target platform. However, notice that we do not preprocess the C files. For those files, we consider the entire configuration space, as we explain in what follows.

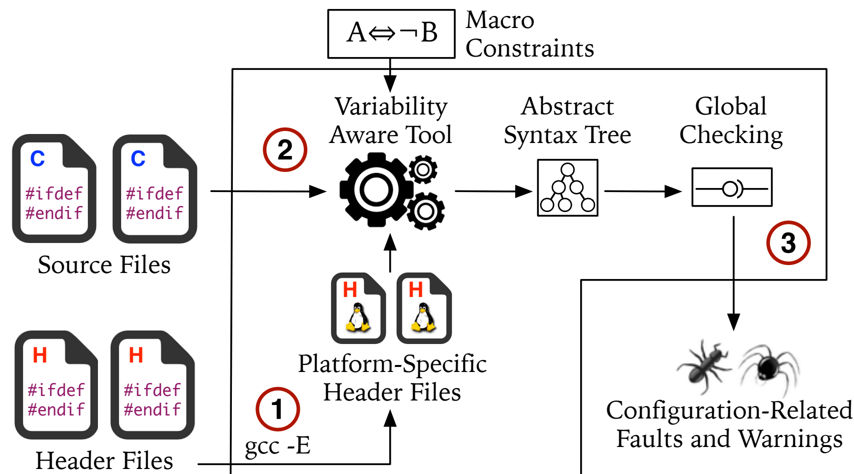


Figure 5.2: Strategy to detect bugs using platform-specific headers.

In *Step 2*, we use a variability-aware tool to parse the source code (C files) and generate an abstract syntax tree for each source file. When parsing each source file, the tool uses the platform-specific header files generated in the first step. Since we do not preprocess the source files, they still contain preprocessor conditional directives. Therefore, the resulting abstract syntax tree has choice nodes to represent the optional and alternative code blocks. During this step, our strategy may receive constraints to eliminate invalid configurations. We pass this information to the parser, which then ignores the invalid configurations.

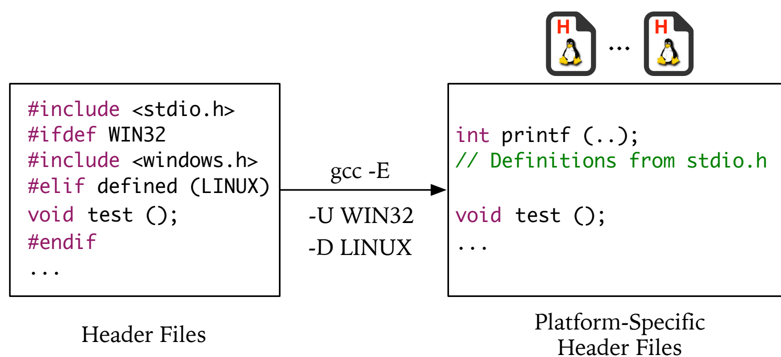


Figure 5.3: Generating platform-specific headers for *Linux*.

Step 3 uses the abstract syntax trees of the source files to detect the bugs. Notice that we consider the abstract syntax trees of all source files, which allow us to detect configuration-

related bugs that span multiple files. At this point, we have the following variability-aware checkers implemented: undeclared variables, unused variables, undeclared functions, and unused functions. Nonetheless, we can extend our infrastructure to add other checkers, such as checking for return types, and fields in structure declarations.

To detect configuration-related bugs in repositories, we extend our strategy to detect bugs to analyze *Git* repositories. For each set of files of a given commit in the repository, we apply our strategies to detect bugs. In the first commit of a given program family, we analyze all files. In the following commits, we only consider the updated and added files. In this way, we avoid the overhead of analyzing files that have not changed, as explained in Chapter 4.

5.2 Research Study

In this section, we present the settings of an empirical study performed to understand configuration-related bugs and to evaluate our variability-aware strategy. To perform the study, we instantiate our strategy to detect bugs using the well-known *Gcc* compiler, *TypeChef* [17], a variability-aware parser widely used in previous studies [47; 92; 113; 12], and the *Linux* operating system to generate platform-specific header files. We choose *Linux* because it provides simple and effective packaging tools to identify and install the software system dependencies. In this study, we considered the following types of configuration-related bugs: syntax errors, undeclared variables, unused variables, undeclared functions, and unused functions.

5.2.1 Overall Study Design

In particular, this empirical study addresses the following research questions:

- **RQ1.** What are the frequencies of configuration-related bugs and warnings?
- **RQ2.** Do configuration-related bugs involve multiple macros?
- **RQ3.** How do developers introduce configuration-related bugs?
- **RQ4.** How long do configuration-related bugs remain in the code?

Before answering the research questions, we consider feedback from the actual system developers to confirm each configuration-related bug. So, all numbers we report here do not include false positives. We also receive feedback regarding macro constraints and we used this information to avoid checking invalid configurations. To answer **RQ1**, we parse the code to detect syntax errors, execute four bug checkers (i.e., undeclared function, unused function, undeclared variables, and unused variables), and count their frequencies. Regarding **RQ2**, we count the number of preprocessor macros involved in each configuration-related bug. In **RQ3**, we analyze each bug to verify how developers introduced them by using the source file history in the software repository. Regarding **RQ4**, we analyze the dates that developers introduced and fixed the bugs to measure the time in-between.

Subjects Selection

We analyzed 40 subject systems written in C ranging from 2681 to 1 536 979 lines of code. These systems are from different domains, such as revision control systems, programming languages, and games. Furthermore, we considered mature systems with many developers as well as small systems with few developers. We selected these subject systems inspired by previous work [3; 30; 4], which performed studies with the C preprocessor. We present the details of each subject system in Table 5.1.

Instrumentation

We used the strategy presented in this chapter to investigate configuration-related bugs. We checked all systems by using stubs, and 15 systems by checking one platform at a time (see column *"Platform"* in Table 5.1). For some subject systems with *Git* software repository available, we also considered the commits history of the source files, as presented in column *"Git"* of Table 5.1. We used TypeChef version 0.3.5 to parse all possible configurations, CDT version 8.1.2 to create the stubs, and *Gcc* version 4.2.1 to generate platform-specific headers. Furthermore, to automatize our strategy, we used Eclipse Classic 4.2.2 to implement and run a *plug-in* to analyze the source code of the systems. We also counted the number of lines of code and the number of files of each system using the Count Lines of Code tool version 1.56, which eliminates blank lines and comments. Finally, we used *Git* version 1.7.12.4 to identify changes in files and get information about project repositories.

Operation

As a first part of our analysis, we executed our strategy using stubs to find configuration-related bugs in all 40 C systems we considered in this study. Next, we performed an analysis of 15 projects based on one platform at a time. Then, we investigated configuration-related bugs in projects history using the *Git* repositories of the subject systems. During the analysis of the repositories, we considered only the trunk, i.e., we do not analyze the individual branches. Next, we interpret and discuss the results of this empirical study to investigate configuration-related bugs.

5.2.2 Results and Discussion

In this section, we answer the research questions, discuss the patches we submitted, and present the threats to validity. All data used in this study are available on our Websites.[1]

RQ1: What are the frequencies of configuration-related bugs and warnings?

By analyzing the 40 subject systems, we detected 24 configuration-related syntax errors. For instance, Figure 5.4 depicts a configuration-related syntax error that we found in the *Vim* project. In this example, if we define macro `WIN32`, an error arises, as there is a missing logical operator at Line 4. Notice that it is a syntax error that any traditional compiler, such as *Gcc*, detect when compiling the source code. However, the variability hinders the detection of even simple configuration-related syntax errors, as they appear only when we compile specific configurations of the source code. Because of variability, more than 74% of developers believe that configuration-related bugs are more difficult to detect than bugs that appear in all configurations [29].

In 15 subject systems, which we analyzed using the one platform at a time approach, we found 14 undeclared functions; 7 unused functions; 2 undeclared variables; and 23 unused variables. Overall, we detected 39 configuration-related bugs of these four types. Figure 5.5 presents an example of undeclared variable. This code excerpt is part of the *Libpng* project, and it fails to compile when we enable `SPLT` and disable `POINTER`. In this configuration,

[1] `http://www.dsc.ufcg.edu.br/~spg/gpce2013/` and `http://www.dsc.ufcg.edu.br/~spg/gpce2015`.

Table 5.1: Subject characterization and number of bugs

Family	**Application Domain**	**LOC**	**Platform**	**Git**	**Bugs/Warnings**
Apache	web server	144 768		X	3
Atlantis	operating system	2681			
Bash	command language interpreter	44 824	X	X	24
Bc	calculator	5177	X		
Berkeley DB	database system	185 111			
Bison	parser generator	24 325			
Cherokee	web server	63 109			
Clamav	antivirus	107 548			
Cvs	version control system	76 125			1
Dia	diagramming software	28 074		X	2
Expat	XML library	17 103	X	X	
Flex	lexical analyzer	16 501	X	X	
Fvwm	windows manager	102 301			
Gawk	GAWK interpreter	43 070			
Ghostscript	postscript interpreter	1 536 979			
Gnuchess	chess player	9293	X	X	1
Gnuplot	plotting tool	79 557		X	5
Gzip	file compressor	5809	X	X	3
Irssi	IRC client	51 356			
Kin DB	database system	64 120			
Libdsmcc	DVB library	5453	X		
Libpng	PNG library	44 828	X	X	12
Libsoup	SOUP library	40 061	X	X	
Libssh	SSH library	28 015	X	X	4
Libxml2	XML library	234 934		X	2
Lighttpd	web server	38 847			
Lua	programming language	14 503	X	X	2
Lynx	web browser	80 334			
M4	macro expander	10 469	X	X	1
Mpsolve	mathematical software	10 278			
Mptris	game	4988	X		
Prc-tools	C/C++ library for palm OS	14 371			
Privoxy	proxy server	29 021	X		1
Sendmail	mail transfer agent	91 288			
Sqlite	database system	94 113			
Sylpheed	e-mail client	83 528			
Rcs	revision control system	11 916	X	X	
Vim	text editor	288 654			4
Xfig	vector graphics editor	70 493		X	1
Xterm	terminal emulator	50 830			2
Total		**3 860 078**			**68**

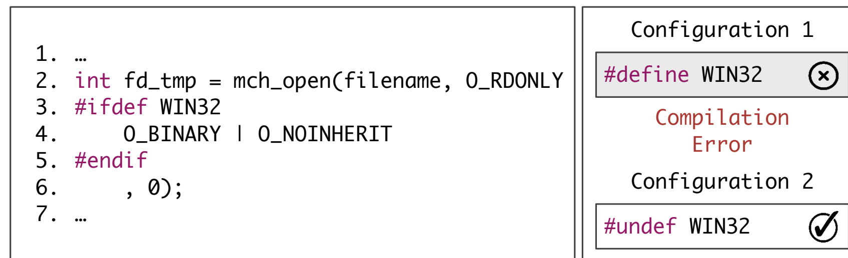


Figure 5.4: Code snippet of *Vim* with a syntax error.

developers declare variable `p` at Line 5 only when `POINTER` is enabled. The problem is that they use this variable at Lines 9 and 10, in which preprocessor macro `POINTER` is disabled, causing a compilation error.

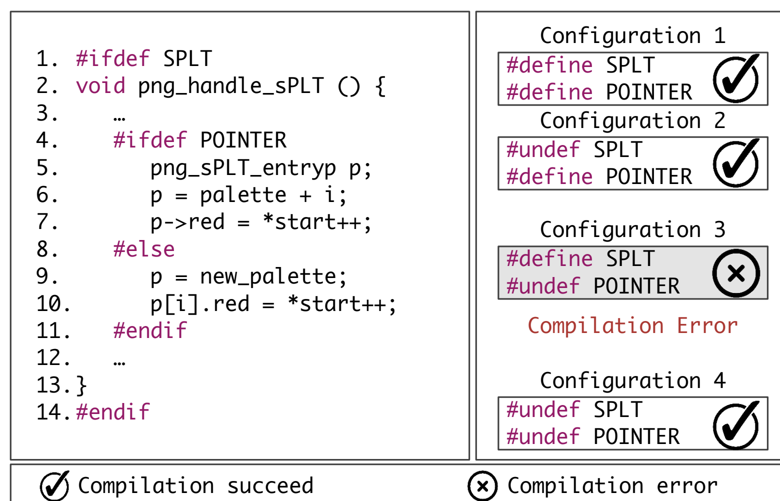


Figure 5.5: An undeclared variable in *Libpng*.

We also found unused variables and functions. Traditional C compilers raise warnings like unused variables and functions when developers set specific command line parameters. Still, we are able to find several unused variables and functions related to configurability. As these warnings do not cause compilation errors, developers might neglect them, even in mandatory code. Figure 5.6 presents a code excerpt with an unused variable in *Libssh*. In this code excerpt, variable `strong` is not used when we disable `LIBCRYPTO` and enable `LIBCRYPT`. The warning disappears when the opposite configuration selection happens. Although unused variable is a simple warning, some developers still care about them, by raising bug reports and providing patches to fix them. Indeed, we found bug reports and

patches to fix unused variables and functions, such as the one to fix the *Libssh* warning, presented in Figure 5.6.

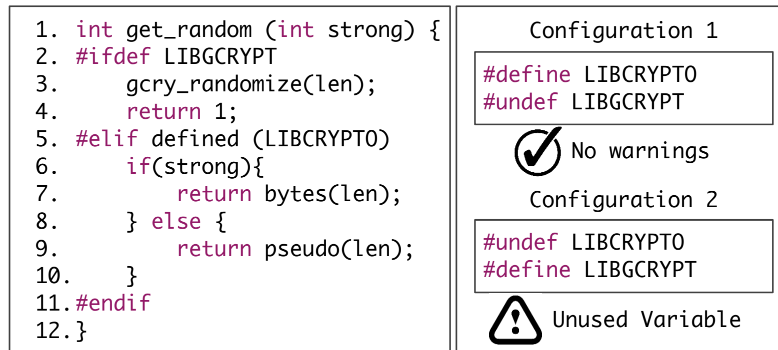


Figure 5.6: An unused variable in *Libssh*.

SUMMARY

We found configuration-related bugs and warnings of all types considered in our empirical study. The most frequent type of bug and warnings is unused variables, followed by undeclared functions, unused functions, syntax errors, and undeclared variables respectively.

RQ2: Do configuration-related bugs involve multiple macros?

We found that the majority of configuration-related bugs (more than 89%) involve two or less preprocessor macros. The number of preprocessor macros involved in a configuration-related bug is the number of macros that one needs to enable or disable to find a specific bug. Table 5.2 details the number of preprocessor macros involved in the bugs. For example, we found 16 bugs involving only one preprocessor macro. We also found 18 bugs depending on two macros, 3 bugs involving three preprocessor macros, and only 2 bugs when setting four or more preprocessor macros.

Studies that detected configuration-related bugs by analyzing software repositories and by using sampling analysis report similar results [16; 19], as discussed in Chapter 4. Because we use a different technique, i.e., variability-aware analysis, our empirical study provides more evidence that configuration-related bugs involving more than two preprocessor macros

are not common in C open source systems. Our findings also support the effectiveness of sampling algorithms, as the majority of the bugs do not involve high numbers of preprocessor macros.

Table 5.2: Preprocessor macros involved in bugs.

Some preprocessor macros enabled	25
a	20
$a \wedge b$	5
Some preprocessor macros disabled	22
$!a$	14
$!a \wedge !b$	7
$!a \wedge !b \wedge !c$	1
Some options enabled and some disabled	16
$a \wedge !b$	9
$a \vee !b$	1
$a \wedge !b \wedge !c$	4
$a \wedge !b \wedge !c \wedge !d$	1
$a \wedge b \wedge c \wedge d \wedge e \wedge \mathrm{f} \wedge !g$	1

SUMMARY

The majority of configuration-related bugs (more than 89%) detected involve two or less preprocessor macros. Our results are in line with the results of previous studies [19; 16] and support the effectiveness of sampling algorithms to detect configuration-related bugs, such as LSA and pair-wise [36].

RQ3: How do developers introduce configuration-related bugs?

We investigated how developers introduce the bugs we report in our empirical study. Our goal here is to identify whether developers introduce more bugs when implementing new functionalities or fixing other bugs in the code. According to the results, developers introduce more undeclared variables and functions, and unused variables and functions (73%) when introducing new functionalities, such as a new source file, or adding a new function [39]. In contrast to configuration-related syntax errors, the results are the opposite: developers introduce the majority of syntax errors (85%) when fixing existing code [12]. We now present

the results in the following order: syntax errors, undeclared functions, undeclared variables, unused functions, and unused variables.

Developers introduce syntax errors in eight different ways: **(I)** introducing a conditional `if` statement with a syntax error in its condition; **(II)** altering an `if` statement condition; **(III)** introducing directives to encompass `if` and `else if` statements; **(IV)** adding a new statement inside a conditional case; **(V)** modifying a function prototype; **(VI)** removing conditional directives that encompass closing brackets; **(VII)** adding optional elements in an array; and **(VIII)** modifying conditional parameters in a function call. Figure 5.7 illustrates these eight cases with small code excerpts. Overall, we found syntax errors in the following projects, as presented in Table 5.3.

