Master's Thesis Computing Science

Transforming Compile-Time to Load-Time Variability in C Projects

David Korsman
s4619579

8 February 2024

First supervisor/assessor:
Dr. Daniel Strüber

Second assessor:
Dr. Mathijs Schuts

Radboud University

Abstract

Programs written in the C language commonly use preprocessor directives such as `#ifdef` to delimit parts of the code that should not always be included in the final program. The editing decisions stemming from these directives are made at time of compilation, meaning any desired changes to the configuration require the program to be compiled anew. This is often an advantage, but it can also lead to problems. Furthermore, the intermixing of two structures in the source code – the one formed by the preprocessor directives and the to-be-formed structure of the C code itself – make unpreprocessed programs challenging to parse by analysis tools, and this is therefore seldom supported by such tools.

In this research we have studied the transformation of C code from its usual compile-time variability form, containing `#ifdef` preprocessor directives, to a run-time variability form, using equivalent `if` statements and other solutions. We have given a definition of twelve nontrivial cases for such transformations as well as an approach for transforming these cases programmatically, and we have developed a tool that aims to implement this approach, with partial success. We have also tested our tool on the source code of 22 open-source projects, including Linux. These tests identify a somewhat promising success rate, with an overall percentage of 97.0% of all files that the tool could transform without running into issues. Manual verification of a sample taken from these transformed files reveals that transformations in 87 out of 110 files were apparently valid, with varying success rates between different projects.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Daniel Strüber, for his advice, feedback and patience throughout this project. His guidance contributed to my self-confidence, and allowed me to finish this project with success. I would also like to thank my second assessor, Mathijs Schuts, for his feedback. Last but not least, I would like to thank my parents for always being encouraging and supportive.

Contents

Chapter 1

Introduction

Software products can contain features that can be added or removed at will. Being able to create different versions of a program by removing unneeded functionality and only adding what is needed can have benefits, such as making the resulting program less complicated, saving storage space, having a smaller attack vector or making the software work at all in a specific environment [5].

A common method for defining variability is preprocessor directives in C and C++, such as `#if` or `#ifdef`. This is an example of compile-time variability, which means a certain variant of a program needs to be derived from the source code at compile time. However, when needing to analyze all program variants, needing to build and test each different derived version may be infeasible due to the combinatorial explosion caused by even small numbers of features. Therefore, bugs that only occur in specific variants may go unnoticed, and it may be harder to maintain correctness of the program when having to account for all possible configurations. Even outside of program analysis and correctness, there are disadvantages to compile-time variability. For example, where it is desirable to change configuration options, it is not possible to do so dynamically at runtime, since the program needs to be recompiled entirely.

It may therefore be beneficial to transform instances of compile-time variability in a program (e.g. `#ifdef A <code> #endif`) to load-time variability (e.g. `if (pd_defined(A)) { <code> }`). Here, `pd_defined()` is a function which has access to the configuration while the program is being run. The goal thus is to eliminate preprocessor directives from the code, while keeping the behavior of the program equivalent. This could allow static analysis tools to analyze all variants of a program at once from a single

build (or a significantly reduced number of builds), and simplify the testing process. This is especially true because significant parts of the code in any product will be shared between all variants. Another benefit would be that load-time variability could provide greater flexibility in general, since the software will not need to be recompiled to make changes to the configuration, as long as sacrificing the advantages of compile-time variability (such as better performance and a smaller memory footprint) is acceptable.

Even if it is not possible to transform 100% of the compile-time variability to load-time variability, a partial transformation could still provide part of the advantages, such as increased analyzability and more flexible run-time configuration. This is related to *staged configuration*, where the number of possible configurations is reduced in stages by making parts of the feature selections at different points in time [6]. Indeed, a partial transformation from compile-time variability to load-time variability reduces the compile-time feature model.

The conversion process from compile-time variability to load-time variability is challenging, however. Preprocessor directives can "textually" include or exclude any arbitrary code at any point in the program, even where `if` statements required for load-time variability can not be used, and the programmer can choose to include or exclude even parts of statements in certain conditions. Therefore, the conversion process will need to apply different transformations to the code to ensure the result will remain legal and produce the same observable behavior as an original compile-time-derived variant of the program would.

The code example in listing 1.1 demonstrates a simple use of preprocessor directives for conditionally-compiled code. In this case, if the feature `A` is enabled, the statement is compiled as `return 1 + 2;`, but if it is disabled, the statement is instead compiled as `return 1;`. A simple find-and-replace of the `#ifdef` to a regular C `if` statement will not be possible, since `if` statements cannot occur in the middle of other statements. Therefore, a valid transformation to run-time variability might look as in listing 1.2.

```
1  int f(void) {
2      return 1
3  #ifdef A
4              + 2
5  #endif
6      ;
7  }
```

Listing 1.1: A simple use of preprocessor directives

```
1  int f(void) {
2      if (pd_defined("A")) {
3          return 1
4                  + 2
5          ;
6      } else {
7          return 1
8          ;
9      }
10 }
```

Listing 1.2: A result of transforming to run-time variability

In the example in listing 1.3, where a `struct` has optional fields depending on compile-time configuration, such an approach is not possible. `if` statements can only occur inside functions, and struct definitions cannot be made conditional in any way. A possible solution to this problem in this case might be to leave all attributes in the struct unconditional, as shown in listing 1.4. Like this example, there are many more conceivable examples of code that introduce challenges to transform, because `#ifdef` is exceedingly flexible.

```
1  struct point {
2      int x;
3      int y;
4  #ifdef POINT_3D
5      int z;
6  #endif
7  };
8
9  int f(struct point s) {
10     return s.x + s.y
11 #ifdef POINT_3D
12             + s.z
13 #endif
14     ;
15 }
```

Listing 1.3: Optional struct fields

```
1  struct point {
2      int x;
3      int y;
4      int z;
5  };
6
7  int f(struct point s) {
8      if (pd_defined("POINT_3D")) {
9          return s.x + s.y
10                 + s.z
11         ;
12     } else {
13         return s.x + s.y
14         ;
15     }
16 }
```

Listing 1.4: A result of transforming to run-time variability

Our research contribution will be a definition of nontrivial cases for transforming compile-time variability in C programs to run-time variability, an approach to programmatically transform these cases, and a tool that aims

to implement this approach. Due to the flexibility of `#ifdef` directives, which can textually include or exclude any arbitrary code at any point in the program, finding and handling all possible cases would be a monumental task. Luckily, real-world code may not be actively, "maliciously", trying to seek out these cases. Therefore, our approach is to support a systematically elected subset of problematic cases, retrieved from real software projects. To scope the project, part of these solutions were implemented into the tool, and part of the cases require manual intervention, while we do provide a description of how each case could be transformed programmatically.

While preprocessor directives are a broader concept that also applies to certain other languages, notably C++, we have decided to only focus on C in this project for a simpler and more achievable goal.

We address the following research questions:

RQ1 How can compile-time variability be transformed to run-time variability in C source code?

- RQ1.1 What (trivial and nontrivial) cases for transformation can exist in source code?
- RQ1.2 How could those cases be transformed?
- RQ1.3 Are there cases where compile-time variability should not be transformed to run-time variability?

RQ2 What is the quality of the developed tool for transforming compile-time variability to run-time variability?

- RQ2.1 What is the correctness of the tool?
- RQ2.2 What is the performance behavior of the tool?
- RQ2.3 How does our tool compare against existing, state-of-the art tools?
- RQ2.4 What additional insights does the tool give about open-source projects?