I
```
  void f1(){
   …
+ #ifdef A
+   if (..)){
+     …
+   }
+ #endif
  }
```

II
```
  void f2(){
   …
  #ifdef A
-   if (..){
+   if (..)){
      …
    }
  #endif
  }
```

III
```
  void f3(){
   …
+ #ifdef A
    if (..)){
     …
    }
+ #endif
+ #ifdef B
    else if (..){
     …
    }
+ #endif
  }
```

IV
```
  void f4(){
   …
  #ifdef A
   case X:
+    p1 = call()
     …
   }
  #endif
  }
```

V
```
- void f5(){
+ #ifdef A
+ void f5(){
+ #endif
+ #ifdef B
+ int f5(){
+ #endif
     …
  }
```

VI
```
  void f6(){
   …
  #ifdef A
   if (..){
  #endif
    …
- #ifdef A
   }
- #endif
  }
```

VII
```
  void f7(){
   int p2[10] = {
   0,
   1,
+ #ifdef A
+  2
+ #endif
+ #ifdef B
+  3
+ #endif
   };
  }
```

VIII
```
  void f8(){
   call(param1
  #ifdef A
-   , param2
+   param3
  #endif
   );
  }
```

`+` Including line `-` Removing line

Figure 5.7: Introducing configuration-related syntax errors.

Table 5.3: Occurrences of configuration-related syntax errors.

Case	**Occurrences**
I	*Apache* (1), *Cherokee* (2), *Irssi* (1), *Libpng* (1), *Libssh* (3), and *Vim* (4).
II	*Apache* (1) and *Bash* (1).
III	*Apache* (1), *Dia* (1), and *Libxml2* (1).
IV	*Bash* (1), *Libxml2* (1), and *Vim* (1).
V	*Gnuplot* (1) and *Vim* (2).
VI	*Gnuplot* (1), *Libpng* (1), and *Xterm* (2).
VII	*Gnuplot* (1), *Xfig* (1).
VIII	*Libpng* (1), *Libssh* (1), and *Vim* (2).

Developers introduce configuration-related undeclared functions in three different cases: **(I)** adding a call to an existing function without checking the preprocessor directives that encompass such function definition; **(II)** adding a call to a function without including the header file with the function definition; and **(III)** changing a function definition without modifying the corresponding function calls. Figure 5.8 illustrates these three cases with small code excerpts. Table 5.4 presents the occurrences of undeclared functions.

Table 5.4: Occurrences of configuration-related undeclared functions.

Case	**Occurrences**
I	*Bash* (1), *Gnuchess* (1), *Gzip* (2), *Libpng* (6), and *Privoxy* (1).
II	*Lua* (1).
III	*Libssh* (1), and *Lua* (1).

```
I
#ifdef A
void func1 () {
  ...
}
#endif
+ void func2 () {
+   func1();
+ }
```

```
II
#ifdef B
void func3 () {
+     func4();
}
#endif
```

```
III
#ifdef C
void func5 () {
    func6();
}
#endif
- void func6 () {
+ void func6 (int p) {
    ...
}
```

`-` Removing line `+` Including line

Figure 5.8: Introducing configuration-related undeclared functions.

Figure 5.9 presents the only two cases we detected for undeclared variables. In case **(I)**, developers try to eliminate a shadowed declaration of variable `p1` at Line 6. However, they change the conditional directive at Line 1, raising an undeclared variable at Line 9. Developers introduce another undeclared variable following case **(II)**, i.e., they introduce a new source file that defines variable `p2` conditionally, but uses it in mandatory code. We found only one bug for each case: **(I)** *Libpng* (1), and **(II)** *Gzip* (1).

```
I
-  1. #ifndef A
+  2. #ifdef A
   3.   int p1;
   4. #endif
   5. #ifdef A
-  6.   int p;
   7.   p1 = func1();
   8. #else
   9.   p1 = func2();
  10. #endif
```

```
II
+ void func3 () {
+ #ifdef A
+   int p2;
+ #endif
+   ...
+   p2 = func4();
+   ...
+ }
```

`-` Removing line `+` Including line

Figure 5.9: Introducing configuration-related undeclared variables.

Developers introduce unused functions in two cases: **(I)** conditionally defining a function and calling it in code encompassed with different preprocessor directives; and **(II)** removing a call to a conditionally defined function, and adding another call to a mandatory function. Figure 5.10 depicts these two cases. We found unused functions following case **(I)**: *Bash* (4), *Libpng* (1), and *M4* (1); and **(II)**: *Libpng* (1).

```
                I                                    II
                                       - #ifdef A
                                       - void func3 () {
+  #ifdef A                            -    ...
+  void func1 () {                     - }
+     ...                              - #endif
+  }                                     void func4 () {
+  #endif                                   ...
+  void func2 (){                        }
+  #if defined(A) && defined(B)          void func5 () {
+     func1();                           #ifdef A
+  #endif                              -    func3();
+  }                                   +    func4();
                                         #endif
                                         }

 -  Removing line        +  Including line
```

Figure 5.10: Introducing configuration-related unused functions.

We found unused variables being introduced in the following three cases: **(I)** adding a new variable to an optional code without using such variable; **(II)** adding a new variable to mandatory code and using this variable only in optional code; and **(III)** moving the uses of a variable to optional code. Figure 5.11 depicts these three cases. We found the following occurrences of case **(I)**: *Bash* (6); **(II)**: *Bash* (8); and **(III)**: *Bash* (1), and *Libssh* (1).

```
         I                      II                       III
                       void func2 () {
  #ifdef A                ...                     void func4 () {
  ...                  +  int p2;                    int p3;
+ void func1 () {         ...                        ...
+    int p1;           +  #ifdef B                +  #ifdef C
+    ...               +     p2 = func3();           p3 = func5();
+ }                    +  #endif                  +  #endif
  #endif                  ...                        ...
                       }                          }

 +  Including line
```

Figure 5.11: Introducing configuration-related unused variables.

SUMMARY

Developers introduce configuration-related bugs by introducing new functionalities and by fixing existing code. Most undeclared variables and functions, and unused variables and functions (73%), developers introduce when adding new functionalities. In contrast to configuration-related syntax errors, the results are the opposite: developers introduce the majority of syntax errors (85%) when fixing existing code.

RQ4: For how long do configuration-related bugs remain in the code?

In this section, we analyze the time that developers take to fix configuration-related bugs. Our results show that the time varies from days to years. For example, developers fix a bug of the *Libssh* system (`keyfiles.c`) after 69 days. In contrast, developers took more than 5 years to fix the error in `parser.c` of *Gnuplot*. Notice that we only list bugs we know exactly when developers introduce them, and bugs already fixed.

Developers may take a long time to fix bugs due to different reasons. First, the configuration-related bugs may be difficult to detect because of variability [29]. Second, developers might have problems to understand code that they are not familiar with, possibly written by another developer [29]. Third, in case the bugs arise in not exercised or deliverable configurations, developers tend to rank the fixing task as lower priority [12].

SUMMARY

Developers take a long time to fix even simple configuration-related bugs, such as syntax errors, which are detected by traditional C tools, such as Gcc. However, the variability of program families hinders the detection of configuration-related bugs in practice.

Submitting Patches to Fix Bugs

We submitted 38 patches—for each bug not fixed—to 6 program families: *Bash* (21), *CVS* (1), *Libpng* (7), *Libssh* (6), *Vim* (2), and *Xfig* (1). We submitted these patches using bug tracking systems and via email directly to the main developer of the subject system. We consider that developers accept a patch when they mention that it is a problem by email, or keep the patch open after updating information, such as priority. Conversely, we consider that developers reject the patch when they mention it is not a problem by email, or update this information on the patch. Thus, developers accepted 13 patches, rejected 6 patches, ignored 15 patches, and we did not receive feedback regarding 4 patches we submitted to *Libssh*. Notice that we do not consider these 4 bugs of *Libssh* in our statistics. We present information about the patches submitted in Table 5.6, which does not include the 15 patches ignored by the *Bash* developers.

Table 5.5: Time to fix configuration-related bugs.

Family	**File**	**Kind**	**Days to Fix**
Apache	ssl_util_ssl.c	syntax error	278
Apache	ab.c	syntax error	222
Apache	mod_include.c	syntax error	353
Bash	getcppsyms.c	syntax error	119
Dia	app_procs.c	syntax error	232
Dia	preferences.c	syntax error	385
Gnuplot	plot.c	syntax error	160
Gnuplot	util.c	syntax error	7
Gnuplot	parser.c	syntax error	1924
Gnuplot	graph3d.c	syntax error	78
Gnuplot	datafile.c	syntax error	414
Gzip	deflate.c	undeclared function	6678
Gzip	util.c	undeclared function	5983
Libpng	iccfrompng.c	undeclared function	1289
Libpng	iccfrompng.c	undeclared function	1289
Libpng	iccfrompng.c	undeclared function	1289
Libpng	iccfrompng.c	undeclared function	1289
Libpng	pngpixel.c	undeclared function	1289
Libpng	pngpixel.c	undeclared function	1289
Libpng	pngrutil.c	undeclared variable	530
Libpng	pngtrans.c	syntax error	259
Libssh	keyfiles.c	undeclared function	69
Libssh	channels.c	unused variable	4
Libssh	dh.c	syntax error	268
Libxml2	xmlregexp.c	syntax error	150
Libxml2	xpath.c	syntax error	2
Lua	loadlib_rel.c	undeclared function	748
Lua	loadlib_rel.c	undeclared function	999
Vim	ex_cmds2.c	syntax error	99

We submitted 20 patches to *Bash* and developers accepted only one. Four bugs do not happen in practice (i.e., they are false positives), as the build system avoids the specific configurations they appear in. In addition, one particular developer confirmed but ignored the 15 patches we report to fix unused variables: "*I don't care about unused variables too much; the compiler gets rid of them. So, they have no cost.*"

Despite having no performance cost, unused variables and functions slightly pollute the code, which might explain other developers caring about them. For instance, we find a single patch to *Gnuchess* that fixes 19 unused variables.

Regarding the patches we submit to *Libpng*, developers accepted all 7 patches, and they have already fixed the bugs in the software repository. *Libpng* developers fixed a syntax error immediately after our patch submission. *Vim* developers accepted one patch and rejected another by just arguing that it arises in an invalid configuration. Developers rejected a patch submitted to the *Xfig* program family as well. In this case, developers mention they do not use (at least for now) the erroneous macro we identify. According to the following quotation, it seems that the macro will be used when they decide to distribute the *Xfig* manual in Japanese. So, we still count this as an error, since it may arise in the future, as mentioned by a developer: "*It is not used now as Japanese PDF manual is not distributed with Xfig, and I think you can simply ignore it.*"

5.2.3 Threats to Validity

In this section, we discuss some threats to validity.

Construct Validity

Checking whether the configuration-related bugs detected are real or represent false positives threatens **construct validity**. To minimize this threat, we perform two tasks: (i) for the systems we know preprocessor macro constraints in advance, we set *TypeChef* to take them into account and consequently avoid analyzing invalid configurations; and (ii) ask the actual developers to confirm each bug not fixed in the software repository. Developers accepted 13 configuration-related bugs we report.

Internal Validity

We analyzed the bugs manually, which is a time-consuming and error-prone activity, which threatens internal validity. Nevertheless, because we got feedback from developers and confirmed the bugs we report, we minimized this threat.

Our strategy excludes `#include` directives to eliminate external libraries in order to scale. However, notice that we may face false negatives due to the exclusion of these `#include` directives, which makes our strategy unsound. Some external libraries may

Table 5.6: Patches submitted to subject systems.

Family	File	Accept	Status	Variable / Function
Bash	execute_cmd.c	valid	open	syntax error
Bash	execute_cmd.c	valid	open	`arith_cmd` undeclared
Bash	bashline.c	invalid	closed	`add_history` undeclared
Bash	flags.c	invalid	closed	`init_hist` undeclared
Bash	jobs.c	invalid	closed	`imp_sigchld` undeclared
Bash	strerror.c	invalid	closed	`strerror` undeclared
Cvs	buffer.c	valid	open	syntax error
Libpng	iccfrompng.c	valid	fixed	`init_io` undeclared
Libpng	iccfrompng.c	valid	fixed	`get_iCCP` undeclared
Libpng	iccfrompng.c	valid	fixed	`read_info` undeclared
Libpng	iccfrompng.c	valid	fixed	`destroy` undeclared
Libpng	pngpixel.c	valid	fixed	`get_depth` undeclared
Libpng	pngpixel.c	valid	fixed	`get_type` undeclared
Libpng	pngvalid.c	valid	fixed	syntax error
Libssh	keys.c	valid	fixed	syntax error
Libssh	keys.c	valid	fixed	syntax error
Libssh	sftp.c	-	open	`sftp_read` undeclared
Libssh	main.c	-	open	`sftp_open` undeclared
Libssh	torture_rand.c	-	open	`ssh_pthread` undeclared
Libssh	chmodtest.c	-	open	`sftp_new` undeclared
Vim	os_unix.c	valid	open	syntax error
Vim	if_mzsch.c	invalid	closed	syntax error
Xfig	w_cmdpanel.c	invalid	closed	syntax error

(-) We did not receive feedback regarding 4 bugs of *Libssh*, so we do not consider them in our statistics.

introduce additional code through macro definitions that may cause configuration-related bugs into the family source code. In this context, our strategy may miss some syntax errors. Moreover, the strategy may yield false positives due to types and macros that the CDT parser does not identify, i.e., these types and macros may not be included in our `stubs.h` file. So, we add the type or macro manually, which is an error-prone task.

The strategy considers only one configuration of header files when performing our second simplification to scale. We used *Gcc* and generated header files for the *Linux* platform only. However, notice that we may face false negatives due to this simplification, which threatens internal validity. In this context, our strategy may miss some configuration-related bugs that occur only for other platforms, such as *Windows* and *Mac OS*. Still, in our study, we found

63 configuration-related bugs, and we confirmed them either by checking if developers fixed them in software repositories or by getting feedback from developers.

Our strategy analyzes only updated and added files in software repositories from the second to the last commit. However, this approach may lead to false negatives. For instance, developers may update a macro definition in a file *A*, which leads to errors in a different file *B*. In our approach, because only *A* has been modified, we only analyze *A*. However, later, if developers modify *B*, the strategy may catch the bug. Furthermore, we may miss some bugs during the analysis of the repositories since we analyze only the trunk, i.e., branches may contain configuration-related bugs as well.

External Validity

We analyzed 40 systems of different domains, sizes, and different number of developers. We selected well-known and active program families used in industrial practice. The families communities exist for years and seem very active: there are commits in 2016. In this way, we alleviate this threat.

Chapter 6

Catalog of Refactorings

In this chapter, we present our catalog of refactorings to resolve undisciplined directives and an evaluation of the evaluation of the catalog of refactorings considering four perspectives: frequency of application possibilities in real-world software systems, opinion of developers, behavior preservation, and quality of the refactored code in terms of amount of additional code clone, lines of code, and preprocessor directives.

In Section 6.1, we present the catalog of refactorings, explaining the code transformation and showing the preconditions of each refactoring. In Section 6.2, we present the evaluation of the catalog of refactorings in detail.

6.1 Refactorings

Our refactorings are transformations, where each transformation is an unidirectional refactoring and consists of two templates of C code snippets: Left-Hand Side (LHS) and Right-Hand Side (RHS). The LHS defines a template of C code that contains undisciplined preprocessor usage. The RHS defines a corresponding template of the refactored code without undisciplined preprocessor usage. We can apply a refactoring whenever the LHS template is matched by a piece of C code and satisfies the preconditions ($\rightarrow$). A matching is an assignment of all meta-variables occurring in the LHS/RHS templates to concrete values arising from the code. We highlight meta-variables using capital letters and $\oplus$ to represent boolean operators. Any element not mentioned in both C code snippets remains unchanged, so the refactoring templates only show the differences between pieces of code.

To define this catalog, we analyzed preprocessor directives in 12 C subject systems, and identified recurrent patterns of undisciplined directives that occur frequently in practice. In Table 6.1, we present the characterization of these systems. Overall, we defined 14 refactorings and classify them into four categories: *single statements*, *conditions*, *wrappers*, and *comma-separated elements*. Next, we present the catalogue of refactorings. In Appendix A, we list the complete list of refactorings, including some refactoring variations that we omit in this chapter.

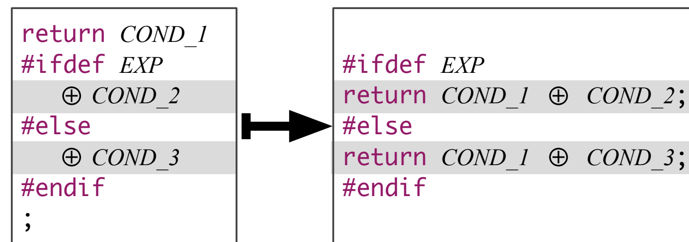
Table 6.1: Subject characterization

Project	**Application Domain**	**LOC**	**#ifdefs**
Apache	web server	144 768	2173
Bc	calculator	5177	91
Dia	diagramming software	28 074	320
Expat	XML library	17 103	362
Flex	lexical analyzer	16 501	216
Fvwm	windows manager	102 301	1375
Ghostscript	postscript interpreter	1 536 979	3168
Gnuchess	chess player	9293	67
Gzip	file compressor	5809	298
Lighttpd	web server	38 847	933
Lua	programming language	14 503	193
Mptris	game	4988	61
Total		**1 916 828**	**9257**

6.1.1 Single Statements

A single statement contains no compound blocks, such as variable initializations, function calls, and `return` statements. In Refactoring 1, we present our refactoring to resolve undisciplined preprocessor usage in single statements. In this refactoring, we duplicate language tokens to encompass with preprocessor directives entire statements only. Notice that we duplicate the token *COND_1* to make the preprocessor directive disciplined. We use a `return` statement as an example, but we handle similar statements in the same way.