This report will detail the approach of transforming compile-time variability to run-time variability in C projects, as well as the process of developing a tool to do so. The remainder of this paper is structured as follows. Chapter 2 will give some background information, including about preprocessor directives and parser design. Chapter 4 will give an overview of our research process, and the (planned) process of developing our tool. Chapter 3 will outline related work. Chapter 5 will detail our approach of transforming compile-time variability to run-time variability, by examining different cases and patterns of C code, encompassing the answers to RQ1. Chapter 6 will

describe our implementation of this approach into a tool. Chapter 7 will focus on the evaluation of the developed tool, encompassing the answers to RQ2.

Chapter 2

Background

2.1 Preprocessor directives

In our previous work, we have also studied preprocessor directives, so this section with background information is shared with our previous work [11].

Preprocessor directives in C code follow a relatively simple syntax, where a line in a source file either is a directive or is not a directive. If the first non-whitespace character on a line is a hash symbol (`#`), then it is a directive. The hash is followed by the instruction given to the preprocessor, for example `include` or `ifdef`. Preprocessor directives are standardized in the C standard [3].

Some example code using preprocessor directives can be given as follows:

```
1  #ifdef __linux__
2      #include <sys/utsname.h>
3
4      struct utsname lin_ver;
5      uname(&lin_ver);
6
7      printf("Linux version: %s\n", lin_ver.release);
8  #endif
9  #ifdef _WIN32
10     #include <windows.h>
11
12     OSVERSIONINFOA win_ver;
13     ZeroMemory(&win_ver, sizeof(OSVERSIONINFOA));
14     win_ver.dwOSVersionInfoSize = sizeof(OSVERSIONINFOA);
15     GetVersionExA(&win_ver);
16
17     printf("Windows version: %d.%d\n", win_ver.dwMajorVersion,
```

```
        win_ver.dwMinorVersion);
#endif
```

This example also shows that code which can not be compiled (because it references missing header files and APIs meant for another operating system) can still be textually present, because the preprocessor decides whether to include or remove it before it is passed to the compiler. When this system is therefore used to implement optional features, it can really make the resulting product smaller and more optimized if certain features are unneeded. Just like `#ifdef`, there is also `#ifndef`, to only include code if a certain identifier is not defined.

Preprocessor directives can get more complicated than `#ifdef` and `#ifndef`. In fact, `#ifdef X` itself is shorthand notation for `#if defined(X)`. `#if` directives function on a constant expression that can consist of logic, comparison, and certain operators like `defined`. `#elif` and `#else` directives also exist, and work as one might expect. A more complicated example can thus be given as follows:

```
#if defined(__linux__) || defined(__unix__)
    /* Linux/Unix code... */
#elif defined(_WIN32)
    /* Windows code... */
#else
    /* Other code... */
#endif
```

In this example, we can reason that the second block of code will be included if the following condition holds:

```
!(defined(__linux__) || defined(__unix__)) && defined(_WIN32)
```

And the third block of code will be included if the following holds:

```
!(defined(__linux__) || defined(__unix__)) && !defined(_WIN32)
```

2.2 Parser design

Syntactically valid source code in a programming language usually follows a certain structure that can be described by the grammar of that language [4]. For example, the grammar could define that a program consists of a list of functions, which consist of lists of whole statements, where statements could be variable assignments or function calls, but also whole blocks that contain lists of statements themselves (such as for `if` statements and loops). A

parser for that programming language could thus create a tree-based data structure representing the source code, also known as an Abstract Syntax Tree (AST). An AST is a structured and balanced data structure that follows the grammar of the programming language.

Normally, a complete parser consists of a *lexer* or *tokenizer* step, followed by a *parser* step. The lexer or tokenizer translates a plain-text source code file (i.e. consisting of a sequence of characters such as `'i' 'f' '(' '1' '2' ')'`) into a sequence of *tokens*, which have a defined meaning in the programming language (such as `IF` and decimal `12` (twelve)).

The parser step then translates this sequence of tokens into an Abstract Syntax Tree (AST). This AST gives unambiguous semantics to the combinations and arrangements of tokens. For example, the code `val * 10` would, after tokenization, be an identifier `val` (function or variable), an asterisk (which is used for both multiplication and pointer declaration and dereferencing) and the decimal number 10. After parsing, this would unambiguously become the multiplication of two values: the variable `val` and the decimal number 10. A visual representation of this design is given in figure 2.1.

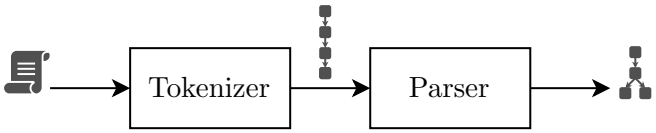

Figure 2.1: Regular parser design

In a compiler, the AST can be used to generate machine code. In a library like `pycparser`, which is not a compiler itself, the AST can be used for code analysis or similar purposes.

The C language, as well as related languages like C++, introduce a complication for this standard parser design: preprocessor directives can include or remove any arbitrary text at the time the source code is compiled, regardless of the grammar of the C language. The existence of these directives would significantly hamper generation of an AST, because preprocessor directives can subvert the regular structure and balance of the program. For a compiler, this is not an issue, since preprocessing is intended to be a step before parsing takes place, where the directives can be completely removed by making the corresponding "edits" in the source code directly. The conditions given by preprocessor directives are constant expressions that can be evaluated at time of compilation, such as features being enabled or disabled, or the operating system that the program is compiled for.

For tools that need to process or analyze source code as a whole, the situ-

ation is more complicated. It may be impossible to view the source code as a structured and balanced tree, because preprocessor directives may arbitrarily cut parts of the code covered by distinct nodes of the tree, cut out intermediate parent nodes, and anything else possibly resulting in a combinatorial explosion of possible program variants. Analysis tools that reason about source code as a whole may therefore prefer to not support preprocessor directives altogether, and therefore assume that a preprocessor has already been applied to source code. This includes `pycparser` [1], the library we based our tool on.

Parsing and working with C code containing preprocessor directives has been the subject of earlier tool development. For instance, a well-known tool is TypeChef [2], which is a research project to discover bugs caused by variability in C projects, and to typecheck the whole source code in a variability-aware fashion. Related is SuperC [8], which is an approach by Gazzillo et al. to parse C code containing preprocessor directives, by forking new *subparsers* when encountering a conditional and merging them again after the conditional.

Chapter 3

Related work

Von Rhein et al. made an analysis of transforming compile-time variability to load-time variability, which means creating a variant simulator of a program [18]. Their main contributions are related to a simplified version of the Java language. They also discuss hypothetical solutions to certain problems when transforming compile-time variability with preprocessor directives to load-time variability, as in C. These hypothetical solutions seem to not have been implemented in practice thus far - they are merely part of a discussion about the challenges of potentially automatically creating variant simulators of real-world programs in a language like C using preprocessor directives.

Rosenmüller et al. have presented an approach to allow seamless implementation of features without being forced to make choices between compile-time and load-time variability during the design process of the program [15]. In other words, their approach allows a programmer to implement a software product line (SPL) once and to decide per feature at deployment time whether it should be bound statically or dynamically. Their approach is based on feature-oriented programming (FOP), and is implemented on top of FeatureC++. The main drawback of this approach is that a programmer has to specifically choose to use this system at the time they design their program. It therefore does not concern existing real-world programs, unless the approach gains popularity.

Dimovski et al. have implemented an analysis for proving program termination, for `#if`-based C programs [7]. Since static-analysis tools can traditionally only analyze pre-processed single programs, they use variability-aware (lifted) analysis to prove that all variants of a program terminate. While termination may be an important property for certain types of programs, also from a validity and security viewpoint, some programs are never inten-

ded to terminate (e.g. server software) and it is only a small part of what constitutes a correct and secure program.

Lazar et al. have documented their experiences implementing software transformations with the goal to 'modernize' source code [9]. This transformation involves specifically structured code, which needed to be changed from an imperative style to a declarative, side-effect free style.

Schuts et al. have implemented and described a large-scale semi-automated migration of legacy testing code written in C and C++ [16]. This particular migration would not have been conducted without automation, due to the scale of the project and the risk of introducing errors and loss of productivity with a manual migration.

Meinicke et al. have analyzed the execution of Java programs with variability-aware execution, to study feature interactions in configurable programs [14]. They find that the *essential configuration complexity* ("the configuration-related differences in an execution that actually need to be explored given an optimal execution strategy") of the programs they studied is much lower than would be indicated by the combinatorial explosion of the configuration space, due to sharing of code among variants.

Liebig et al. have made a distinction between *disciplined* and *undisciplined* preprocessor directives - disciplined directives being defined as those directives that encompass only entire statements, functions, type definitions, or elements inside type definitions [13]. The purpose of this distinction was to make the implementation of tool support for code analysis significantly easier, by defining a standard that, if the source code of a program adheres to it, the code can be analyzed by a much wider variety of tools. They have also discussed transformations of undisciplined directives to disciplined ones.

Finally, von Rhein has similarly implemented a tool, Hercules, with the goal to convert compile-time variability (with `#ifdef`s) to load-time variability in an automated or semi-automated manner [17]. This tool is based on TypeChef, which is a variability-aware parser, and is written in the Scala language. As far as we are aware, this is the only previously available tool that aims to solve this problem. We have made a comparison between this and our own tool as part of this research.

Chapter 4

Methodology

This project has researched transformations of compile-time variability with preprocessor directives in C to load-time variability. This involves designing solutions for a number of different non-trivial applications of preprocessor directives for conversion to load-time variability. This process has been partially helped by prior research (such as [17] and [18]), which has given hypothetical solutions for certain problematic cases. Prior research does not provide complete coverage for all problematic cases one may encounter (more details on specific cases can be found in section 5.2), therefore we have needed to study possible cases and design solutions for those cases. We then attempted to apply these solutions in practice by implementing them into a tool. The goal to work towards is to be able to transform compile-time variability to load-time variability in real-world C programs.

The research methodology of this thesis is best described as a form of design science [10]. We developed an innovative artifact (the tool) to help solving a concrete problem, with cycles of problem understanding, implementation, and evaluation. This process comprises three main cycles, where the first cycle mostly focuses on prototyping a basic initial version, the second cycle consists of smaller "sub-cycles" of designing solutions to individual problematic cases, and the third cycle mostly focuses on evaluation and hardening the existing functionality. The phases in each "sub-cycle" are mostly related to the goals of the tool being able to support more and more cases: to understand the problem, we have had to discover the most important programming patterns that are blocking the tool from functioning on expected or real-world code. We then designed and implemented a solution to each part of the problem, and evaluated whether our solution works with specific test cases.

The goals of developing the tool are both to be useful in practice – for example, helping to increase the analyzability and flexibility of software – and also to explore our research questions. The development process of the tool itself will be described in more detail in chapter 6.

Main process

The starting point was a basic proof-of-concept, where a C program with compile-time variability limited to the level of whole statements inside function bodies could be transformed to load-time variability. Starting with the source code of `pycparser`, we extended the tokenization and parsing steps with basic recognition of `#ifdef` and `#endif` tokens. At the most basic level, these could unconditionally be transformed into corresponding `if` statements. This working example was then extended with problematic cases of variability, meaning recognition of cases had to be introduced where `#ifdef`s should be removed rather than transformed, or should be transformed in a different way.

Problematic cases can mainly be found where inserting a C `if` statement to replace the preprocessor `#if` directives is not possible without resulting in invalid syntax or different observable behavior. The aforementioned example with optional struct fields is an example of this. Another example is variable declarations conditional to an `#if` directive, because the transformation to run-time variability involves introducing a new block (at the level of C), meaning the variable has a smaller scope than originally intended. There are multiple ways to discover these special cases, which we have used, to a greater or lesser extent, for this research. Some cases are already documented in existing literature, for example [17] and [18]. Other cases were "self-developed", that is, deduced from prior knowledge of and experience with C. For lack of a better term, we will apply the word "deduction" to label such cases. It is also possible to explicitly look for additional cases by studying the syntax of C [3] or the source code of real-world software. This falls under roughly the same category, as a person's prior experience with C would also have been shaped by studying the language and reading existing code. For the end result, it does not matter if a case was conceived from memories formed by studying in the past, or directly from studying. (RQ1.1 and RQ1.2)

Once we could make valid transformations of more complicated constructs in programs, we applied testing techniques in order to test the implementation more rigorously, to further indicate and improve correctness. For each transformation case, we wrote a test case, and manually verified that the

expected results for these test cases were correct. These verified results are stored, so that if the result for a test case does not change, that case does not need further attention. This is also useful as a regression test suite: if the result for a test case changes later, it's easier to know what caused it. (RQ2.1)

The performance behavior of the tool can be measured by using the tool to process larger projects, and measuring the time it takes to complete each project. Knowing the scale of each project – such as the number of lines and source files – can help clarify whether the tool is practical enough to be applied to average software projects. (RQ2.2). We have examined the documentation and working of Hercules, the existing tool we are aware of which serves a similar purpose. (RQ2.3) By using the tool on real-world software, we may encounter additional insights about this software, such as unexpected parsing errors that were not caused by mistakes in our parser, or interesting patterns that were previously unknown. (RQ2.4)

Project selection

Since our ambition was to study real-world software projects, we needed to select such projects. As part of my research internship about higher-order feature interactions in open-source software projects [11][12], we selected 33 open-source C and C++ projects, which mainly consisted of popular GitHub repositories and projects studied in prior research. In this thesis, only C will be studied, not C++. 22 of the 33 projects from the previous selection would still qualify, as their (main) programming language is C. The full list of projects used in this research can be found in table 4.1.

Project	What is it	URL
Apache HTTP server	An HTTP server	https://github.com/apache/httpd
axTLS	A very configurable TLS library	http://axtls.sourceforge.net/
Busybox	UNIX toolkit	https://git.busybox.net/busybox/
Emacs	A (terminal and GUI) text editor	https://github.com/emacs-mirror/emacs
GIMP	A graphics editor	https://gitlab.gnome.org/GNOME/gimp
Gnumeric	A spreadsheet program	https://gitlab.gnome.org/GNOME/gnumeric
gnuplot	A plotting tool	https://github.com/gnuplot/gnuplot
Irssi	An IRC client	https://github.com/irssi/irssi
libxml2	An XML parser	https://gitlab.gnome.org/GNOME/libxml2
lighttpd	An HTTP server	https://git.lighttpd.net/lighttpd/lighttpd1.4.git/
Linux	The Linux kernel	https://github.com/torvalds/linux
mbedTLS	A portable, easy to use, readable and flexible TLS library	https://github.com/ARMmbed/mbedtls
MPSolve	A polynom solver	https://github.com/robol/MPSolve
Netdata	A real-time infrastructure monitoring system	https://github.com/netdata/netdata
NGINX	An HTTP server	https://github.com/nginx/nginx
OpenSSL	A TLS and crypto library	https://github.com/openssl/openssl
OpenVPN	A VPN client	https://github.com/OpenVPN/openvpn
Parrot	A virtual machine	https://github.com/parrot/parrot
Redis	An in-memory database that persists on disk	https://github.com/redis/redis
SQLite	A file-based database system	https://github.com/sqlite/sqlite
uClibc-ng	An embedded C library	https://uclibc-ng.org/
Vim	A terminal-based text editor	https://github.com/vim/vim

Table 4.1: The full list of projects

Experience from the same research internship can be applied to answer RQ1.3: we expect it to be necessary to make certain exceptions to the transformation from compile-time variability to run-time variability, particularly when APIs for operating systems such as Windows, macOS and Linux are used in software. The findings from that research can help to select which feature identifiers are exclusive to which operating system. To exempt such directives from being transformed, the approach is to preprocess those directives as would normally be done by a preprocessor. That is, our tool recognizes operating system recognition features like `_WIN32`, and only includes such sections if a Windows version of the code is requested. This way, the number of possible program variants would be limited to one variant per operating system. Alternatively, these directives could simply be retained in the transformed code.