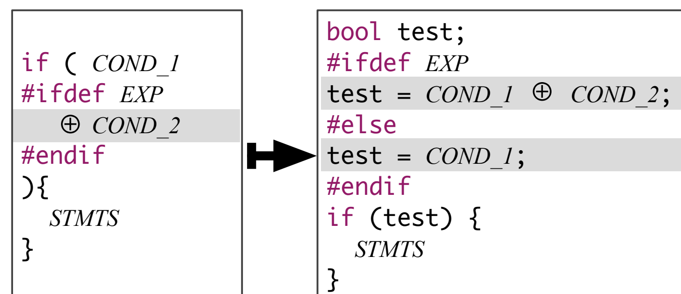
Refactoring 1 ⟨undisciplined returns⟩



6.1.2 Conditions

To resolve undisciplined directives surrounding boolean expressions (used in `if` and `while` statements), we propose Refactoring 2. In this refactoring, we use an extra variable to preserve the statement's conditions. In this sense, we define a precondition that the code is not using the specific identifier (`test`), as we cannot define variables with the same identifier in the same scope. We refactor `while` statements with undisciplined conditions using a similar refactoring.

Refactoring 2 ⟨undisciplined `if` conditions⟩

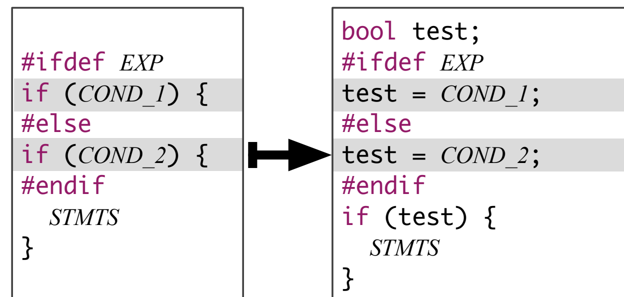


(→) `test` is not used in the code

6.1.3 Wrappers

In Refactoring 3, we target another case of undisciplined preprocessor usage: alternative statements. We use an alternative `if` statement as an example, but there are similar refactorings for other alternative control-flow statements, such as `while` and `switch` statements. In this refactoring, we also need an extra program variable to keep the statement condition. Notice that `test` receives the evaluation of *COND_1* or *COND_2* depending on whether we define macro *EXP* or not. Likewise, we define a precondition that `text` is not used in the code to avoid possible compilation errors.

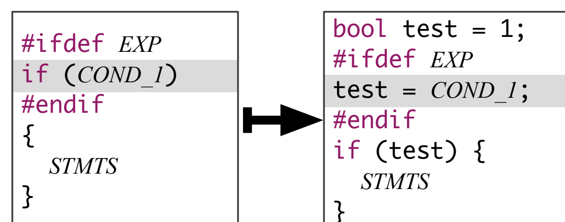
Refactoring 3 ⟨alternative `if` statements⟩



(→) `test` is not used in the code

In Refactoring 4, we present a refactoring to remove wrappers. In this refactoring, we also use variable `test` to preserve the statement's condition and to discipline the preprocessor directive. We use an `if` wrapper as an example, but there are similar refactorings for removing undisciplined `while`, `for`, and `else-if` wrappers.

Refactoring 4 ⟨if wrapper⟩



(→) `test` is not used in the code

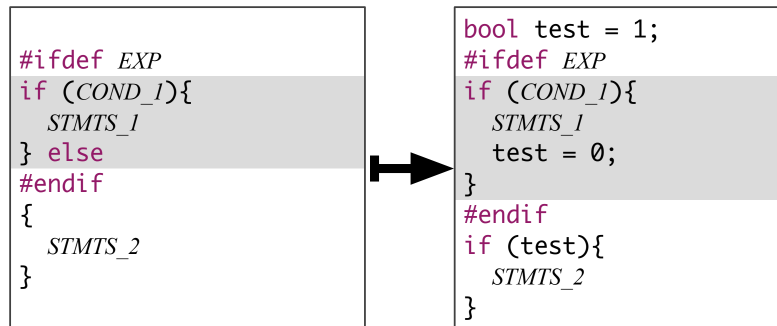
In Refactoring 5, we define a refactoring to remove `if` statements ending with an `else` statement. In this case, we replace the `else` by another `if` statement to resolve the undisciplined usage of the preprocessor. In this refactoring, variable `test` works like a flag to avoid executing *STMTS_2* when macro *EXP* is disabled.

6.1.4 Comma-Separated Elements

Refactoring 6 targets undisciplined directives in comma-separated program elements. In this refactoring, we set a precondition that the original code does not define a macro `PARAM` or contains a token with that name, such as a type definition or identifier. If we change a macro

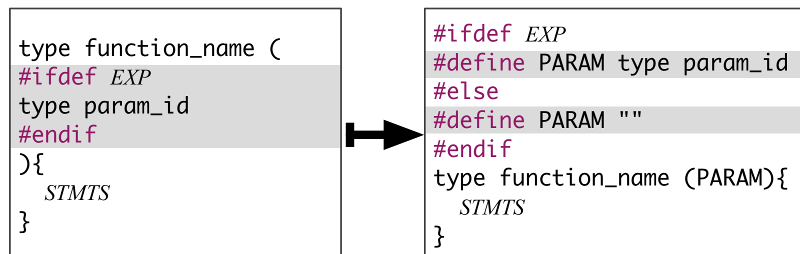
definition that the original code is already using, we may introduce behavioral changes. This way, we modify the code locally without global impact. We handle other types of comma-separated elements, such as array and enum elements, with a similar refactoring.

Refactoring 5 ⟨`if` statements with an `else`⟩



(→) `test` is not used in the code

Refactoring 6 ⟨undisciplined function definitions⟩



(→) `PARAM` is not used in the code

6.2 Evaluation

In this section, we present the settings of our study performed to evaluate the catalog of refactorings. We performed complimentary empirical studies to evaluate the catalog of refactorings with regards to the frequency of application possibilities in practice, opinion of developers, behavior preservation, and quality of the refactored code in terms of code clone, LOC, and number of preprocessor directives. The goal of our evaluation is to provide evidence that developers prefer to use our refactored code instead of using the preprocessor in undisciplined ways, to increase confidence that our catalog of refactorings resolves undisciplined preprocessor usage without introducing behavioural changes, and to show that there

are many application possibilities to use our refactorings in real-world C projects. All data used in this study are available on our Website.[1]

6.2.1 Overall Study Design

In particular, to answer the following research questions:

- **RQ1:** What is the frequency of possibilities to apply the refactorings in practice?
- **RQ2:** What is the opinion of developers regarding the catalog of refactorings?
- **RQ3:** Are the refactorings behavior preserving?
- **RQ4:** Do the refactorings increase code clones, LOC and preprocessor directives?

To answer **RQ1**, we analyzed 63 subject system searching for opportunities to apply the refactorings of the catalog in practice. We considered systems of different sizes and from various domains, such as games, operating systems, web servers, and database systems.

Regarding **RQ2**, we analyzed data from an online survey among 202 developers [29]. We asked developers about their code preferences showing two equivalent code snippets: (1) the original code from a real-world system with undisciplined directives, and (2) a disciplined version of the original code snippet created by applying one of our refactorings.

To answer **RQ3**, we developed a formal model of a subset of the C language and a corresponding code generator, based on *Alloy*, to automatically generate program families with application possibilities for our refactorings and test cases. Appendix B presents more information about the C model. In addition, we used *BusyBox*, *OpenSSL*, and *SQLite*, three real-world systems with test cases available, and applied our refactorings. For all subjects, we ran the test cases before and after applying our refactorings, in the generated program families as well as in the three real-world systems to verify behavior preservation.

To answer **RQ4**, we used a similarity detector to identify clones in the parts modified by the refactorings and compare the original and the refactored source files. Then, we counted the LOC and the number of preprocessor directives of the original source files and the LOC of the source files after applying our refactorings.

6.2.2 Results and Discussion

Next, we present the results and discuss each research question.

[1] http://www.dsc.ufcg.edu.br/~spg/catalog/.

RQ1: What is the frequency of possibilities to apply the refactorings in practice?

To count the number of application possibilities for the refactorings of our catalog, we performed an analysis of 63 C popular systems, including *Bash*, *Gcc*, *Linux*, and *Vim*. We selected projects from different domains, such as games, text editors, web servers, and operating systems. Furthermore, our analysis considers popular, big and mature projects, but also newer and smaller systems without widespread use in practice. We selected the projects based on the corpus of prior studies on the C preprocessor [3; 67], covering a range of different sizes (2.6 thousand to 7.8 million lines of code). Moreover, we also selected systems that use *GitHub* and its pull request infrastructure actively to submit patches to the subject systems.

We used the *SrcML*[2] tool to identify application possibilities for our refactorings. *SrcML* transforms C source code into an XML representation, which we used to detect the different patterns of undisciplined directives. Table 6.2 presents the number of application possibilities in the 63 systems analyzed in this study. Overall, we found 5670 opportunities, showing that we can apply our refactorings to several real-world subject systems.

According to our analysis, Refactorings 2 and 6 are the most frequent ones in practice, while Refactorings 1 and 3 are the less frequent. We found that some projects heavily make use of undisciplined preprocessor directives, such as *Gcc*, *Glibc*, *Linux*, and *Vim*. There are also projects that avoid undisciplined directives at all, such as *Bison* and *Mpsolve*, and others use only a few undisciplined directives, such as *Libssh* and *Totem*. We found application possibilities in almost all projects analyzed (97%) in this study, showing that developers use undisciplined directives in practice.

SUMMARY

We considered subject systems of different sizes and from various domains, we found 5670 application possibilities (90 per project) for the refactorings in practice. There are places to apply the refactorings in almost all systems (97%) analyzed in this study, showing that developers use undisciplined directives in practice.

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

Table 6.2: Application possibilities in 63 C projects.

Project	**Version**	**Domain**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**
Angband	4.0.4	game	0	0	1	1	0	1
Amxmodx	1.8.3	server administration tool	0	21	7	12	84	6
Asfmapready	3.2.1	command line tools	0	0	3	0	0	0
Bash	4.2	command language interpreter	2	5	26	12	6	7
Berkeley DB	4.7.25	database system	5	18	6	1	9	16
Bison	2.0	parser generator	0	0	0	0	0	0
Busybox	1.23.1	common UNIX utilities	20	15	6	20	4	19
Cherokee	1.2.101	Web server	0	7	1	2	23	0
Clamav	0.97.6	antivirus software	9	9	4	4	17	12
Collectd	5.5.0	system administration tool	0	5	2	0	1	3
Curl	7.46.0	data transferring tool	5	19	2	8	38	7
Cvs	1.11.17	version control system	4	23	7	14	26	6
Dmd	2.069.2	language interpreter	2	37	12	9	2	15
Emacs	24.4	text editor	20	41	9	24	34	14
Ethersex	0.1.2	processor firmware	5	11	30	11	5	3
Freeradius	3.0.10	radius server	0	19	1	4	21	44
Gawk	3.1.4	interpreter	0	11	4	7	32	5
Gcc	4.9.2	compiler	51	371	32	172	121	114
Glibc	2.20	C library	29	53	16	76	71	38
Gnumeric	1.12.20	spreadsheet program	4	0	1	1	0	5
Gnuplot	4.6.1	plotting tool	2	6	4	15	42	7
Irssi	0.8.15	chat client	0	0	3	1	4	0
Kerberos	1.14	network authentication protocol	0	10	4	3	3	4
Kindb	1.0	database system	0	0	7	0	2	0
Hexchat	2.10.2	chat client	0	0	5	2	2	5
Libdsmcc	0.5	DVB library	0	0	0	0	0	0
Libpng	1.5.14	PNG library	5	12	9	5	23	1
Libsoup	2.41.1	SOUP library	0	0	0	0	0	0
Libssh	0.5.3	SSH library	0	0	0	0	1	1
Libxml2	2.9.0	XML library	1	27	6	5	57	8
Linux	3.18.5	operating system kernel	129	60	40	71	277	518
M4	1.4.17	macro expander	0	3	4	5	15	2
Machinekit	0.1	machine control platform	0	5	3	1	3	4
Mapserver	7.0.0	Web application framework	0	4	4	2	5	2
Mongo	1.1.8	MongoDB client library	0	2	0	0	1	1
Mpsolve	2.2	mathematical software	0	0	0	0	0	0
Opensc	0.15.0	smart card tools and middleware	0	0	3	0	0	7
Openssl	1.0.2	SSL library	3	11	23	8	114	9
Opentx	2.1.6	radio transmitter firmware	0	3	1	1	2	7
Openvpn	2.3.6	virtual network tool	8	14	5	5	20	2
Ossec-hids	2.8.3	intrusion detection system	0	5	12	4	9	13
Pacemaker	1.1	cluster resource manager	0	1	0	0	0	5
Parrot	7.0.2	virtual machine	0	1	0	1	0	38
Pidgin	2.10.11	chat client	11	17	2	2	4	10
Prc-tools	2.3	gcc for Palm OS	1	0	0	0	0	0
Privoxy	3.0.19	proxy server	2	11	12	7	9	5
Python	2.7.9	language interpreter	34	33	12	14	49	72
Rcs	5.7	revision control system	2	0	0	0	0	1
Retroarch	1.2.2	libretro API	9	11	11	8	23	14
Sendmail	8.14.6	mail transfer agent	5	21	16	3	9	2
Sleuthkit	4.2.0	command line tools	0	1	5	0	15	6
Sqlite	3080200	database system	8	7	4	5	17	6
Syslog-ng	3.7	log management application	0	2	1	0	0	6
Sylpheed	3.3.0	e-mail client	13	2	3	0	2	2
Taulabs	20150922	autopilot system library	0	6	3	3	8	24
Tk	8.6.3	widget toolkit	2	5	2	1	0	8
Totem	2.17.5	video application	0	0	0	0	1	1
Uwsgi	1.9	application container	0	3	0	0	3	9
Vim	6.0	text editor	62	279	46	82	365	14
Wiredtiger	2.6.1	data management platform	0	3	0	0	0	9
Xfig	3.2.4	vector graphics editor	1	3	0	0	20	10
Xorg-server	1.9.3	window system	14	47	14	20	48	7
Xterm	2.2.4	terminal emulator	2	15	1	9	1	6
Total			**470**	**1295**	**435**	**661**	**1648**	**1161**

RQ2: What is the opinion of developers regarding the catalog of refactorings?

To learn about the opinion of developers regarding our catalogue of refactorings and undisciplined directives, we performed a survey among 202 developers and submitted 28 patches to real-world C project converting undisciplined into disciplined preprocessor directives.

Survey

We performed an online survey among 202 developers asking for developer's preferences [29]. To select participants, we collected information about developers by mining the repositories of several popular systems, including the *Linux Kernel* and *Apache*. This way, we randomly selected a number of developers from each system, and sent 3091 emails asking developers to fill our survey. Overall, 202 (6.5%) developers completed the online survey, as discussed in Chapter 3.

In particular, we asked three specific questions about our refactorings. We presented three pairs of two equivalent code snippets: (1) the original code from a real C project; and (2) the refactored version of the original code created by applying one of our refactorings. For each pair of code snippets, we asked developers about their preferences. Next, we present the code snippets used in the survey questions.

In Figure 6.1, we present a concrete instance of Refactoring 1. Figure 6.1 (a) shows part of the *Vim* source code with undisciplined preprocessor directives. Figure 6.1 (b) presents the refactored (i.e., disciplined) version of the code snippet. According to the results of our survey, 87% of developers preferred the refactored version of the code snippet, 7% preferred the undisciplined version, and 6% mentioned that they have no preference.

```
mfp = open(mf_fname
#ifdef UNIX
   , (mode_t)0600
#else
   , S_IREAD | S_IWRITE
#endif
);
```
(a)

```
#ifdef UNIX
mfp = open(mf_fname, (mode_t)0600);
#else
mfp = open(mf_fname, S_IREAD | S_IWRITE);
#endif
```
(b)

Figure 6.1: Duplicating tokens to discipline preprocessor directives.