The 'ideal' final goal for the project is complete support for transforming any valid C program from compile-time variability to load-time variability. Despite the relative simplicity of C compared to a more expressive language like C++ or Java, being able to transform any valid C program is unfortunately elusive, due to the nature of preprocessing to freely include or exclude parts of the source code before it is tokenized and parsed. Therefore, instead of attempting to implement support for all edge cases, we imposed constraints to forbid rarely used and/or difficult to handle cases or language constructs, and focus on a more realizable subset of the language.

Chapter 5

Approach

Our research process has consisted of several elements, such as design of an approach to transforming compile-time variability to run-time variability on paper, the development of a tool with the aim to implement that approach, and application of knowledge from prior research. This chapter will focus on the approach from a conceptual perspective, thus encompassing the answers to RQ1.

5.1 Transforming preprocessor directives

As mentioned in chapter 2, there are several reasons why one might want to transform compile-time variability to load-time variability. One of those reasons is that analysis of all configurations of a C program becomes much more difficult, because preprocessor directives coexist with and subvert the structure of the C code itself. Our approach thus aims to help general analysis of C code by removing preprocessor directives and transforming them into equivalent run-time code, before the code can be fed into tools that do not support preprocessor directives. On the other hand, we are now facing the exact same challenge: we are developing a tool that needs to reason about and modify the structure of C code that may contain arbitrary `#ifdef` statements.

In the most trivial case, an `#ifdef` is applied to a whole statement or set of statements inside a function. In this case, the `#ifdef` can be transformed to an `if` statement with an opening brace ({), and the corresponding `#endif` can be transformed to a closing brace (}). The conditional for the `if` statement needs to be possible to evaluate at run-time. Therefore, a static feature

like `FEAT_A` may need to be replaced by a variable or function call. We have decided to define the function `pd_defined()`, which takes a string representing the feature identifier as an argument. This choice was made for full flexibility, since it allows for the configuration to be easily human-modifiable, by editing a configuration file or even by employing a user interface at run-time. Thus, `#ifdef A` would be transformed into `if (pd_defined("A")) {`. A result of this transformation can be seen in listings 5.1 and 5.2.

```
1 int f(void) {
2     handle(x);
3     handle(y);
4 #ifdef POINT_3D
5     handle(z);
6 #endif
7 }
```

Listing 5.1: A use of preprocessor directives on a whole statement

```
1 int f(void) {
2     handle(x);
3     handle(y);
4     if (pd_defined("POINT_3D")) {
5         handle(z);
6     }
7 }
```

Listing 5.2: A result of transforming to run-time variability

Most cases cannot be solved with a simple find-and-replace action. If an `#ifdef` is applied to parts of statements, the transformation would need to duplicate unconditional parts of the statement to make the syntax valid again, since `if` statements cannot be applied to parts of statements. According to the definition of Liebig et al. [13], this is an example of an *undisciplined* directive. The code example in listing 5.3 demonstrates this. In this case, if the feature `A` is enabled, the statement is compiled as `return 1 + 2;`, but if it is disabled, the statement is instead compiled as `return 1;`. A transformation to run-time variability might look as in listing 5.4.

```
1 int f(void) {
2     return 1
3 #ifdef A
4         + 2
5 #endif
6     ;
7 }
```

Listing 5.3: An *undisciplined* use of preprocessor directives

```
1  int f(void) {
2      if (pd_defined("A")) {
3          return 1
4                 + 2
5          ;
6      } else {
7          return 1
8          ;
9      }
10 }
```

Listing 5.4: A result of transforming to run-time variability

5.2 Nontrivial transformation cases

We will detail the difficult transformation cases we discovered, as well as possible solutions to transform these cases, in this section. An overview of all cases and the extent of support can be found in table 5.1. The cases will each be explained in more detail in the sections below.

In addition to our tool, we found a tool with a similar goal - Hercules [17] - which is based on TypeChef. Since it is the only previously available approach for the problem that we are aware of, we have included it in the table for comparison. While Hercules seems to have support for a broader set of cases than our tool, we believe there are situations where our tool would be more suitable. This comparison between our tool and Hercules is explained in more detail in section 7.2.

The column "Origin" lists the method or source through which we discovered each case. As described in chapter 4, some cases were documented in prior research, and others (which are labeled "Trivial" or "Deduction") were deduced from knowledge and experience with C (possibly including studying done for this research project specifically).

There is a caveat to this column. While some cases were inspired by or quoted from prior research, we have always studied possible solutions ourselves. We therefore do not always agree with the solutions given in prior research, since there is not necessarily a single exact method to resolve a case. In some cases, the prior research offered a solution idea, but did not implement the solution. Therefore, this column should mostly be interpreted as "Inspiration": we discovered a problem case by reading prior research. The arguments for these disagreements will be given in more detail in the sections corresponding to each case.

		Supported in our tool	Supported in Hercules
General functionality			
Multiple files support		●	○
Can parse with incomplete declarations or big `#include`s		●	○
Supports run-time variability		●	○
Transformation cases	**Origin**		
1. Optional whole statement	Trivial	●	●
2. Optional part of statement	Trivial	●	●
3. Optional conditionals and braces	Deduction	●	●
4. `#if 0` and `#ifdef __cplusplus`	Deduction	●	◒
5. Struct with optional fields	[18]	●	◒
6. Alternative types	[18]	○ †	◒
7. Variable declarations inside `#ifdef`s	Deduction	○ †	◒
8. Optional shadowing	[18]	○ †	○
9. Conditional `#define`	[17]	◒ †	◒
10. Conditional goto	[17]	○ †	○ †
11. Optional switch cases	[17]	○ †	○ †
12. Compile-time string concatenation	[17]	◒ †	◒

Table 5.1: The nontrivial cases we discovered
○ no support ◒ partial support ● full support
† hypothetically resolved in paper

In the end, while we believe all these cases should be possible to transform programmatically, we have needed to consider many of these nontrivial cases out of scope for our tool due to the amount of time and work needed. However, where encountered, these can still be transformed manually, according to the transformations we give in the following sections.

5.2.1 Case 1: Optional whole statement

As described in section 5.1, by the definition of Liebig et al., the simplest transformation case is where `#ifdef`s are *disciplined*, such as those that enclose whole statements inside functions only. An example of this case is given by listings 5.1 and 5.2, and described further in section 5.1.

5.2.2 Case 2: Optional part of statement

Similarly, if an `#ifdef` splits up a statement into separate parts that will be merged into a single statement by the preprocessor, a transformation to run-time variability should duplicate every variant of the entire statement to let the transformation result remain valid C code. An example of this case is given by listings 5.3 and 5.4, and described further in section 5.1.