In Figure 6.2, we show a pair of code snippets, in which we show an application of Refactoring 2. Figure 6.2 (a) shows part of the *Libpng* source code with the refactoring application

possibility, that is, the original code with undisciplined preprocessor usage. Figure 6.2 (b) presents the refactored version of the code snippet (i.e., with disciplined preprocessor usage only). From the 202 developers that completed the online survey, 67% preferred the refactored version, 20% preferred the undisciplined version, and 13% of the developers stated that they have no preference.

```
if (bit_depth < 8
#if defined (TESTS_SUPPORTED)
    && row != NULL
#endif
){
    // Lines of code here..
}
              (a)
```

```
int test = (bit_depth < 8);
#if defined (TESTS_SUPPORTED)
test = test && (row != NULL);
#endif
if (test){
    // Lines of code here..
}
              (b)
```

Figure 6.2: Adding a local variable to discipline preprocessor directives.

In Figure 6.3, we depict an instance of Refactoring 6. Figure 6.3 (a) shows part of the source code of *Vim*, while Figure 6.3 (b) presents the refactored version of the code snippet. From the developers that completed our survey, 57% preferred the refactored version, 30% preferred the undisciplined version, and 13% stated that they have no preference.

```
void msgNetbeansW32(
#if defined (GUI_W32)
  Xt client,
#endif
XtInputId *id){
  // lines of code..
}
          (a)
```

```
#if defined (GUI_W32)
#define PARAM Xt client,
#else
#define PARAM
#endif
void msgNetbeansW32(PARAM XtInputId *id){
  // Lines of code..
}
                 (b)
```

Figure 6.3: Using macros to discipline preprocessor directives.

So, overall we considered three types of refactorings: (1) a refactoring that introduces local variables, (2) a refactoring that introduces macros, and (3) a refactoring that duplicates a few language tokens. We conclude that most developers agree with our three strategies of resolving undisciplined directives. The repetition of a few tokens, as we present in Figure 6.1, received support from almost 90% of developers, while the use of preprocessor macros, as presented in Figure 6.3, received the weakest support. So, developers seem prefer to resolve undisciplined directives by duplicating a few tokens and by adding a local variable, showing that we should make use of preprocessor macros more carefully.

SUMMARY

The results of our survey reveal that most developers preferred to use the refactored (i.e., disciplined) code instead of using the preprocessor in undisciplined ways. Almost 90% *of developers support the repetition of a few tokens to discipline preprocessor directives, but we should use additional macros more carefully.*

Patches

To further understand how developers perceive undisciplined directives, we submitted 38 patches to open source projects. For selecting projects, we used *SHTorrent*[3] for identifying active projects that heavily use pull requests on *GitHub*,[4] the infrastructure we used to submit the patches. We submitted 28 patches to the most active projects in which we found application possibilities for our refactorings. Overall, developers accepted 21 (75%) patches submitted: *Angband* (1); *Amxmodx* (1); *Asfmapready* (1); *Collectd* (1); *Curl* (1); *Dmd* (1); *Libpng* (1); *Linux* (1); *Mapserver* (1); *Machinekit* (1); *Mongo* (1); *Opensc* (1); *Openssl* (1); *Opentx* (1); *Ossec-hids* (1); *Retroarch* (1); *Sleuthkit* (1); *Syslog-ng* (1); *Taulabs* (1); *Uwsgi* (1); and *Wiredtiger* (1).

The feedback we received supports the perception that undisciplined preprocessor usage influences the code quality negatively. We received feedback by using the *GitHub*, which allows us to talk to developers by including comments in each patch. We submitted one refactoring converting an undisciplined into a disciplined directive per patch, and one patch per system. In this sense, our patches were judged by a broad audience of developers of the 28 systems used in this study. This way, we minimized the problem of having many patches accepted by the same developers.

For most patches, developers agreed with our suggestion to resolve the undisciplined preprocessor usage. For example, one developer mentioned that the refactoring "makes sense [to him] and it is a good idea." Developers accepted 12 patches without asking for changes. However, for some patches, developers asked us, for example, to rename local variables, and to include or exclude spaces between brackets to better follow the project's standards. For instance, one developers said that "[the patch] would be fine except for the unnecessary

[3] http://ghtorrent.org/
[4] https://github.com/

extra parentheses." Table 6.3 presents the patches developers accepted after we applied a few minor changes.

Table 6.3: Patches accepted after minor changes.

Project	**Changes requested by developers**
Dmd	Remove unnecessary parentheses.
Linux	Fix typo.
Libpng	Duplicate the code instead of adding a new local variable.
Machinekit	Fix indentation.
Openssl	Rename local variable.
Opentx	Remove unnecessary parentheses.
Retroarch	Use integer instead of boolean.
Syslog-ng	Extract code to a helper function.
Uwsgi	Extract directives to a macro.

We noticed that some developers are resistant to apply any changes to their source code. Such developers raised some reasons, saying that "we know that [the code] works, and a change there would need very close scrutiny to ensure [that] no combination of features gets broken, review time needed." In another project, developers complained about introducing local variables, for example, a developer said that "I agree with you. But the resources are limited [in our context] and we should make every effort to not waste them." So, they did not accept our patch because of a new local variable that we used to discipline the preprocessor directives. Table 6.4 presents the patches rejected by developers.

Table 6.4: Patches rejected.

Project	**Argument against changes**
Ethersex	Patch defines a new local variable, we have limited resources.
Freeradius	Patch needs improvements, it is harder to read.
Hexchat	New code is harder to read.
Kerberos	The code is old, what we need is to remove the conditional directives.
Irssi	Patch needs to be improved.
Openvpn	The code is working and changes will require test effort and time.
Pacemaker	Changes require test effort and time.

SUMMARY

Overall, we conclude that developers support the idea of converting undisciplined into disciplined directives. Developers accepted 21 (75%) out of the 28 patches submitted, showing more evidence that they prefer to use disciplined directives.

RQ3: Are the refactorings behavior preserving?

The C preprocessor hinders the development of tool support available in other languages, such as automated refactoring [21; 17; 22; 23; 24; 25]. After applying refactorings in C, developers need time to review the different configurations of the source code. This way, it is important to make sure that the refactorings of our catalog do not introduce behavioral changes. To get confidence into our catalog, we analyzed a subset of application possibilities to improve confidence in behavior preservation by using manual code reviews. Furthermore, we used automated testing applying the refactoring in: (1) programs automatically generated based on a formal model of a subset of the C language, which we specified using *Alloy*;[5] and (2) real-world projects with test cases available, as discussed next.

Formal Model

To test behavior preservation, we use regression testing by running test cases before and after applying the refactorings. For this purpose, we used a strategy proposed by Soares et al. [125], as illustrated in Figure 6.4. In *Step 1*, we created a formal model to generate program families (i.e., *A, B,* and *C*) with an opportunity to apply our refactorings. In *Step 2*, we select each family generated previously (e.g., family *A*) and use the preprocessor to create all different configurations of that specific family. In Figure 6.4, we show the two possible configurations of family *A*: (*C1*) with macro `EXP` enabled; and (*C2*) with macro `EXP` disabled. Then, for each configuration of the generated family, in *Step 3*, we generate test cases automatically by using a test case generator for C programs [126]. In *Step 4*, we apply a refactoring of our catalogue to each family generated previously using *Colligens*, our tool that we will present in Chapter 7. For example, considering our example family *A*, it generates an equivalent family *A'* without undisciplined preprocessor usage. In *Step 5*, we use the preprocessor to generate each possible configuration for the refactored families

[5] `http://alloy.mit.edu`

(i.e., *C1'* and *C2'*). In *Step 6*, we run the test cases using the original and refactored families to search for behavioral changes. For instance, the output of a test case for family *A*, with macro `EXP` enabled, must be the same as the output for family *A'*, with macro `EXP` enabled, giving the same input value for both families. The same must hold for all configurations of the generated program families.

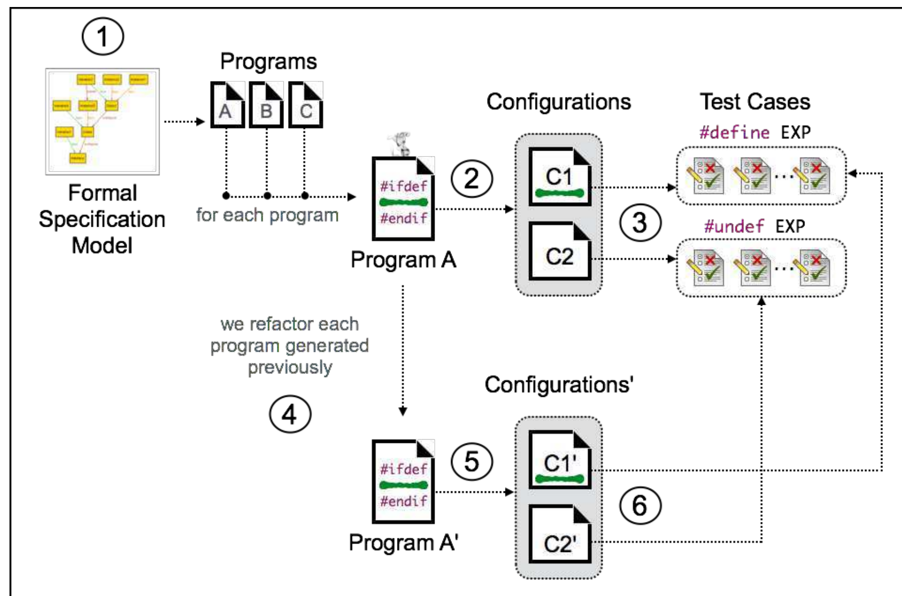


Figure 6.4: Applying regression testing to verify behavior preservation.

In Figure 6.5 (a), we list a family generated with the possibility to apply Refactoring 2. The preprocessor directives at Lines 11 and 13 split up parts of the `if` condition, that is, it is an undisciplined directive. In Figure 6.5 (b), we present the code snippet generated after applying Refactoring 2. Notice that our strategy generates small families like the one presented in Figure 6.5 (a). This way, we can use a *brute force* approach to test for behavioral changes, i.e., checking all possible configurations.

Table 6.5 presents the results obtained from generating 10K program families for each refactoring. Regarding Refactoring 4, which contains two variations, we generated 10K families for each. According to the results, our formal model generated up to 77% of valid families. We applied our refactorings to all valid families and we introduced no compilation errors after applying the refactorings. Overall, we detected 13 behavioral changes: five behavioral changes caused by a conceptual problem in the first version of Refactoring 2

```
1.  int Global0 = 1;
2.
3.  float F1(float P0){
4.    Global0 = 0;
5.    return P0;
6.  }
7.
8.  float F0(float P0){
9.    float Local0 = 1;
10.   if (Global0
11.#ifdef TAG
12.        & F1(P0)
13.#endif
14.   ){
15.     Local0 += 9;
16.     return P0;
17.   }
18.   return P0;
19.}
            (a)
```

```
1.  int Global0 = 1;
2.
3.  float F1(float P0){
4.    Global0 = 0;
5.    return P0;
6.  }
7.
8.  float F0(float P0){
9.    float Local0 = 1;
10.   bool test = Global0;
11.#ifdef TAG
12.   test = test & F1(P0);
13.#endif
14.   if (test){
15.     Local0 += 9;
16.     return P0;
17.   }
18.   return P0;
19.}
            (b)
```

Figure 6.5: Examples of generated and refactored programs.

(already fixed in the current version), and eight behavioral changes caused by bugs in the implementation of our refactorings.

Table 6.5: Results of behavioral changes regarding the generated families.

	R2	**R3**	**R4**	**R5**
Valid Programs	7746	7723	14 448	6700
Invalid Programs	2254	2277	5452	2300
Valid Refactorings	7746	7723	14 448	6700
Behavioral Changes	5	1	5	2
Behavioral Changes (after fixes)	0	0	0	0

Refactoring 6.6 presents the previous version of Refactoring 2 that introduced behavioral changes. The problem with this refactoring is that the C language does not specify the order of precedence when evaluating expressions with boolean operators, which makes different compilers to evaluate `if` conditions differently. We detected this problem when verifying the behavior of the family presented in Figure 6.5 (a). When running this program family on *Linux* using *Gcc*, the compiler evaluates the function call (`F1`) at Line 12 before evaluating variable `Global0` at Line 10. On the other hand, *Gcc* evaluates variable `Global0` first when running the program on *Mac OS*. Notice that, by applying Refactoring 6.6, as we can see in Figure 6.5 (b), variable `Global0` is always evaluated before calling function `F1`. This way, Refactoring 6.6 introduces a behavioral change when running the program family on the *Linux* platform. Refactoring 2 fixed this problem.

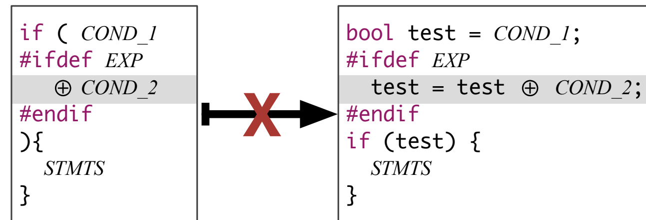


Figure 6.6: Undisciplined `if` condition that introduced behavioral changes.

Regarding behavioral changes caused by bugs in the implementation of our refactorings, we found five bugs in the pretty printer, which missed white spaces between identifiers and operators, and three bugs related to the use of integer instead of boolean variables in `if` conditions. Our catalogue of refactorings uses the boolean type as defined in the `stdbool` library. By using integer variables, the implementation of the catalog of refactorings caused behavioral changes when converting float values to integer. We fixed all these bugs in the current implementation.

SUMMARY

By performing regression testing in the generated program families to verify behavior preservation, we found and fixed a few behavioral changes introduced by our refactorings and a number of problems in the implementation of our catalog, the majority related to unspecified behavior in the C language. This way, we improved confidence that the refactorings are behavior-preserving.

Application in Practice

To evaluate behavior preservation in real-world subject systems, we implemented the refactorings on the *Morpheus* infrastructure [92]. We selected *BusyBox*,[6] *OpenSSL*,[7] and *SQLite*[8] as our case studies. *BusyBox* is a project that combines small versions of many common *UNIX* utilities into a single small executable. It contains 522 files and 19K lines of C code (version 1.18.5). *BusyBox* provides 792 preprocessor macros implemented with preprocessor directives. *OpenSSL* implements secure internet protocols, contains 733 files and 233K lines of C code. *OpenSSL* provides 589 preprocessor macros. *SQLite* is a library

[6] http://www.busybox.net/
[7] https://www.openssl.org/
[8] https://www.sqlite.org/

implementing a relational database management system, its code base consists only of two source-code files (amalgamation version 3.8.1), with 143K lines of C code, which can be configured using 93 preprocessor macros.

We applied our refactorings to all 45 cases of undisciplined directives of *BusyBox*, all 146 cases in *OpenSSL*, and all 33 cases in *SQLite*, as presented in Table 6.6. *BusyBox* comes with a test suite with 410 test cases for 74 files, out of which 46 tests fail (which we ignored during our evaluation). *OpenSSL* provides a test suite for each individual component, including the implementation of hashing functions (such as *MD5* and *SHA-256*) and key-generation and encryption algorithms. The test suite of *OpenSSL* does not indicate the exact number of test cases, but it provides an output message informing the failure or success of the complete test suite. For *SQLite*, we used the proprietary *TH3* test suite.

To test that our refactorings are behavior preserving, we applied the approach used by Liebig et al. [92]. We used two oracles: (1) the code of our subject systems still compiles; and (2) the results of the test cases of the projects (pre-refactoring and post-refactoring) do not vary. To incorporate variability, we detected the configurations affected by refactorings and test them. Notice that the *brute-force* approach used previously does not scale to these systems. So, we consider only the configurations impacted by refactorings. After running the test cases before and after applying our refactorings in the three systems using the *Morpheus* infrastructure, we found no behavioral changes or implementation problems in our catalog of refactorings.

Table 6.6: Results of testing on *BusyBox*, *OpenSSL*, and *SQLite*.

	R2	**R3**	**R4**	**R5**
BusyBox	15	6	20	4
OpenSSL	11	23	8	114
SQLite	7	4	5	17
Behavioral changes	0	0	0	0

SUMMARY

By performing regression testing in three real-world C systems (BusyBox, OpenSSL, and SQLite), we found no behavioral changes in the catalog of refactorings, improving confidence that the refactorings of our catalog are behavior-preserving.

RQ4. Do the refactorings increase code clones, LOC and preprocessor directives?

We used the catalog to remove 477 undisciplined directives without cloning code. We did not find any block of code clone with at least two lines of code introduced by our refactorings (we used the *Simian* similarity analyser). Notice that we only analyzed the parts of the source code that we modify with our catalog of refactorings. The catalog introduced 0.04% lines of code regarding the total lines of code for all families. Furthermore, the catalog introduced extra directives, which represents 2.10% of the total number of directives of all families.

Table 6.7: Subject characterization

Project	Undisc. Directives	Cloning	LOC	LOC (%)	#ifdefs	#ifdefs (%)
Apache	178	0	+257	+0.18%	+48	+2.21%
Bc	6	0	+6	+0.12%	0	0.00%
Dia	31	0	+59	+0.31%	+13	+4.06%
Expat	31	0	+76	+0.44%	+14	+3.87%
Flex	16	0	+16	+0.09%	0	0.00%
Fvwm	61	0	+115	+0.11%	+46	+3.35%
Ghostscript	87	0	+143	+0.01%	+30	+0.95%
Gnuchess	2	0	+2	+0.02%	0	0.00%
Gzip	19	0	+37	+0.64%	+12	+4.03%
Lighttpd	23	0	+33	+0.08%	+11	+1.18%
Lua	6	0	+18	+0.12%	+6	+3.11%
Mptris	17	0	+39	+0.78%	+11	+3.05%
Total	**477**	**0**	**+801**	**+0.04%**	**+191**	**+2.1%**

SUMMARY

The catalog of refactoring does not introduce code clone as previous refactorings [32; 67; 30], but introduces an insignificant amount of preprocessor conditional directives and lines of code.

6.2.3 Threats to Validity

In this section, we discuss some threats to validity.

Internal validity

We defined our catalog of refactorings based on patterns of undisciplined directives detected in 12 subject systems [50], including *Apache*, *Gzip* and *Lighttpd*. By using the catalog of

refactorings, we removed 477 undisciplined directives in these projects. The catalog is not complete, and variations of our refactorings are necessary to remove undisciplined directives in other systems. However, the catalog of refactorings considers the most frequent patterns of undisciplined directives that we detected in practice.

We asked developers about their preferences using two equivalent code snippets. We did not ask developers about each refactoring individually and considered only one refactoring of each group: (1) a refactoring that introduces local variables; (2) a refactoring that introduces macros; and (3) a refactoring that duplicates a few language tokens. This way, we can only conclude that developers accept our three strategies to resolve undisciplined directives.

Regarding application possibilities in practice, we used an XML-based tool to detect application possibilities. *SrcML*[9] uses heuristics that may fail in code with undisciplined directives. To minimize this threat, we also determined the application possibilities of three projects (*BusyBox*, *Libssh*, and *Libpng*) using *TypeChef* [17], which works soundly in the presence of undisciplined preprocessor directives. *TypeChef* requires a time-consuming setup, though, hindering the analysis of all 63 projects. The numbers of possibilities vary by two percentage points when comparing the results of *TypeChef* and *SrcML*.

External validity

We used a formal model to generate program families with application possibilities for our refactorings. Our model considers only a subset of the C language, though. Thus, we might miss behavioral changes caused by other C constructs that we have not considered. Furthermore, the undisciplined directives that we generate might be different from the ones used in practice. To minimize this threat, we also used three real-world projects, *BusyBox*, *OpenSSL*, and *SQLite*, to test for behavior preservation.

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

Chapter 7

Tool Support: Colligens

In this chapter, we present our tool—named *Colligens*—that automatizes our strategies to detect configuration-related bugs, detects bad smells, and applies our catalog of refactorings to remove undisciplined directives automatically. *Colligens* is an *Eclipse plug-in* written in *Java* and provides an integrated, sampling-based, and variability-aware environment to develop and evolve C program families. *Colligens* is open-source and the tool is available for downloading at our Website.[1]

In Section 7.1, we show the integration of the macro constraints editor of *FeatureIDE* [71] and *Colligens*. Next, we present the three main functionalities of *Colligens*: investigation of configuration-related bugs based on variability-aware analysis in Section 7.2; investigation of configuration-related bugs using sampling-based analysis in Section 7.3; and refactorings to remove undisciplined directives in Section 7.4.

7.1 Macro Constraints Integration

To illustrate how *Colligens* integrates the macro constraints of *FeatureIDE*, we use an example of *Libssh*, as presented in Figure 7.1. By making *TypeChef* aware of the macro constraints, it recognizes that features `HAVE_LIBCRYPTO` and `HAVE_LIBGCRYPT` are alternatives and does not detect any syntax error. In other words, *TypeChef* now knows that the problematic configuration `!HAVE_LIBCRYPTO` and `!HAVE_LIBGCRYPT` is invalid. *Colligens* makes it possible by implementing a mapping between features of the model and

[1] https://sites.google.com/a/ic.ufal.br/colligens/

preprocessor macros. This way, developers do not waste time analyzing bugs in invalid configurations. The explain the other configuration parameters in the following sections.

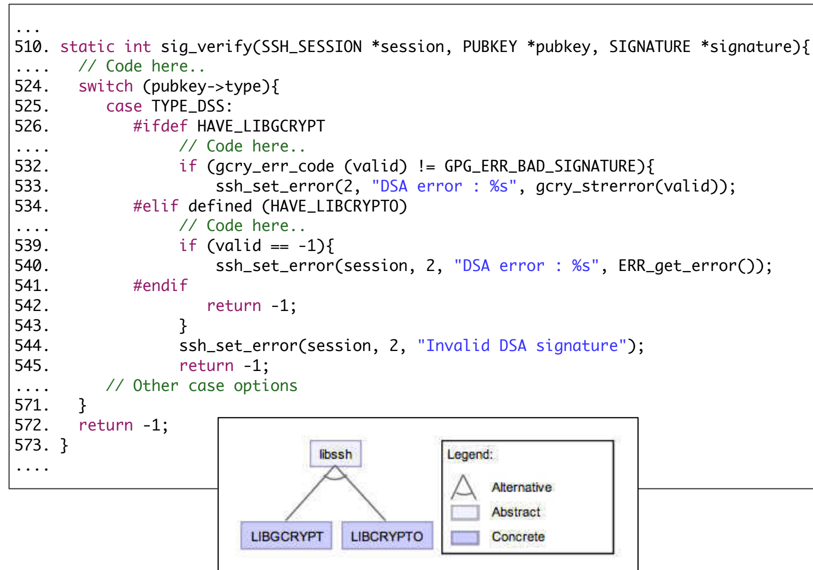


Figure 7.1: Code snippet of *Libssh* and its macro constraints.

In Figure 7.2, we present some configuration parameters of *Colligens*. Notice that there is a configuration parameter to decide whether the tool takes the macro constraints into account. The reason is that many open source projects in C do not provide macro constraint information. So, *Colligens* provides de option of ignoring macros constraints entirely. By setting this configuration, developers will restrict the number of configurations to analyze. This way, *Colligens* considers the macro constraints, and ignores invalid configurations.

7.2 Variability-Aware Analysis

Colligens integrates functionalities of *TypeChef* [17] and *FeatureIDE* [71] to automatize our variability-aware strategy to detect configuration-related bugs. By using our tool, developers can set how *Colligens* performs this investigation of bugs as we can see in Figure 7.2. We explain each configuration parameter with regards to variability-aware analysis next:

- Use `#include` directives: developers can investigate the presence of configuration-related bugs in C program families by considering all `#include` directives;

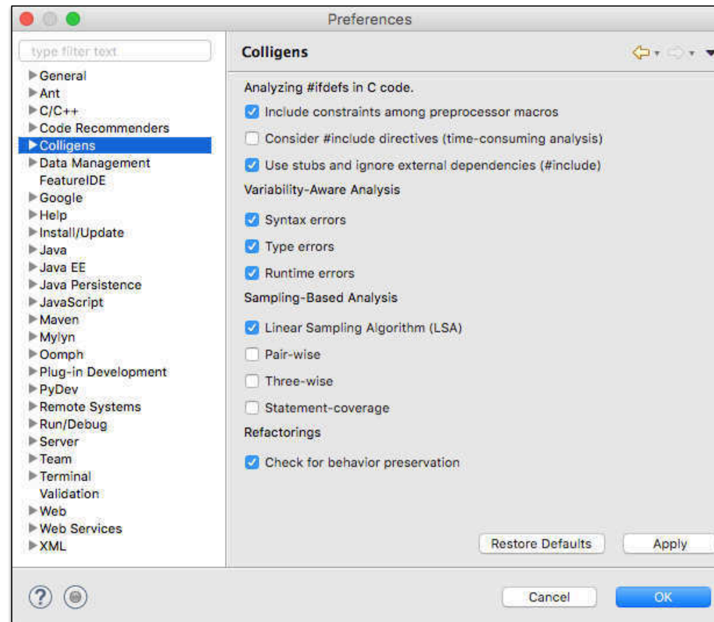


Figure 7.2: *Colligens* view to set configuration parameters.

- Use stubs: developers can investigate the presence of configuration-related bugs in C program families by ignoring `#include` directives, i.e., using our stubs and making the analysis faster;
- Syntax errors: this option allows *Colligens* to execute its analysis to detect configuration-related bugs regarding syntax issues;
- Type errors: this option allows *Colligens* to execute its analysis to detect configuration-related bugs with regards to type issues, such as undeclared variables and functions, and unused variables and functions;
- Runtime errors: this option allows *Colligens* to execute its analysis to detect configuration-related runtime bugs, such as memory and resource leaks, uninitialized variables, and dereferences of null pointers.

After running *Colligens* to investigate configuration-related bugs, the tool presents the bugs detected in a view, as presented in Figure 7.3. By using this view, developers see the files with configuration-related bugs and the problematic configurations, i.e., the con-

figurations with configuration-related bugs. Furthermore, by clicking on the view over a configuration-related bug, developers can reach the exact line of the code with the bug.

Bugs

Description	Resource	Path	Feature configuration
dh.c (2 errors)			
end of input expected (List())	dh.c	/libssh/src/	(def(LIBCRYPTO)|def(LIBCRYPT))
end of input expected (List())	dh.c	/libssh/src/	(!def(LIBCRYPTO)&!def(LIBCRYPT))

Figure 7.3: *Colligens* view to present bugs detected by using variability-aware analysis.

7.3 Sampling-Based Analysis

Colligens also integrates functionalities of the *CppCheck* tool to automatize our strategy to detect configuration-related bugs. When using our tool, developers can investigate the presence of configuration-related bugs in a program family using a sampling-based approach. In this context, *Colligens* preprocesses the source code to generate individual configurations and uses *CppCheck* to check each generated configuration individually. In this analysis, developers can select different sampling algorithms to select configurations to test, such as *LSA*, *pair-wise*, and *statement-coverage*. In Figure 7.2, we can see the *Colligens* view to select sampling algorithms.

After running *Colligens* to investigate configuration-related bugs, the tool lists the bugs detected in a view, as presented in Figure 7.4. Here, developers can also reach the exact line of the source code with the configuration-related bug by clicking on the view over a bug.

Bugs

Msg	Line	Severity	Config	Id
/src/test.c (1 error)				
Memory leak: ptr	11	error	LIBCRYPTO	memleak

Figure 7.4: *Colligens* view to present bugs detected by using sampling.

7.4 Detecting and Removing Bad Smells

Colligens also implements our strategy to detect bad smells and applies our catalog of refactorings automatically, i.e., it removes undisciplined directives (bad smells) using the refactorings presented in Chapter 6. We present a refactoring example using *Colligens* in Figure 7.5. In this refactoring, we select a file with an undisciplined directive and the tool proposes a refactoring to remove the undisciplined directives. Developers can check the refactored source code proposed by *Colligens* before accept it. Otherwise, developers can just cancel it and *Colligens* makes no changes on the source code. In Figure 7.2, we can see the *Colligens* view to select whether the tool check for behavior preservation after applying the refactorings.

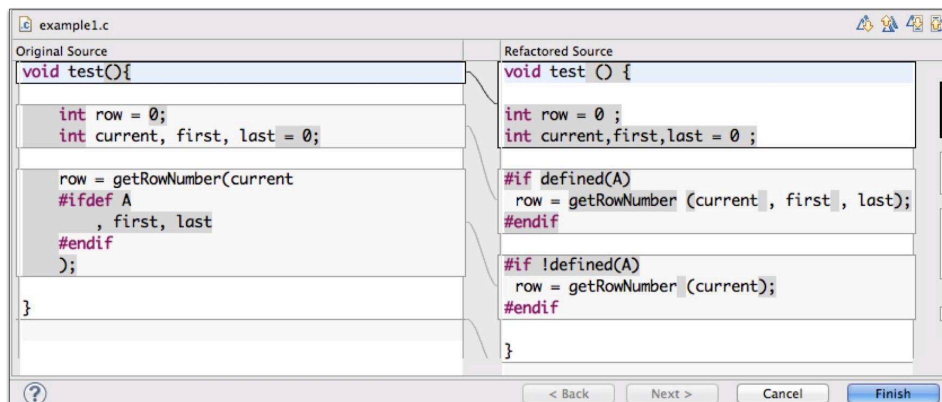


Figure 7.5: *Colligens* view to refactor undisciplined directives.

Chapter 8

Related Work

In this chapter we present the related work. We separate it into six areas directly related to our study. In Section 8.1, we present research that studies the use of the preprocessor. We discuss studies that propose static analysis tools to detect bugs in Section 8.2 and Section 8.3 presents variability-aware parsers. In Section 8.4, we discuss approaches to perform combinatorial testing to detect configuration-related bugs. Then, we present strategies to extract variability information from source code in Section 8.5. Finally, we present research to refactor program families in Section 8.6.

8.1 Analysis of C Preprocessor Usage

Some approaches studied the way developers use the C preprocessor in practice. Ernst et al. [3] presented an empirical study on how the C preprocessor by analyzing 26 packages comprising 1.4 MLOC. They found that most C preprocessor usage follows simple patterns. It also discussed about the undisciplined use of the C preprocessor and its problems, such as that it makes the program more difficult to understand. However, it focused mainly on macro definitions using `#define` directives. In this sense, our work complements the analysis of using the C preprocessor and presents findings about configuration-related bugs in practice.

Baxter and Mehlich proposed DMS, a source-code transformation tool for C and C++ [68]. In a more recent work, these authors used DMS and emphasized the problem of using unstructured directives [13], similar to undisciplined directives. Furthermore, the authors presented examples of configuration-related syntax errors, emphasizing the error prone characteristics of the C preprocessor as discussed in our study.

Liebig et al. [4] analyzed 40 systems, and also suggested that developers can introduce subtle syntax errors when using undisciplined directives. The authors found that the use of undisciplined directives corresponds to 15.6% of the total number of directives. Garrido et al. use the term incomplete as a substitute for undisciplined directives [81; 107; 30].

Others approaches also complemented these studies providing more information about the preprocessor usage. In a previous work, Ribeiro et al. [104] analyzed how often methods with preprocessor directives contain feature dependencies. Liebig et al. [67] proposed and collected some metrics using 40 subject systems to analyze the feature code scattering and tangling when using preprocessor directives.

All prior studies on the C preprocessor that we are aware of were based on conceptual arguments or evidence extracted from software repositories. In our study, we elicited the *perception* of developers by talking to them, and by performing an online survey.

8.2 Static Analysis to Find Bugs

We also find studies proposing tools that perform static analysis to find bugs, such as memory leaks, resource leaks, null dereferences and initialized variables. Torlak et al. [76] present *Tracker*, a tool to identify resource leaks by performing inter-procedural analysis in Java source code. Hovemeyer et al. [127] presents *FindBugs* based on automatic detectors for a variety of bug patterns of Java code as well. Artho and Biere propose *Jlint2*, a tool that performs static analysis in large-scale and multi-threaded Java systems [128].

In the context of the C language, Evans and Larochelle propose the *Splint* tool to detect semantic bugs in C [77; 78]. *Splint* statically checks C programs for security vulnerabilities and coding mistakes. Novark et al. presents a tool, named *Plug*, to detect memory leaks in C and C++ programs [79]. Nethercote and Seward proposes *Valgrind*, an instrumentation framework for building dynamic analysis tools. There are tools implemented, for example, to detect memory management and threading bugs [129].

Other studies have analyzed software repositories by considering bugs already fixed by developers to understand the characteristics of configuration-related bugs [12; 16]. In this context, researchers analyzed configuration-related bugs in configurable systems [19; 130]. They concluded that the majority of configuration-related bugs involve a few macros,

a result similar to ours. Abal et al. [16] analyzed the *Linux Kernel* software repository to study configuration-related bugs. Tartler et al. [20] also performed studies using the tool *Undertaker* [73] to find configuration-related bugs in the *Linux Kernel*. We considered some configuration-related bugs reported by these previous studies in our study to compare the sampling algorithms, as discussed in Chapter 4. By understanding the tradeoffs of sampling algorithms, we can leverage these tools to detect configuration-related bugs, even the ones that do not take variability into account.

8.3 Variability-Aware Analysis

There are some strategies to parse C code in the presence of preprocessor directives. Many approaches [131; 30; 89] applied the strategy of preprocessing or modifying the source code before parsing it. However, this strategy is not interesting to analyze variability since we loose information about the preprocessor directives. Other researchers introduce additional language constructions [132], which are not supported by traditional compilers, hindering their widespread use in practice. In our study, we used a variability-aware parser to consider all variability information, allowing us to detect configuration-related bugs also performing variability-aware analysis.

Kästner et al. [17] proposed a variability-aware parser that analyzes all configurations of a C program family at once. In addition, it performs type checking analysis [133; 134]. In our work, we used *TypeChef* to identify bugs in C program families, i.e., it is the basis of our variability-aware strategy to investigate configuration-related bugs bugs. Furthermore, we also used *TypeChef* to generate abstract syntax tree enhanced with variability information to perform our refactorings. Gazzillo and Grimm [38] proposed another variability-aware parser named *SuperC*. This parser is faster than *TypeChef*, but it does not perform type checking analysis.