5.2.3 Case 3: Optional conditionals and braces

It may occur that a certain block of code should always run in a certain compile-time configuration, but be conditional at runtime in another compile-time configuration. For example, an authentication feature may prevent the user from carrying out certain actions if not logged in, but if the authentication feature is not configured, any user should be able to carry out those actions. This is shown in listing 5.5.

```
1 #ifdef AUTH
2     if (logged_in)
3 #endif
4         do_something();
```

Listing 5.5: Optional if conditional

```
1 if (!pd_defined("AUTH") || logged_in)
2     do_something();
```

Listing 5.6: A result of transforming to run-time variability

The most elegant transformation (as would be made by a human programmer) is given in listing 5.6. A valid alternative however, is to simply duplicate the call to `do_something()`, and have one branch where `logged_in` is checked and another branch where the function is called unconditionally. This is the approach taken by both our tool and Hercules.

A similar edge case is when both an opening brace and its corresponding closing brace are removed by `#ifdef`s, as given in listing 5.7. The interesting implication of this is that depending on the value of the feature given by the `#ifdef`, only the first, or all statements in the block become optional at run-time. A possible transformation is therefore given in listing 5.8.

```
1 if (special_check)
2 #ifdef AA
3 {
4 #endif
5     do_something_sometimes();
6     do_something_maybe_always();
7 #ifdef AA
8 }
9 #endif
```

Listing 5.7: Optional braces

```
1  if (pd_defined("AA")) {
2      if (special_check)
3      {
4          do_something_sometimes();
5          do_something_maybe_always();
6      }
7  } else {
8      if (special_check)
9          do_something_sometimes();
10     do_something_maybe_always();
11 }
```

Listing 5.8: A result of transforming to run-time variability

5.2.4 Case 4: `#if 0` and `#ifdef __cplusplus`

Code within an `#if 0` directive will never evaluate to be included. It is therefore sometimes used by programmers to "comment out" blocks of code without having to use standard `/* */` comments, since the latter requires existing `/* */` comments in the code to be modified or removed to avoid a syntactical conflict (for example, by changing existing `*/` to `* /`).

There is no requirement that text commented out by an `#if 0` is valid C code, and in fact, real-world code sometimes does include unparseable code in `#if 0` blocks [11]. An extreme example of this possibility, which is entirely legal and balanced, is given as follows:

```
1 #if 0
2 This is not actually code! But this comment is real: /* this
3 #endif
4 #endif
5 aa */ aa
6 #endif
```

Listing 5.9: A demonstration of `#if 0` and comments

Depending on the goal of the transformation, it may or may not be necessary to remove the block. In either case, it should not be transformed into an `if (0)` statement, since this would cause the resulting code to be syntactically invalid regardless of configuration or further preprocessing. If the goal is to compile the program after the transformation, then the `#if 0` block could be left in place as-is, because it does not affect the number of configurations and does not stop compilers from building the program, as long as the regular preprocessor is still in place. However, if the goal is to analyze the source code using analysis tools, then it would be better to remove the block completely. This allows tools that do not support preprocessor directives to parse the code without them.

A similar problem applies to the feature identifier `__cplusplus`, which is used to distinguish if a compiler is set to compile C or C++. This is used in case code needs to be able to compile as both C and C++, and a C++ construct must be used without affecting compilations as C. Since C++ syntax is unsupported by tools which focus on C, including our parser, we decided to treat `__cplusplus` as always `0`, and thus remove the blocks altogether in our transformations.

5.2.5 Case 5: Struct with optional fields

In this case, `#ifdef`s appear in the middle of a struct definition, so the struct has different fields in different configurations.

```
1  struct point {
2      int x;
3      int y;
4  #ifdef POINT_3D
5      int z;
6  #endif
7  };
8
9  int f(struct point s) {
10     return s.x + s.y
11 #ifdef POINT_3D
12             + s.z
13 #endif
14     ;
15 }
```

Listing 5.10: Optional struct fields

```
1  struct point {
2      int x;
3      int y;
4      int z;
5  };
6
7  int f(struct point s) {
8      if (pd_defined("POINT_3D")) {
9          return s.x + s.y
10                 + s.z
11         ;
12     } else {
13         return s.x + s.y
14         ;
15     }
16 }
```

Listing 5.11: A result of transforming to run-time variability

This case is also described in [18]. However, we provide an alternative solution. The solution presented in [18] is to duplicate each unique variant of the struct, and give a unique name to each. In all parts of the code that refer to the original struct, conditional code needs to be added to switch between the correct struct variant at runtime. The reason for this code duplication is that the decision to include or exclude fields from a struct affects the return value of `sizeof()` at runtime, which could count as a change in behavior. However, we think this is not a big problem. The intended usecase for `sizeof()` is to check the required amount of memory to store the data. Transformed code might break if it had unusual workarounds such as checking size to see which fields are in the struct rather than using `#ifdef`s, but we have yet to encounter this in real-world software.

Therefore, our solution is to simply make all fields unconditional, as seen in listing 5.11. There is space reserved for unused struct fields, but this is otherwise harmless since these fields will not be used. A possible conflict might occur if two fields in mutually incompatible configurations have the same name, and now occur together. The solution in that case would be to rename one of the fields and all its usages in code.

A possible change in behavior might happen if a union contains multiple structs, of which not all have optional fields, and most importantly, the program writes to one of the structs and then reads from another. An example of this situation is given in listing 5.12. In this case, when transformed, the fields of the different structs may be aligned differently than in the original program. This seems to be a very rare scenario, and the solution may not be very straightforward unless structures share a common initial sequence of member types, so this would require manual intervention. In listing 5.13, our solution is to dynamically change the field that is read out. In more complicated scenarios, for example if the initial types mismatch between the structs, it would also be possible to insert extra structs inside the union to use in different configurations.

```
1  union u {
2      struct {
3  #ifdef HAS_X
4          int x;
5  #endif
6          int y;
7          int z;
8      } point;
9      struct {
10         int mem_0;
11         int mem_1;
12         int mem_2;
13     } mem;
14 };
15
16 union u uu;
17 uu.point.y = 2;
18 uu.point.z = 5;
19 printf("%d\n", uu.mem.mem_1);
```

Listing 5.12: Optional struct fields inside a union

```
1  union u {
2      struct {
3          int x;
4          int y;
5          int z;
6      } point;
7      struct {
8          int mem_0;
9          int mem_1;
10         int mem_2;
11     } mem;
12 };
13
14 union u uu;
15 uu.point.y = 2;
16 uu.point.z = 5;
17 printf("%d\n", pd_defined("HAS_X")
       ? uu.mem.mem_1 :
       uu.mem.mem_2);
```

Listing 5.13: A result of transforming to run-time variability

5.2.6 Case 6: Alternative types

As briefly described in [18], it would be possible to declare a variable with a type dependent on configuration. An example of this is given in listing 5.14, where the variable name `altt_i` is of type `double` if the configuration option `DO_DOUBLE` is set, and type `int` otherwise. As noted, the variable declaration needs to be duplicated with separate identifiers, and all uses of the variable need to be modified accordingly as well. A solution here would thus be to declare a separate `double _double_altt_t = 1;` as

well as `int _int_altt_i = 1;`. However, for behavior preservation, it is also important to note that the expression that initializes a variable (in this case, `1`) may not necessarily be free of side effects. For example, it can be the return value of a function call, or it can contain increment (`++`) or decrement (`--`) operations. Therefore, our solution is to make the initial value of the variable conditional to whether that variant of the variable should exist based on configuration, as shown in listing 5.15.