Difficulties in setting up these tools and narrow classes of detectable faults limit their applicability and lead to false positives. This is the reason that motivated us to propose the two simplifications (stubs and platform-specific headers), presented in Chapter 5, to make our variability-aware analysis scalable. Furthermore, variability-aware tools work at the preprocessor level, which hinders the reuse of existing bug checkers of traditional C tools,

including *Gcc* and *Clang*. Our sampling-based strategy allows us to reuse these tools by performing sampling.

8.4 Sampling Analysis

Although researchers have proposed approaches to analyze complete configuration spaces in a sound fashion for some classes of defects [71; 17; 72; 38; 18], the vast majority of mature quality-assurance techniques consider only a single configuration at a time, such as *Gcc*, *Clang*, and *Eclipse*. Static-analysis tools operate typically on C code after the C preprocessor has resolved the variability implemented through conditional compilation (e.g., implemented with `#ifdef` directives). To reuse state-of-the-art tools, such as *gcc*, to detect configuration-related bugs, *sampling* is a viable alternative [34; 35; 36; 37; 20] that we used in our research study to detect configuration-related bugs.

Researchers have proposed various strategies to deal with configuration-related bugs. They considered combinatorial testing to check different combinations of configuration options and prioritize test cases [135; 130; 136; 137; 46; 109]. For instance, Nie et al. [90] performed a survey with combinatorial testing approaches. Other researchers used *t-wise* sampling algorithms to cover all t configuration option combinations [34; 35; 36; 37; 45]. Petke et al. [138] compared strategies to generate covering arrays for *t-wise* algorithms, such as simulated annealing and greedy algorithms. Tartler et al. proposed the *statement-coverage* [73] sampling algorithm, and Abal et al. [16] suggested the *one-disabled* algorithm. However, many studies on sampling make assumptions that might not be realistic in practice, such as ignoring constraints among macros. Including constraints, build-system information, and header files is a non-trivial task. Sánchez et al. [139] applied realistic settings and studied the use of non-functional data for test case prioritization. Other researchers applied *t-wise* algorithms with constraints [45; 36; 120; 121], and Grindal et al. [140] studied different constraint handling methods. In our study, we compared different sampling algorithms and the influence of constraints, build-system information, header files, and global analysis on sampling, as discussed in Chapter 4.

Some studies have compared sample-based and variability-aware strategies. Apel et al. [141] developed a model checking tool for product lines and used it to compare sample-based and variability-aware strategies with regard to verification performance and the ability to find defects. Liebig et al. [47] performed studies to detect the strengths and weaknesses of variability-aware and sampling-based analyses. They considered two type of analysis (type checking and liveness analysis) and applied them to a number of subject systems, such as *Busybox* and the *Linux kernel*. Kolesnikov et al. [142] compared variability-aware, feature-based, and product-based type checking. In our study, we performed complimentary analyses regarding sampling algorithms and filled a gap by comparing sampling algorithms considering the influence of assumptions made in previous studies.

8.5 Extracting Variability Information

Others proposed techniques to extract variability information from C program families. Some researches considered the *Linux* kernel in their studies and analyzed its source code files, Kconfig files, and Makefiles [143; 144; 111; 145]. Other researches analyzed the rapid evolution of the *Linux* configurations. The number of features had doubled in the period analyzed [146]. She et al. [147] analyzed operating systems, such as *FreeBSD* and *eCos*. In our work, we decide to contact the developers of the program families to check configuration constraints, i.e., avoiding the effort of gathering information about configuration constraints for each family.

Tartler et al. [106] revealed the presence of zombie configurations in the *Linux Kernel*, i.e., preprocessor macros that cannot be either enabled or disabled at all. Other researches found several inconsistencies in the *Linux* kernel by analyzing source files, Kconfig and makefiles [91; 18]. In our work, we focused only on configuration-related bugs in source code files and their presence in valid configurations. To the best of our knowledge, there is no existing work that investigated the impact of configuration-related bugs considering such a high number of C program families.

8.6 Refactoring Program Families

Opdyke [80] defines refactoring as a behavior-preserving program transformation. To check behavior preservation, an approach with successive compilation and tests is used. Opdyke considers refactorings in object-oriented frameworks, which focuses only on refactorings of a single program. Fowler [33] uses the concept of bad smells, i.e., code with poor quality like methods with many lines of code, several parameters, and duplicated code. He also presents refactorings to object-oriented systems, such as extract method, replace array with object, and pull up method. In other words, structural refactorings for languages like Java. The refactoring of C code is different from refactoring in other languages due to the presence of the C preprocessor. In this context, we have a number of program variants and not a single program. This way, refactoring tools have to consider all possible program variants. In addition, refactorings in C focuses on code inside functions and not on structural refactorings, since C is a structural programming language.

There are some approaches to refactor C code with preprocessor directives. Garrido and Johnson [107] developed the *CRefactory*, a refactoring tool for C program families that considers all possible configurations. Garrido and Johnson also propose a strategy to remove undisciplined directives [30], but it introduces code cloning. Moreover, *CRefactory* focuses on C refactorings such as renaming functions and extracting macros [22]. Our work has a different focus. We propose C refactorings to the directives themselves to remove undisciplined directives without cloning code. Thus, we minimize the problems related to undisciplined directives, such as syntax bugs and code understanding.

Vittek presents *Xrefactory*, a refactoring browser for C source code and discusses certain complications introduced by the CPP [21]. Vittek uses a strategy that preprocesses the code keeping information about the conditional directives and refactoring the code directly. In our work, we perform our refactorings on the abstract syntax trees, which contain variability information, i.e., we do not preprocess the code. Thus, we take into account all the variability information.

Tokuda and Batory also propose a refactoring tool to class diagrams of C++ programs [148]. Their work focuses on refactorings of object-oriented systems. Moreover, their work does not deal with preprocessor directives. This way, applying a simple refactoring, such as a function renaming, may introduce behavioral changes, since the tool does not change the

function name in all possible configurations. Basically, this work refactors only a single C++ program. In our study, we refactor a C program family, and not a single program.

In a recent study, Liebig et al. [92] proposed a variability-aware refactoring approach, which preserves the behavior of all variants of a configurable system. Liebig et al. uses variability-aware analysis, which considers all possible configurations of the source code at the same time. Their study keeps all variability information, different from strategies that preprocess or modify the source code before parsing it [89; 131]. Liebig at el. showed the applicability and scalability of their approach by implementing a sound refactoring engine (*Morpheus*) and by performing refactorings (Extract Function, Rename, and Inline Function) in three real-world projects: *BusyBox*, *OpenSSL*, and *SQLite*. Liebig et al. also provided evidence for the correctness of the refactorings implemented by running the original test cases of the projects before and after applying the refactorings. Our work also use *TypeChef* as Liebig et al., but we focus on refactorings to remove undisciplined directives, different from all studies discussed.

Other studies investigate the refactorings of conditional directives into aspects. Adams et al. [28] propose a model and analyze the feasibility of refactoring `#ifdef` to aspects, but it does not implement any tool to perform the refactorings automatically. According to their work, it is possible to refactor 99% of the conditional compilation into aspects. Lohmann et al. [15] refactor the *eCos* operating system kernel using AspectC++, an Aspect-Oriented Programming (AOP) extension to the C++ language, and analyze the runtime and memory costs of aspects. Our work also focuses on refactorings of preprocessor directives, but we refactor the directives without introducing another variability implementation mechanism like aspects.

Borba et al. [149] define a theory to refactor software families. In their work, they use specific artifacts, such as feature models and configuration knowledges, and propose a theory to detect when a product line refactors another. Furthermore, it defines a theory using a formal specification language and proves some compositionality properties of the theory. Alves et al. [150] extend the theory of refactorings in software families with refactorings based on feature models. We used the theory of Borba et al. to verify behavior preservation in our refactorings. The theory specifies refactorings that involve changes in the source code, feature model, and configuration knowledge. As our refactorings do not change the

feature model and the configuration knowledge of the program family, we focused only on refactorings of the source code.

In another study, Ferreira et at. [151] present an implementation of this software family theory. It proposes tools to evaluate if an SPL transformation preserves behavior. These tools use test cases to avoid behavioral changes in refactorings. They are based on *SafeRefactor*, which creates test cases automatically to increase confidence that a transformation preserves behavior [152]. In their study, they define four strategies to identify behavioral changes using dynamic analysis. Thus, *SafeRefactor* generates test cases and runs the same test suite in the original and refactored code to detect bugs in refactoring engines. In our work, we used a similar approach as in *SafeRefactor*, but extending it the context of program families, to increase confidence that our refactorings are behavior-preserving.

Recent study proposes a technique to test C refactoring engines [153]. It uses a program generator (*CDolly*) and a test case generator to detect bugs in refactoring engines, strategy similar to *SafeRefactor*. By analyzing refactoring engines, such as *Eclipse*, it finds some bugs, including bugs related to preprocessor directives. In our study, we have a different focus, i.e., we refactor the source code using our catalogue. However, the technique proposed do not support to test our refactorings, since *CDolly* does not generate program with different types of preprocessor directives as we find in real word.

Other studies propose strategies to verify if all program variants are well-formed. They used strategies to verify type errors and missing dependencies by using feature models, SAT solvers [154; 155; 156], and configuration knowledges, i.e., the safe composition problem. However, existing C program families, such as *Apache*, *Dia* and *Gzip*, do not have some artifacts that these studies uses, such as feature models and configuration knowledge. In our study, our strategy to apply refactorings does not strictly require feature models and configuration knowledge, making it possible to apply refactorings in real-world C systems and to check behavior preservation.

Chapter 9

Concluding Remarks

In this work, we propose an approach to safely evolve configuration-related program families in C and performed interviews to consider the perception of developers. To support defective evolution, we presented strategies to detect configuration-related bugs using sampling and variability-aware analysis. To support perfective evolution and to remove bad smells in preprocessor directives, we proposed 14 refactorings to remove undisciplined directives in C program families.

We evaluated the proposed strategies using a corpus of 40 subject systems to investigate configuration-related bugs. The results of our study, including interviews with 40 developers and a survey among 202 participants, show that configuration-related bugs occur in practice and developers perceive these bugs as a problem, giving relevance to the problem we address in this thesis. According to the perception of developers, the use of undisciplined directives is also problematic, which motivated our catalog of refactorings.

We evaluated our catalog regarding frequency of application possibilities in practice, opinion of developers, behavior preservation, and introduction of code clone. We found 5670 application possibilities for our refactorings in 63 real-world projects and our survey show that the majority of developers prefer to use our refactoring, disciplined code instead of undisciplined directives. We submitted 28 patches to convert undisciplined into disciplined directives and developers accepted 21 (75%). To verify that our refactorings are behavior preserving, we applied differential testing to more than 36 thousand programs generated automatically using a formal model as well as in three real-world projects. Based on the results of our study, we concluded that our refactorings do not clone code, different from previous studies.

To support developers when implementing C program families, we presented *Colligens*, a tool capable of detecting different types of configuration-related bugs, including syntax problems, memory leaks, resource leaks, null dereferences, and uninitialized variables. In addition, our tool applies our refactorings automatically. By using *Colligens*, developers gain the benefits of a sampling-based and variability-aware environment to safely evolve configuration-related C program families.

9.1 Review of the Contributions

By performing an interview study to understand how developers perceive the C preprocessor and complimentary studies (survey, literature review, and repository mining) to cross-validate and to quantify the results [29], we found that:

- Developers are aware of the criticism the C preprocessor receives, but still use it in the following situations: (1) supporting portability, (2) supporting variability, (3) providing code optimizations, (4) supporting code evolution, and (5) overcoming limitations of the C language;
- Developers do not see any current technologies that can entirely replace the C preprocessor. However, some developers routinely use alternate coding styles such as dividing functionality into separate files or functions (preferred by 60%) and using run-time checks instead of #ifdef checks (preferred by 19 %) to avoid preprocessor directives;
- Developers face three configuration-related problems: (1) configuration-related bugs (do not appear often, but are perceived as more critical than other bugs), (2) combinatorial testing (conditional directives increase number of configurations to test), and (3) code comprehension (due to the cluttering of #ifdefs and C statements, and the deep nesting of #ifdefs);
- The majority of developers agree that the use of undisciplined directives influences code understanding, maintainability, and error proneness negatively. However, there are cases where developers use undisciplined annotations to avoid code clones and compiler warnings.

We performed a comparison of 10 sampling algorithms for program families regarding effort and bug-detection capability [43], and we proposed the *Linear Sampling Algorithm (LSA)*, guiding developers to perform combinatorial testing. We found that:

- All 10 algorithms analyzed are able to detect at least 66% of the configuration-related bugs considered in our study; *most-enabled-disabled*, *pair-wise* and *statement-coverage* are the most efficient algorithms;
- Some combinations of sampling algorithms provide an useful balance between sample size and bug-detection capabilities, such as the combination of *most-enabled-disabled*, *one-enabled*, and *one-disabled*, which we proposed as the *linear sampling algorithm*;
- When considering constraints to perform sampling, we substantially reduce false positives, but high costs for generating sample sets; it is infeasible for three-wise and higher at large scale;
- Using a global analysis when sampling configurations, we can potentially detect non-modular bugs that span multiple files; it causes an explosion in the number of considered preprocessor macros that leads to large sample sets; too large for *t-wise* and statement-coverage;
- When incorporating header files during sampling, there is a potential to detect additional bugs from header files; but a difficult setup and much larger sample sets (if feasible at all);
- When including build-system information to select configurations, the analysis considers a few more macros, but no significant changes.

We performed empirical studies to quantify and investigate configuration-related bugs in C program families. We considered bugs of different types, including memory and resource leaks, undeclared functions, uninitialized variables, syntax errors, and dereferences of null pointers [12; 39; 49]. To perform these empirical studies considering several real-world projects, we proposed strategies to detect configuration-related bugs in C program families using sampling and variability-aware analysis [12; 39; 48]. Based on the results of our studies, we concluded that:

- Developers face configuration-related bugs in practice as frequent as they face bugs that appear in all configurations. We found configuration-related bugs of all types considered in our empirical studies;

- Configuration-related bugs remain almost three times longer in the source code, on the average, than bugs that appear in all configurations. The variability of program families hinders the detection of even simple configuration-related bugs, such as syntax errors;

- The majority of configuration-related bugs (more than 89%) detected involve two or less preprocessor macros. Our results support the effectiveness of sampling algorithms to detect configuration-related bugs.

We proposed a catalog of refactorings to remove undisciplined directives in C program families without code cloning, leaving the source code less conducive to introduce configuration-related bugs and improving code readability [50; 12]. The *Colligens* tool applies our catalogue of refactorings automatically [51]. When evaluating the catalog of refactorings, we found:

- 5670 application possibilities for the refactorings in practice considering real-world systems of different sizes and from various domains. There are places to apply the refactorings in almost all systems (97%) analyzed in this study, showing that developers still use undisciplined directives in practice;

- Most developers preferred to use the refactored (i.e., disciplined) code instead of using the preprocessor in undisciplined ways;

- Developers support the idea of converting undisciplined into disciplined directives. Developers accepted 21 (75%) out of the 28 patches submitted, showing more evidence that they prefer to use disciplined directives;

- By performing differential testing in the generated program families to verify behavior preservation, we found and fixed a few behavioral changes introduced by our refactorings and a number of problems in the implementation of our catalog, the majority related to unspecified behavior in the C language. This way, we improved confidence that the refactorings are behavior-preserving.

- By performing differential testing in three real-world C systems (*BusyBox*, *OpenSSL*, and *SQLite*), we found no behavioral changes in the catalog of refactorings, improving confidence that the refactorings of our catalog are behavior-preserving.

- The catalog of refactoring does not introduce code clone as previous refactorings [32; 4; 30], but introduces a minimal amount of preprocessor conditional directives and lines of code.

9.2 Future Work

We proposed strategies to detect configuration-related bugs based on sampling and variability-aware analysis. However, there are opportunities to implement new bug checkers using a variability-aware approach, such as checkers to detect memory and resource leaks, uninitialized variables, and dereferences of null pointers. Notice that we detected these types of bugs using sampling and *Cppcheck*. There are also possibilities to apply our sampling-based strategy using other static analysis tools, such as *Clang* and *Gcc*. Furthermore, we can extend the strategies to detect configuration-related bugs by performing other kinds of analysis, such as dynamic analysis and symbolic execution.

Regarding the comparison of sampling algorithms for configurable systems, there are possibilities to extend our comparative study to configurable systems that implement variability using other mechanisms, such as aspect-oriented programming [28] and delta-oriented programming [157], as we have considered only conditional compilation. Likewise, there are possibilities to extend the evaluation of our proposed sampling algorithm (i.e., *LSA*) as well to consider other types of variability implement mechanisms. Furthermore, one can perform more studies comparing sampling-based and variability-aware strategies with regards to effort, bug-detection capabilities, and time of analysis.

We have proposed refactorings to remove undisciplined directives, but many possibilities remain to explore. There are possibilities to work on refactorings that remove preprocessor directives entirely from source files (.c), moving them all to header files (.h). This way, we refactor the source code and apply a similar strategy to evaluate these new refactorings, that is, submitting patches to C projects, and performing interviews and surveys to evaluate the other catalogs of refactorings.

Regarding our evaluation of behavior preservation, there are opportunities to extend the evaluation to a higher number of real-world systems. One can extend our technique to improve confidence in behavior preservation to all types of refactorings in C program families. Thus, we can evaluate our refactorings, such as the ones implemented by Liebig et al. [47], to improve behavior preservation.

There are also possibilities to extend *Colligens*. We concluded that developers support the idea of converting undisciplined into disciplined directives. However, it is also clear that some developers prefer to refactor the code when making other necessary changes in the code, for example, to fix bugs. This way, our results show that we need better integration between refactoring tools and software repositories. Thus, it might be appropriate to perform more research studies with regards to the integration of *Colligens* and *GitHub*, for example, to suggest refactorings based on events of the *GitHub* pull request infrastructure.

Bibliography

[1] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the International Conference on Software Engineering*, pages 311–320. ACM, 2008.

[2] Henry Spencer. Ifdef considered harmful, or portability experience with C news. In *USENIX Annual Technical Conference*, pages 185–197, 1992.

[3] Michael Ernst, Greg Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, 2002.

[4] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 191–202. ACM, 2011.

[5] David Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:33–40, 1976.

[6] Troy Pearse and Paul Oman. Experiences developing and maintaining software in a multi-platform environment. In *International Conference on Software Maintenance*, pages 270–277. IEEE, 1997.

[7] Jean-Marie Favre. Understanding-in-the-large. In *Proceedings of the International Workshop on Program Comprehension*, pages 29–38, Mar 1997.

[8] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Laguë. C/C++ conditional compilation analysis using symbolic execution. In *Proceeding of the International Conference on Software Maintenance*, pages 196–206. IEEE, 2000.

[9] Michalis Anastasopoules and Critina Gacek. Implementing product line variabilities. In *Proceedings of Symposium on Software Reusability*, pages 109–117. ACM, 2001.

[10] Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, 2007.

[11] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *International Conference on Software Engineering*. IEEE, 1994.

[12] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating Preprocessor-Based Syntax Errors. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, pages 75–84. ACM, 2013.

[13] Ira Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 281–290. IEEE, 2001.

[14] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. *Electronic Communications of the European Association for the Study of Science and Technology*, 2008.

[15] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *ACM European Conference on Computer Systems*, pages 191–204. ACM, 2006.

[16] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the International Conference on Automated Software Engineering*, pages 421–432. IEEE/ACM, 2014.

[17] Christian Kästner, Paolo Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the ACM SIGPLAN Object-Oriented Programming Systems Languages and Applications*, pages 805–824. ACM, 2011.

[18] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer*, 14(5):531–551, 2012.

[19] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *IEEE International Symposium on Software Reliability Engineering*, pages 90–99, 2011.

[20] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *2014 USENIX Annual Technical Conference*, 2014.

[21] Marian Vittek. Refactoring browser with preprocessor. In *European Conference on Software Maintenance and Reengineering*, pages 101–110. IEEE, 2003.

[22] Alejandra Garrido and Ralph Johnson. Embracing the C preprocessor during refactoring. *Journal of Software: Evolution and Process*, page 1285?1304, 2013.

[23] Bill McCloskey and Eric Brewer. Astec: A new approach to refactoring C. *SIGSOFT Software Engineering Notes*, 30(5):21–30, 2005.

[24] Christian Kästner, Sven Apel, and Martin Kuhlemann. A model of refactoring physically and virtually separated features. In *Proceedings of the Conference on Generative Programming and Component Engineering*, pages 157–166. ACM, 2009.

[25] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *International conference on Generative programming and component engineering*, pages 191–200. ACM, 2006.

[26] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of ACM SIGPLAN Conference on Prog. Language Design and Implem.*, pages 156–165. ACM, 1993.

[27] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: A pragmatic approach to software product line imple-

mentation. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2010.

[28] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *ACM International Conference on Aspect-Oriented Software Development*, pages 243–254. ACM, 2009.

[29] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *European Conference on Object-Oriented Programming*, pages 999–1022. Schloss Dagstuhl, 2015.

[30] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a C program. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 379–388. IEEE, 2005.

[31] Robert C. Seacord. *The: 98 Rules for Developing Safe, Reliable, and Secure Systems*. CERT C Coding Standard, 2014.

[32] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the discipline of preprocessor annotations matter?: a controlled experiment. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, pages 65–74, 2013.

[33] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[34] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Lines: Going Beyond*, pages 196–210. Springer, 2010.

[35] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for product lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 459–468. IEEE, 2010.

[36] Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of International Software Product Line Conference - Volume 1*, pages 46–55, 2012.

[37] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the International Software Product Line Conference*, pages 227–235, 2013.

[38] Paul Gazzillo and Robert Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the Programming Language Design and Implementation*, pages 323–334. ACM, 2012.

[39] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, pages 35–44. ACM, 2015.

[40] Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the International Conference on Software Engineering*, pages 989–1000, 2014.

[41] Brian W. Kernighan. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[42] Lowell Jay Arthur. *Software Evolution: The Software Maintenance Challenge*. Wiley-Interscience, 1988.

[43] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of International Conference on Software Engineering*, 2016.

[44] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N. Kacker, and Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 2008.

[45] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[46] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2008.

[47] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[48] Flávio Medeiros. An approach to safely evolve program families in c. In *Companion Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity*, pages 25–27. ACM, 2014.

[49] Flávio Medeiros. Safely evolving configurable systems. In *Companion Proceedings of the International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 85–86. ACM, 2015.

[50] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. A catalogue of refactorings to remove incomplete annotations. *Journal of Universal Computer Science*, pages 746–771, January 2014.

[51] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. Colligens: A tool to support the development of preprocessor-based software product lines in c. In *In Tool Section of the Brazilian Congress on Software*, 2013.

[52] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[53] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[54] K. C. Kang, Jaejoon Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[55] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615?636, September 2010.

[56] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *Proceedings of the International Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126, New York, NY, USA, 2012. ACM.

[57] Elder Cirilo, Ingrid Nunes, Alessandro Garcia, and Carlos J.P. de Lucena. Configuration knowledge of software product lines: A comprehensibility study. In *Proceedings of the International Workshop on Variability & Composition*, pages 1–5. ACM, 2011.

[58] Leopoldo Teixeira, Paulo Borba, and Rohit Gheyi. Safe composition of configuration knowledge-based software product lines. *Journal of Systems and Software*, 86(4):263–272, 2013.

[59] P. Clements. Being proactive pays off. *IEEE Software*, 19(4), Jul 2002.

[60] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., 2004.

[61] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., 2007.

[62] Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proceedings of the International Conference on Software Maintenance*, pages 98–107. IEEE, 2012.

[63] Christopher A. Mennie and Charles L. A. Clarke. Giving meaning to macros. *International Conference on Program Comprehension*, pages 79–85, 2004.

[64] Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *Software Engineering, IEEE Transactions on*, 29(11):1019–1030, 2003.

[65] A. Cox and C. Clarke. Relocating XML elements from preprocessed to unprocessed code. In *Proceedings of the International Workshop on Program Comprehension*, pages 229–238. IEEE, 2002.

[66] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software Practice and Experience*, 35:705–754, 2005.

[67] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the International Conference on Software Engineering*, pages 105–114. ACM, 2010.

[68] Ira Baxter. Design maintenance systems. *Communication of the ACM*, 35(4), 1992.

[69] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990.

[70] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. Partial preprocessing c code for variability analysis. In *Proceedings of the Workshop on Variability Modeling of Software-Intensive Systems*, pages 127–136, New York, NY, USA, 2011. ACM.

[71] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, Jan 2014.

[72] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 773–792. ACM, 2012.

[73] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the Workshop on Programming Languages and Operating Systems*, pages 2:1–2:5. ACM, 2011.

[74] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[75] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.

[76] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 535–544. ACM, 2010.

[77] David Evans. Static detection of dynamic memory errors. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 44–53. ACM, 1996.

[78] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Conference on USENIX Security Symposium*. USENIX Association, 2001.

[79] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the Conference on Object-oriented Programming Systems Languages and Applications*, pages 109–126. ACM, 2008.

[80] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[81] Alejandra Garrido and Ralph Johnson. Challenges of refactoring C programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 6–14, 2002.

[82] Michael Platoff, Michael Wagner, and Joseph Camaratta. An integrated program representation and toolkit for the maintenance of c programs. In *Proceedings of the 6th International Conference on Software Maintenance*, pages 129–137. IEEE, 1991.

[83] László Vidács and Árpád Beszédes. Opening up the c/c++ preprocessor black box. In Pekka Kilpeläinen and Niina Päivinen, editors, *Proceedings of the Symposium on Programming Languages and Software Tools*. University of Kuopio, Department of Computer Science, 2003.

[84] Doug Gregor. A module system for the C family, 11 2012. Remarks by Doug Gregor at The sixth general meeting of LLVM Developers and Users.

[85] Matthew Flatt. Composable and compilable macros: You want it when? In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 72–83. ACM, 2002.

[86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 151–161. ACM, 1986.

[87] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 31–40. ACM, 2002.

[88] Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *IEEE International Conference on Software Maintenance*, pages 98–107. IEEE, 2012.

[89] Yoann Padioleau. Parsing C/C++ code without pre-processing. In *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 109–125. Springer Berlin Heidelberg, 2009.

[90] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2), February 2011.

[91] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the 6th Computer Systems*, pages 47–60. ACM, 2011.

[92] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the International Conference on Software Engineering*, pages 380–391. ACM, 2015.

[93] Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. A foundation for refactoring C with macros. In *Proceeding of the International Symposium on the Foundations of Software Engineering*, pages 75–85. ACM, 2014.

[94] Federico Tomassetti and Daniel Ratiu. Extracting variability from c and lifting it to mbeddr. In *International Workshop on Reverse Variability Engineering*, 2013.

[95] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, December 2011.

[96] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study (Artifact). *Dagstuhl Artifacts Series*, 1(1):1–32, 2015.

[97] JulietM Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1), 1990.

[98] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[99] Steinar Kvale. *InterViews: An Introduction to Qualitative Research Interviewing*. SAGE Publications, 1996.

[100] Uwe Flick. *An Introduction to Qualitative Research*. SAGE Publications, 2014.

[101] M. Greiler, A. van Deursen, and Margrete-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the International Conference on Software Engineering*, pages 244–254, 2012.

[102] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the International Conference on Software Engineering*. ACM, 2014.

[103] Don A. Dillman, Jolene D. Smyth, and Leah Melani Christian. *Internet, Phone, Mail, and Mixed-Mode Surveys: The Tailored Design Method*. Wiley, 2014.

[104] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the Generative Programming and Component Engineering*, pages 23–32. ACM, 2011.

[105] Sarah Nadi and Ric Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8), 2014.

[106] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: finding zombie features in the linux kernel. In *Proceedings of the Feature-Oriented Software Development*, pages 81–86, 2009.

[107] Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. In *Proceedings of the Automated Software Engineering*, pages 323–326. IEEE, 2003.

[108] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *International Software Product Line Conference*, pages 91–100. ACM, 2013.

[109] C. Yilmaz, M.B. Cohen, and A.A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006.

[110] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. In *Proceedings of the International Symposium on Software Evolution*, 2007.

[111] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robust approach for variability extraction from the linux build system. In *Proceedings of the Software Product Line Conference*, pages 21–30. ACM, 2012.

[112] Sarah Nadi and Richard C. Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730?746, 2013.

[113] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-condition simplification in highly configurable systems. In *Proceedings of the International Conference on Software Engineering*, pages 178–188. IEEE, 2015.

[114] Massimiliano Caramia and Paolo Dell'Olmo. *Multi-objective Management in Freight Logistics: Increasing Capacity, Service Level and Safety with Optimization Algorithms*, chapter Multi-objective Optimization, pages 11–36. Springer, 2008.

[115] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *Proceedings of the 4th Variability Modeling of Software-intensive Systems*, 2010.

[116] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proceedings of the International Conference on Automated Software Engineering*, pages 73–82. ACM, 2010.

[117] Sarah Nadi, Thorsten Berger, Kästner, and Krzysztof Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering*, 41(8):820–841, 2015.

[118] Supratik Chakraborty, DanielJ. Fremont, KuldeepS. Meel, SanjitA. Seshia, and MosheY. Vardi. On parallel scalable uniform sat witness generation. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015.

[119] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the International Conference on Software Engineering*, pages 517–528. ACM, 2015.

[120] B.J. Garvin, M.B. Cohen, and M.B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 13–22. IEEE, 2009.

[121] M.N. Borazjany, Linbin Yu, Yu Lei, R. Kacker, and R. Kuhn. Combinatorial testing of acts: A case study. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2012.

[122] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at Google. In *Proceedings of the International Workshop on Managing Technical Debt*, 2012.

[123] A. Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and T.N. Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the International Conference on Software Engineering*, pages 650–660, 2012.

[124] Liming Zhu, Donna Xu, Xiwei Sherry Xu, An Binh Tran, Ingo Weber, and Len Bass. Challenges in practicing high frequency releases in cloud environments. In *Proceedings of the International Workshop on Release Engineering*, 2014.

[125] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[126] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. Scaling testing of refactoring engines. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 371–380. IEEE, 2014.

[127] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[128] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings of the Australian Conference on Software Engineering*, pages 68–70. IEEE, 2001.

[129] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[130] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[131] S. Somé and T. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional. In *Proceedings of the International Workshop on Program Comprehension*, pages 118–120, 1998.

[132] Bill McCloskey and Eric A. Brewer. ASTEC: a new approach to refactoring c. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, 2005.

[133] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: toward type checking #ifdef variability in C. In *Proceedings of the Feature-Oriented Software Development*, pages 25–32. ACM, 2010.

[134] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39, 2012.

[135] David Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[136] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering*, pages 38–48. IEEE, 2003.

[137] S. Sampath, R.C. Bryce, G. Viswanath, V. Kandimalla, and A.G. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 141–150, 2008.

[138] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *Transaction on Software Engineering*, PP(99), 2015.

[139] Ana B. Sánchez, Sergio Segura, José A. Parejo, and Antonio Ruiz-Cortés. Variability testing in the wild: The drupal case study. *Software and Systems Modeling*, 14(52), 2015.

[140] M. Grindal, J. Offutt, and J. Mellin. Handling constraints in the input space when using combination strategies for software testing. Technical Report TR-06-001, University of Skövde, 2006.

[141] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering*, pages 482–491. IEEE, 2013.

[142] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, pages 115–124. ACM, 2013.

[143] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the 9th Generative Programming and Component Engineering*, pages 33–42. ACM, 2010.

[144] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Efficient synthesis of feature models. In *Proceedings of the Software Product Line Conference*, pages 106–115. ACM, 2012.

[145] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wąsowski. Towards a catalog of variability evolution patterns: The linux kernel case. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 62–69. ACM, 2012.

[146] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the linux kernel variability model. In *Proceedings of the Software Product Line Conference*, pages 136–150. Springer-Verlag, 2010.

[147] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the International Conference on Software Engineering*, pages 461–470. ACM, 2011.

[148] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, pages 174–181, 2001.

[149] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. In *Proceedings of the International Colloquium Conference on Theoretical Aspects of Computing*, pages 15–43. Springer-Verlag, 2010.

[150] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 201–210. ACM, 2006.

[151] Felype Ferreira, Paulo Borba, Gustavo Soares, and Rohit Gheyi. Making software product line evolution safer. In *Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse*, pages 21–30. IEEE, 2012.

[152] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27:52–57, July 2010.

[153] Gustavo Wagner Diniz Mendes. Uma abordagem para testar implementações de refatoramentos estruturais e comportamentais de programas C. Master's thesis, Universidade Federal de Campina Grande, 2014.

[154] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Eng.*, 17(3):251–300, September 2010.

[155] Don Batory and Sahil Thaker. Safe composition of product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 95–104, 2007.

[156] Benjamin Delaware, William R. Cook, and Don S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In *Proceedings of the Symposium on the Foundations of Software Engineering*, pages 243–252. ACM, 2009.

[157] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the International Conference on Software Product Lines*, pages 77–91. Springer, 2010.

[158] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, pages 730–733. ACM, 2000.

Appendix A

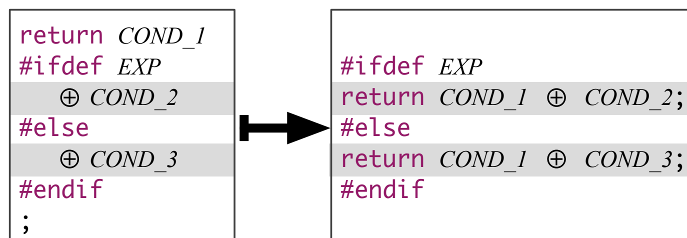
The Complete Catalog of Refactorings

In this appendix, we present the complete catalog of refactorings to remove undisciplined directives, including some refactoring variations that we omitted in Chapter 6. Overall, we present 14 refactorings grouped into four category: *single statements*, *conditions*, *wrappers*, and *comma-separated elements*.