If the variable is unused in the current configuration, the variable will be initialized by a default 'placeholder' value. To avoid introducing compiler warnings, the types of the placeholder values should match or be compatible with the types of each variable. For example, a numeric type could be initialized to 0, and a pointer type could be initialized to `NULL`. Initialization of struct types can be done using compound literals, with a 0 initializer-list as a default value, which looks like `(struct point){0}`.

```
1 #ifdef DO_DOUBLE
2     double
3 #else
4     int
5 #endif
6     altt_i  = 1;
7
8 handle( altt_i );
```

Listing 5.14: Alternative types

```
1 double _double_altt_i =
      pd_defined("DO_DOUBLE") ? 1 :
      0;
2 int  _int_altt_i  =
      !pd_defined("DO_DOUBLE") ? 0
      : 1;
3
4 handle(pd_defined("DO_DOUBLE") ?
      _double_altt_i : _int_altt_i);
```

Listing 5.15: A result of transforming to run-time variability

In Hercules, the variable renaming approach is followed, but initial value expressions are duplicated, even when they cause side-effects. A result of this transformation, when `init()` is used for the initialization of the variable instead of `1`, is given in listing 5.17.

```
1 #ifdef DO_DOUBLE
2     double
3 #else
4     int
5 #endif
6     altt_i  = init();
7
8 handle( altt_i );
```

Listing 5.16: Alternative types

```
1 double _double_altt_i = init();
2 int  _int_altt_i  = init();
3
4 handle(pd_defined("DO_DOUBLE") ?
      _double_altt_i : _int_altt_i);
```

Listing 5.17: An incorrect transformation of alternative types

C is not a functional programming language, so there is no method to ensure the implementation of the `init()` function is free of side-effects. Therefore, because `init()` is called twice, this transformation might have different behavior compared to the original program.

5.2.7 Case 7: Variable declarations inside `#ifdefs` being visible after the new block

If a variable is declared within a block in C (such as within a function, or inside the {} of an `if` statement), then the scope of that variable is limited to that block; the variable ceases to exist beyond the corresponding }. This forms a problem if we simply change `#ifdef`s to `if`s, because `#ifdef`s do not limit the scope of variables declared inside them, while `if` blocks do. This problem may be solved by moving variable declarations to an earlier point in the code as part of the transformation, or by interrupting the generated `if` block for the variable declaration and re-opening it afterwards. We have chosen the latter approach. This is most important for variables that are declared with the `const` qualifier, which enforces the constraint that the variable may not be modified after initialization. If the variable declaration were to be moved to an earlier point in the code, and the first value set later, the `const` qualifier would need to be removed, negating the reason it was used and weakening compiler type checks. The result of this transformation can be seen in figure 5.19. As in Case 6, the expression that initializes the (now unconditionally declared) variable may not be side-effect free, so the original initialization should be changed to only occur if the variable is used in this configuration, using a ternary expression.

```
1  #ifdef AA
2  aa_only_code_1();
3  const int i = aa_only_get_i();
4  aa_only_code_2();
5  #endif
6
7  code();
8
9  #ifdef AA
10 aa_use(i);
11 #endif
```

Listing 5.18: Variable declarations inside #ifdefs

```
1  if (pd_defined("AA")) {
2      aa_only_code_1();
3  }
4  const int i = pd_defined("AA") ?
       aa_only_get_i() : 0;
5  if (pd_defined("AA")) {
6      aa_only_code_2();
7  }
8
9  code();
10
11 if (pd_defined("AA")) {
12     aa_use(i);
13 }
```

Listing 5.19: A result of transforming to run-time variability

5.2.8 Case 8: Optional shadowing

Similar to Case 7, a variable declaration can be declared conditionally to an `#ifdef`, but in this case, the declaration 'shadows' another variable with the same name in an outer scope. This is a feature of C, where a variable in an inner scope can temporarily override another variable with the same name, as long as the scope is different [3]. This case is also highlighted in [18].

```
1  int shadow = 0;
2  for (int i = 0; i < 10; i++) {
3      #ifdef SHADOWS
4      int shadow = 1;
5      #endif
6      shadow++;
7  }
8  int whatshadow = shadow;
```

Listing 5.20: Optional shadowing

```
1  int shadow = 0;
2  for (int i = 0; i < 10; i++) {
3      if (pd_defined("SHADOWS")) {}
           // (interrupted because
           variable declaration)
4      int _opt_shadow = 1;
5      if (pd_defined("SHADOWS")) {}
           // (the compiler will probably
           optimize this away...)
6      if (pd_defined("SHADOWS")) {
7          _opt_shadow++;
8      } else {
9          shadow++;
10     }
11 }
12 int whatshadow = shadow;
```

Listing 5.21: A result of transforming to run-time variability

5.2.9 Case 9: Conditional `#define`

Not just code can be conditional to `#ifdef`s – other preprocessor directives such as `#define` can be conditional too. This means a given identifier can have multiple different definitions in different configurations of the program, and `#ifdef`s can be hidden in code that does not appear to have them, by simply referencing an identifier that depended on an `#ifdef` for its definition. An example of this problem can be found in listing 5.22.

There are two methods to transform this: changing uses of such tokens to ternary conditionals, or copying the `#ifdef`s in place.

If the replaced token is part of an expression, then the former method could be applied. An example is the replacement of `ALTT2_L` in listing 5.23. This method leads to less code duplication, but cannot always be applied, since ternaries cannot be used anywhere in the code. An automatic transforma-

tion would need to recognize which syntax changes are valid and only apply those. Care also needs to be taken not to cause unexpected changes in the order of operations. For example, if the expression `TWO_OR_TWELVE + 8` is changed to `pd_defined("TWO_OR_TWELVE")?2:12 + 8`, then the `+ 8` takes higher priority than the ternary, causing the possible resulting values to change to 2 and 20, not 10 and 20.

Copying `#ifdef`s in place is the second method. In this case, every usage of the defined identifier is changed to include an `#if` chain to give it all possible values for every configuration. The intermediate result of this change could look like listing 5.14 in Case 6 (Alternative types). Afterwards, the `#if` chain can be transformed as would otherwise be done. The end result can be seen in the transformation of `ALTT2_T` in listing 5.23.

```
1  #ifdef DEF_DOUBLE
2    #define ALTT2_L 6
3    #define ALTT2_T double
4  #else
5    #define ALTT2_L 3
6    #define ALTT2_T int
7  #endif
8
9  printf("We're using a type of len
       %d\n", ALTT2_L);
10
11 ALTT2_T altt2_i = 5;
```

Listing 5.22: Conditional #define

```
1  printf("We're using a type of len
       %d\n",
       pd_defined("DEF_DOUBLE") ? 6
       : 3);
2
3  double _double_altt2_i =
       pd_defined("DEF_DOUBLE") ? 5
       : 0;
4  int  _int_altt2_i  =
       !pd_defined("DEF_DOUBLE") ?
       0 : 5;
```

Listing 5.23: A result of transforming to run-time variability

5.2.10 Case 10: Conditional goto

C has the (sometimes controversial) possibility of jumping to a different statement by using a `goto` statement. `goto` can jump to any label in the same function, which is an identifier followed by a colon (`:`) that can be placed before any statement. Of course, it is possible using `#ifdef`s to add and remove labels depending on configuration. Thus, a given `goto` which is not conditional to `#ifdef`s itself, could still jump to different statements depending on compile-time configuration. An example of this setup is given in listing 5.24.

A solution for transforming this to run-time variability would be to rename the labels, and add conditionality to each `goto` statement which jumps to such a label. An example of this solution is given in listing 5.25.

This is also the approach described by von Rhein et al [17]. Interestingly, in our testing, Hercules did not handle this situation. Therefore, this description was likely also intended as a hypothetical approach, not as documentation of implemented behavior.

```
1 #ifdef INFINITE_LOOP
2 cursed:
3 #endif
4   str = "Well...";
5   goto cursed;
6 #ifndef INFINITE_LOOP
7 cursed:
8 #endif
```

Listing 5.24: Conditional goto

```
1 _inf_cursed:
2   str = "Well...";
3
4   if (pd_defined("INFINITE_LOOP"))
5     goto _inf_cursed;
6   else
7     goto _noinf_cursed;
8 _noinf_cursed:
```

Listing 5.25: A result of transforming to run-time variability

5.2.11 Case 11: Optional switch cases that override the default

`switch` statements in C are used as a form of lookup table for the possible values of a given variable. They are more flexible than that, however: if a `break` statement is not used at the end of a case, then the following case will be executed as well. This is called a *fallthrough*. Additionally, it's possible to specify a `default` case, which will be executed if none of the other cases match the value of the variable. Therefore, the behavior of `switch` statements is quite flexible.

Naturally, `#ifdef`s can be added anywhere in them to remove certain (parts of) cases, and thus change the behavior in different configurations. An example of a `switch` statement is given in listing 5.26. In this example, cases 3 and 5 are not present in every configuration. If `ALSO_CASE3` is defined, then case 2 will fallthrough to case 3, otherwise it will fallthrough to case 4. If `i` has the value 3 or 5, and the corresponding case is disabled, then the `default` branch will be used instead.

The main solution to transform this is to insert labels and `goto` statements to simulate the behavior in different configurations dynamically. An example of this transformation is given in listing 5.27. Still, the endless flexibility that `#ifdef`s allow for makes this problem hard to solve generically. For example, a case label could be removed, but not all code below it. Switch statements can also be nested, meaning inner `case`s could be made to look like outer `case`s in only some configurations.

Another valid solution is a bruteforcing approach, to duplicate the entire `switch` block for all possible feature permutations. This can lead to a combinatorial explosion of variants of the same `switch` block, but this is only determined by the amount of features present inside that `switch` statement.

```
1  switch (i)
2  {
3  case 0:
4  case 1:
5  case 2:
6      puts("Cases 0 1 2");
7      // fallthrough !
8  #ifdef ALSO_CASE3
9  case 3:
10     puts("And 3!");
11     break;
12 #endif
13 case 4:
14     puts("And not 3!");
15     break;
16 #ifdef ALSO_CASE5
17 case 5:
18     puts("This is case 5, also
           conditional to a define");
19     break;
20 #endif
21 default:
22     puts("Case not recognized, maybe
           it was 3/5 and not
           ALSO_CASE3/5?");
23 }
```

Listing 5.26: Optional switch cases

```
1  switch (i)
2  {
3  case 0:
4  case 1:
5  case 2:
6      puts("Cases 0 1 2");
7      // fallthrough !
8      if (!pd_defined("ALSO_CASE3"))
9      {
10         goto case_4_sw;
11     }
12 case 3:
13     if (!pd_defined("ALSO_CASE3"))
14     {
15         goto default_sw;
16     }
17     puts("And 3!");
18     break;
19 case 4:
20 case_4_sw:
21     puts("And not 3!");
22     break;
23 case 5:
24     if (!pd_defined("ALSO_CASE5"))
25     {
26         goto default_sw;
27     }
28     puts("This is case 5, also
           conditional to a define");
29     break;
30 default:
31 default_sw:
32     puts("Case not recognized, maybe
           it was 3/5 and not
           ALSO_CASE3/5?");
33 }
```

Listing 5.27: A result of transforming to run-time variability

5.2.12 Case 12: Compile-time string concatenation

Unlike many other programming languages, C does not have a straightforward method to concatenate multiple strings at runtime to form a new

string, such as `"Hello, " + name`. It only allows a string literal to be composed of multiple string literals at compile time, so `"A" "B"` would be equivalent to `"AB"`. It is therefore possible to have differently concatenated string literals depending on which features are enabled or disabled. For example, a program could display a help menu with only documentation about features that are enabled, as displayed in listing 5.28. A transformation could be to simply duplicate every possible version of the string, as shown in listing 5.29.

```
1  puts(
2      "Welcome to the documentation of
           this configurable program!\n"
3      "To launch the rocket, press the
           big LAUNCH button.\n"
4  #ifdef CANCELABLE
5      "To cancel the launch, press the
           CANCEL button.\n"
6  #endif
7  );
```

Listing 5.28: Compile-time string concatenation

```
1  if (pd_defined("CANCELABLE")) {
2      puts(
3          "Welcome to the
               documentation of this
               configurable program!\n"
4          "To launch the rocket, press
               the big LAUNCH
               button.\n"
5          "To cancel the launch, press
               the CANCEL button.\n"
6      );
7  } else {
8      puts(
9          "Welcome to the
               documentation of this
               configurable program!\n"
10         "To launch the rocket, press
               the big LAUNCH
               button.\n"
11     );
12 }
```

Listing 5.29: A result of transforming to run-time variability

The example given here is a relatively trivial case. However, it is possible to include many optional string parts, which would lead to a combinatorial explosion of unique strings. Therefore, a more scalable solution would be to transform cases like this to code that concatenates the string at run-time (using, for example, the `strcat()` family of functions, to a buffer that is the maximum possible length if all string parts were added). Due to time constraints and the complexity involved, we have decided not to implement that solution in our tool, and neither have the authors of Hercules.

5.3 Non-transformation cases

There may be circumstances where some `#ifdef`s should not be transformed to run-time variability, and should in fact remain `#ifdef`s as-is. These cases will be described in this section.

One common case that we encountered in software is **usage of operating system dependent programming interfaces**. Operating systems provide Application Programming Interfaces (APIs) that can be used by programs for various functionality, such as file and window management, or internet access. However, many of these APIs are unique to each individual operating system, and a C program cannot be compiled if it uses "foreign" APIs or header files that do not exist on every operating system, such as `windows.h` which is only present on Microsoft Windows, or `sys/stat.h` which is only present on POSIX-based systems like Linux or macOS. A common solution to this problem is to guard both these `#include` directives as well as the functions that are used from them, with `#ifdef`s, such that no compiler ever encounters missing functionality from another operating system. It would therefore break a program in practice to transform these cases to run-time variability, since a program that uses non-cross-platform functions from two different operating systems will not be compilable on any operating system. Yet, if the goal of transforming the source code is only to analyze it using analysis tools, it may not be a problem to do this transformation. This is because such tools may not require all function definitions or implementations to be present, or to match the operating system the tool is run on.

Whenever it is desired that the transformation result be compilable, another option might be to provide empty placeholder functions for all incompatible operating system-specific APIs, and not support the user selecting a configuration for an incorrect operating system. However, this may not be practical, as it requires assembling and maintaining a list of all functions that exist in each operating system. Another solution is to simply blacklist the relevant feature flags from being transformed to run-time variability. Since there is only a limited number of operating systems, keeping these `#ifdef`s in source code does not result in a very high number of unique program configurations, so it's possible to transform e.g. a Windows variant and a Linux variant of a program separately.

Another commonly used pattern is **include guards**. In this pattern, the code in an entire file (normally a header file) is made conditional to a `#define` placed in the same file, to include the code only once within any compilation unit, even if multiple `#include` directives refer to it. This is

necessary to avoid multiple inclusions of the same file causing the program to be invalid because of duplicated declarations for functions and variables. The main cause of this duplication is that header files often need to include other header files themselves. An example of include guards is given in listing 5.30.

```
1 #ifndef UTIL_H
2 #define UTIL_H
3
4 /* rest of file goes here... */
5
6 #endif
```

Listing 5.30: Example of include guards (`util.h`)

This pattern cannot be transformed to run-time variability, since there is no variability at play here – there is no code that is present in some configurations and absent in others. Therefore, the best option is to leave these untouched.