Single Statements

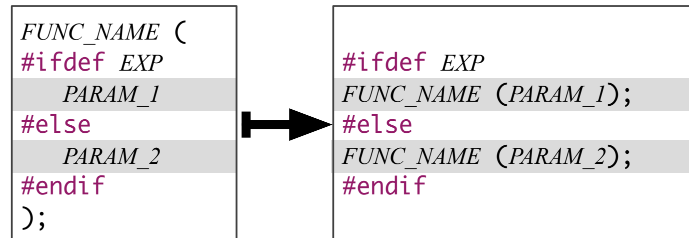
A single statement contains no compound blocks. It includes three kinds of statements: `return` statements, function calls, and variable initializations. We define one refactoring to each kind of statement, as we show in what follows. In Refactoring 1 (a), we present our refactoring to resolve undisciplined preprocessor usage in `return` statements. In this refactoring, we duplicate language tokens to encompass with preprocessor directives entire statements only. Notice that we duplicate the token *COND_1* to make the preprocessor directive disciplined.

Refactoring 1 (a) ⟨undisciplined returns⟩



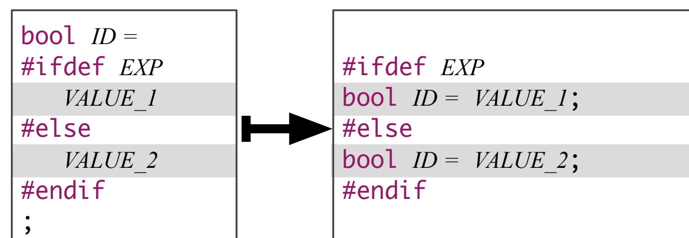
In Refactoring 1 (b), we present our refactoring to remove undisciplined directives in function calls. Likewise, we duplicate language tokens (i.e., *FUNC_NAME*) to encompass with preprocessor directives the entire function call.

Refactoring 1 (b) ⟨undisciplined function calls⟩



Refactoring 3 (c) presents the last variation of undisciplined single statements to remove undisciplined directives in variable attributions. Again, we duplicate language tokens (i.e., *ID*) to encompass with preprocessor directives the entire variable attribution.

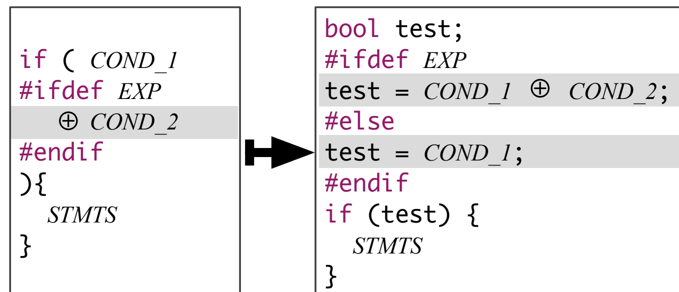
Refactoring 1 (c) ⟨undisciplined variable attributions⟩



Conditions

To resolve undisciplined preprocessor directives surrounding boolean expressions used in `if` statements, we propose Refactoring 2 (a). In this refactoring, we use an extra variable to preserve the statement's conditions. In this sense, we define a precondition that the code is not using the specific identifier (`test`), as we cannot define variables with the same identifier in the same scope.

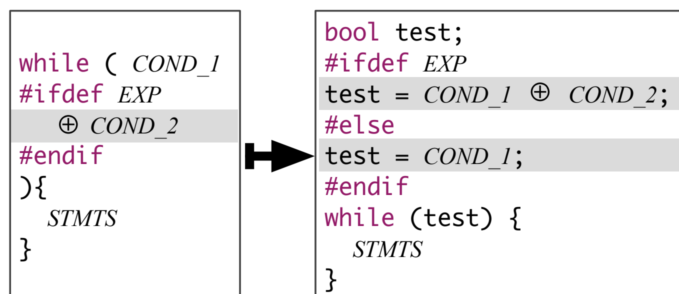
Refactoring 2 (a) ⟨undisciplined `if` conditions⟩



(→) `test` is not used in the code

We refactor `while` statements with undisciplined conditions using a similar refactoring, as presented in Refactoring 2 (b). Here, we also use the local variable to preserve the while statement conditions and the precondition to avoid compilation errors.

Refactoring 2 (b) ⟨undisciplined `while` conditions⟩

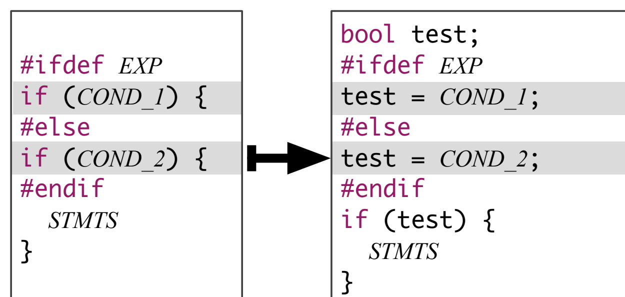


(→) `test` is not used in the code

Wrappers

In Refactoring 3 (a), we target another case of undisciplined preprocessor directive in alternative statements. In this refactoring, we present an `if` statement. Here, we also need an extra program variable to keep the statement's condition. Notice that `test` receives the evaluation of *COND_1* or *COND_2* depending on whether we define macro *EXP* or not. Likewise, we define a precondition that `test` is not used in the code to avoid possible compilation errors.

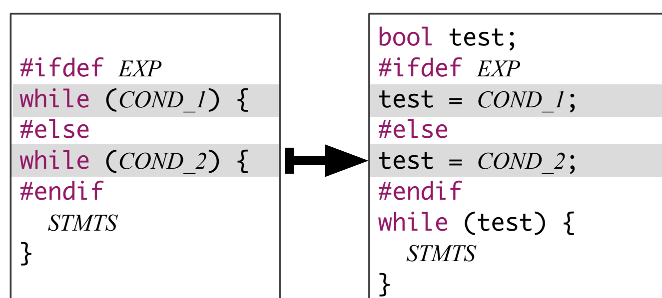
Refactoring 3 (a) ⟨alternative `if` statements⟩



(→) `test` is not used in the code

In Refactoring 3 (b), we present our refactoring to remove undisciplined preprocessor directives in alternative `while` statements. Again, we used a similar refactoring to remove the undisciplined directive in `while` statements, also needing to introduce a local variable to preserve the statement's condition, and a precondition to avoid compilation errors.

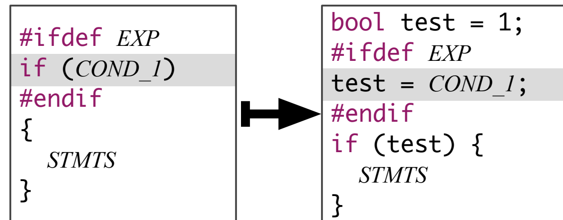
Refactoring 3 (b) ⟨alternative `while` statements⟩



(→) `test` is not used in the code

In Refactoring 4 (a), we present a refactoring to remove `if` wrappers. This refactoring also uses variable `test` to preserve the statement's condition and the precondition with the purpose of disciplining the preprocessor directive.

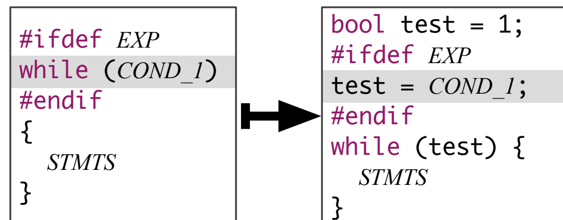
Refactoring 4 (a) ⟨if wrapper⟩



(→) `test` is not used in the code

In Refactoring 4 (b), we present a similar refactoring to remove undisciplined directives in `while` wrappers.

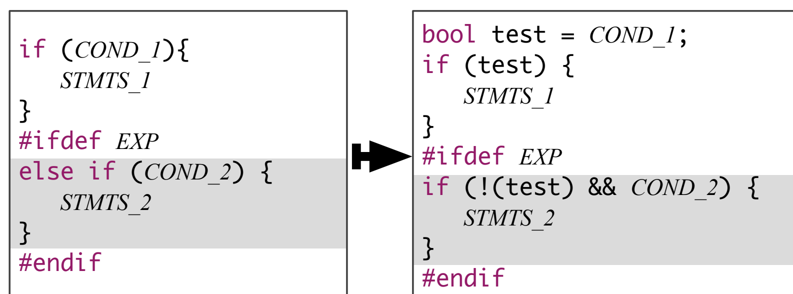
Refactoring 4 (b) ⟨while wrapper⟩



(→) `test` is not used in the code

Refactoring 4 (c) targets a directive surrounding an `else-if` statement. To resolve the undisciplined usage of the preprocessor, we use an extra variable to keep the statement's condition as well as the precondition.

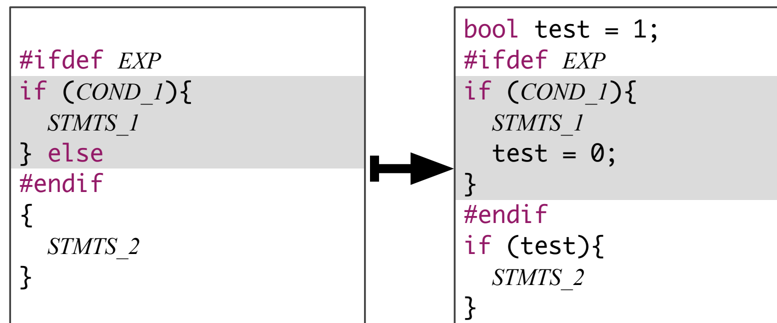
Refactoring 4 (c) ⟨`else-if` wrappers⟩



(→) `test` is not used in the code.

In Refactoring 4 (d), we define a refactoring to remove `if` statements ending with an `else` statement. In this case, we replace the `else` by another `if` statement to resolve the undisciplined usage of the preprocessor. In this refactoring, variable `test` works like a flag to avoid executing *STMTS_2* when macro *EXP* is disabled.

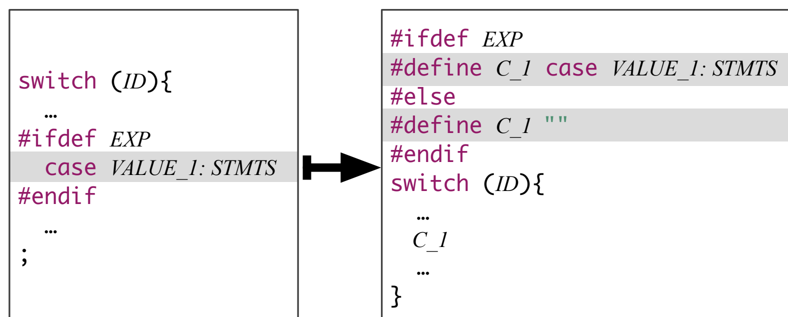
Refactoring 4 (d) ⟨`if` statements with an `else`⟩



(→) `test` is not used in the code.

In Refactoring 4 (e), we show a refactoring to remove `case` wrappers. Here, we use an additional macro to define the `case` statement. Despite we can define the same macro several times, we set a precondition that the code does not define macro `C_1`. If we change a macro definition that the original code is already using, we may add behavioral changes. Using this strategy, we modify the source code locally without global impact.

Refactoring 4 (e) ⟨case wrappers⟩



(→) `CASE1` is not used in the code.

Comma-Separated Elements

Refactoring 5 (a) targets undisciplined directives in comma-separated program elements. In this refactoring, we set a precondition that the original code does not define a macro PARAM or contains a token with that name, such as a type definition or identifier. Again, we use a precondition to modify the code locally without global impact.

Refactoring 5 (a) ⟨undisciplined function definitions⟩

```
TYPE FUNC_NAME (
#ifdef EXP
    TYPE ID
#endif
){
    STMTS
}
```

↦

```
#ifdef EXP
#define PARAM TYPE ID
#else
#define PARAM ""
#endif
TYPE FUNC_NAME (PARAM){
    STMTS
}
```

(→) PARAM is not used in the code

In Refactoring 5 (b), we present a refactoring to remove undisciplined array definitions. Again, we use an additional macro (ELEM) to maintain the array or enum elements and define a precondition to modify the source code locally.

Refactoring 5 (b) ⟨undisciplined array definitions⟩

```
TYPE ID[] = {
    element_1,
    element_2
#ifdef EXP
    , element_3
#endif
};
```

↦

```
#ifdef EXP
    #define ELEM , element_3
#else
    #define ELEM ""
#endif
TYPE ID[] = {
    element_1,
    element_2
    ELEM
};
```

(→) ELEMS is not used in the code.

Appendix B

The C Model

In this appendix, we present the model of a subset of the C language that we used to generate programs with application possibilities for our refactorings automatically, as discussed in Chapter 6. The subset that we consider includes local and global variables, function definitions, `if` statements, and the following types: `char`, `int`, and `float`. We have not considered pointers, structures, loops, and concurrency.

Based on our C model, we used the *Alloy Analyzer* [158] to find instances that satisfy the model constraints. By using the instances provided by the *Alloy Analyzer*, our tool *Colligens* converts the instances into real C configurable programs with application possibilities for our refactorings. *Colligens* is responsible to introduce preprocessor conditional directives, such as `#ifdef` and `#endif`, in the generated programs. We have not considered the C preprocessor language in our model because of the complexities of dealing with undisciplined directives. As we discuss in Chapter 2, undisciplined directives can appear anywhere in the code and may wrap only parts of C constructors, making their specification in *Alloy* difficult.

In Listing B.1, we present part of the C model in which we define signatures to represent the main structure of a C program. We define that C program is a translation unit signature that contains a set of declarations. We define an identifier signature to name variables and functions that need to have unique identification. Next, we define a variable that has a specific type. We have signatures for other elements, such as statements, local and global variables, and parameters. Our complete C model is available at the Web site of the project.[1]

[1] `http://www.dsc.ufcg.edu.br/~spg/catalog/`

Listing B.1: Declarations of the C model.

```
1 abstract sig Declaration {}
2 sig TranslationUnit {
3   declares: set Declaration
4 }
5 abstract sig Identifier {}
6 abstract sig Variable {
7   type: one Type
8 }
9 // more signatures...
```

In Listing B.2, we present a signature for function definitions. In C, a function is a declaration with a unique identifier that receives a set of parameters, returns a value, and contains a set of statements. Notice that we considered in our model only functions that receives a single parameter. Furthermore, all functions considered in our model must return a value and must have exactly one if statement in its body. The reason to add these constraints is to generate programs with application possibilities for our refactorings.

Listing B.2: Declaration of a C function.

```
1 sig Function extends Declaration {
2   id: one FunctionId,
3   returnType: one Type,
4   ...
5   if: one If,
6   returnStmt: lone ReturnStmt
7 }
```

A valid C program must satisfy a number of well-formed rules. For example, a program cannot have two variables with the same identifier in the same scope, and a function should not have statements after returning a value and finish its execution. In Listing B.3, we present a few rules defined in our model. As we can see, we define that all programs must have declarations, all identifier used are unique, and that all local variable are declared in the function body.

Listing B.3: Well-formed rules for a C program.

```
fact Rules {
  translationUnitNotEmpty
  allIdentifiersAreUnique
  allLocalVariablesExistInFunction
  // more rules..
}
pred translationUnitNotEmpty {
  all src:TranslationUnit |
    #src.declares > 0
}
// more predicates..
```

To reduce the number of instances generated by the *Alloy Analyzer* with the purpose of avoiding the explosion of spaces, we defined some optimizations, such as that functions cannot have empty bodies, and all programs must have one global variable, one `if` statement, and two function definitions. We present part of the optimization predicate in Listing B.4.

Listing B.4: Optimizations to avoid explosion of spaces.

```
pred optimization[] {
  ...
  all f:Function | #f.stmt < 4 and #f.stmt > 0
  #Function = 2
  #GlobalVarDecl = 1
  #If = 1
}
```

By using the C model that we specified in *Alloy*, we can generate configurable programs with application possibilities for the refactorings of our catalogue, as we discussed in Chapter 6. After generating the configurable programs, *Colligens* generates the corresponding test cases and applies our refactorings automatically.

In addition to the configurable program presented in Chapter 6, we present another example of generated C program. In Figure B.1 (a), we present a configurable program with application possibility for Refactoring 3, with alternative `if` statements. Notice that the generated program follows the constraints defined in the model, e.g., all functions have a `return` statement and start with a local variable definition. Furthermore, the program does not have functions with empty bodies, contains an `if` statement, and exactly two function definitions. Notice that the generated program can be configured by defining macro `TAG` or not. So, we have two configurations in this program: (1) macro `TAG` enabled, and (2) macro `TAG` disabled. In Figure B.1 (b), we present the code that *Colligens* generates after refactoring the source code of the generated configurable program.

```
float Glob = -1.0F;

int Func1(int P0){
  int Local0 = 2;
  return Local0;
}

int Func0(int P0){
  int Local0 = 2;
#ifdef TAG
  if (Glob && Func1(P0)){
#else
  if (Glob){
#endif
    Glob += 1.0F;
    return Local0;
  }
  return Local0;
}
```
(a)

```
float Glob = -1.0F;

int Func1(int P0){
  int Local0 = 2;
  return Local0;
}

int Func0(int P0){
  int Local0 = 2;
  bool test;
#ifdef TAG
  test = Glob && Func1(P0);
#else
  test = Glob;
#endif
  if (test){
    Glob += 1.0F;
    return Local0;
  }
  return Local0;
}
```
(b)

Figure B.1: Program generated with possibility to apply our refactoring.