There is an alternative to include guards, namely the `#pragma once` directive, which achieves the same goal by simply including it in the code instead. This directive is widely supported by compilers, but it is not defined by the C standard [3]. Some programs may also aim to support older versions of compilers which still lacked this directive. Therefore, traditional include guards are still commonly found in source code.

Chapter 6

Description of the tool

To provide practical support for transformations from compile-time variability to run-time variability, we developed a tool that aims to implement the previously identified cases and solution strategies.

Our tool[1] is written in Python, and is based on the tokenizer of `pycparser` [1], a parser for C99 [3] written in Python. As described in section 2.2, `pycparser` follows the standard design of having a tokenizer, which converts plain text into tokens, followed by a parser, which converts tokens into an Abstract Syntax Tree. As also described, generation of an AST is significantly hampered by the existence of preprocessor directives, therefore `pycparser` does not support them and expects code to be pre-processed before being parsed. Therefore, we have modified `pycparser` to handle preprocessor directives in a special way, as described in chapter 5. Now we will detail our implementation of this approach.

We extended the tokenizer step of `pycparser` with support for preprocessor directives, and replaced the regular parser step by a so-called *transformator* step. The main purpose of the transformator step is to convert all `#ifdef` and `#if` preprocessor directives into plain C code wherever possible, at the level of tokens. In other words, it takes the output from the tokenizer step (a list of tokens), and analyzes and modifies it to give a new list of tokens. This could, depending on the placement of the directives, involve simple changes from preprocessor directives to `if` statements, duplication of parts of statements or blocks to resolve syntactic problems, or in some cases removal of the conditionality.

Conceptually, the output of the *transformator* step should be ready to be

[1] https://github.com/dkorsman/pycparser/tree/trafo

consumed by the original *parser* step that would normally get its input from the *tokenization* step. After all, any conditional preprocessor directives would have been changed into regular C code, so the structure and balance of the program should be valid and consistent. However, there are more complicating factors preventing the parser step from working, mainly the preservation of `#include` and `#define` directives. Since we did not make any changes to the parser step of `pycparser`, meaning none of our processing requires that step, supporting it is considered out of scope for this project. Instead, output from the *transformator* step is output as regular C code, which could be compiled with a real C compiler.

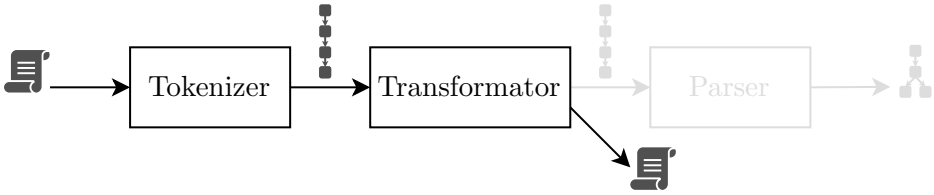

Figure 6.1: Our transformator design

6.1 Functionality

Given a single file, our tool will transform that file and output it. The tool can also be given an input and output directory, in which case it will enumerate all files with a recognized C extension (`.c` or `.h`) in the input directory and its subdirectories. It will then write transformed versions of these files to the output directory.

The tool also has some options to control the transformation process. The option `-p` can be used to change the target platform of the code (such as Linux, Windows or macOS), and can also be set to treat operating system-specific feature identifiers as regular features (if the code does not need to be compilable). Specific features can be treated as always disabled with `-U` (for example, `-U AA -U BB` to treat `AA` and `BB` as always `0`). For better support for *staged configuration* (as described in [6]), it would be relatively simple to modify the tool to also accept a list of features that should be transformed to run-time variability, leaving all other features completely untransformed.

One more challenge was how to make our new function, `pd_defined()`, work in C code in a universally applicable way. The implementation of a function may not be declared multiple times in C [3], so we cannot simply copy the definition into all transformed files. If we add our own source file to a project, we would need to choose the correct folder to place it in, even

though folder layouts are not standardized in C projects. Furthermore, this will likely require modifications to build scripts to compile this new source file. These build scripts may be set up in a unique way in any given project, making it difficult to automate modifying them. Therefore, our aim was to avoid the need to modify build scripts at all.

Our basic solution was to inject our function definitions in any file containing a `main()` function. Since `main()` is a standard function intended for any regular C program to use as its entry point [3] and since definitions of `main()` must be unique like any other function, this method should be reasonably robust for most programs. In other cases, such as when parts of the program are linked independently from each other, or in highly low-level projects such as the Linux kernel, this may still need manual intervention.

6.2 Load-time variability versus run-time variability

Compile-time variability means that configuration options are determined at time of compilation, and cannot be changed without recompiling the program. There is a further distinction between load-time variability and run-time variability. *Load-time* variability implies that the configuration can be changed without recompiling the program, but the program needs to be restarted in order to change the configuration. *Run-time* variability implies that the configuration can be changed at any time, even while the program is still running.

Our tool adds support for transformation to both load-time variability and run-time variability. Load-time variability is implemented with a configuration text file that is read when the transformed program is started, which can contain all features and their values. This text file is editable by the user, and follows a simple `key=value` format. Run-time variability is an optional addition, and is configured with the feature `PD_GUI` as a load-time variability option. If this feature is enabled, a separate window will open with a list of all features as loaded from the configuration file, which can then be enabled or disabled interactively. An example is shown in figure 6.2. Like the implementation of our `pd_defined()` function, the implementation of this user interface is injected into the file containing the `main()` function.

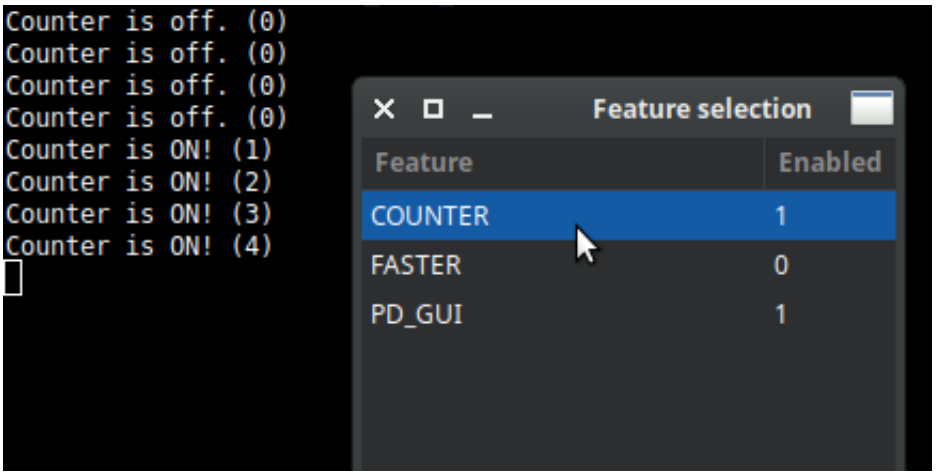

Figure 6.2: A screenshot of the user interface for run-time variability

Run-time variability has a much lower chance of working as expected than load-time variability – as the `#ifdef` conditionals were originally not designed to be able to change during program execution, crucial initialization steps may have been permanently missed by the time the program is up and running. Still, it might be very helpful to test different configurations if it works, especially with staged configuration in mind.

Chapter 7

Tool evaluation

This chapter will focus on an evaluation of the developed tool, encompassing the answers to RQ2. We have evaluated the correctness of transformations made by the tool by testing it on manually-written test cases as well as real-world open-source projects, and we have analyzed in which situations our tool could still be improved. We have also looked at the performance characteristics of our tool, and we have made a comparison between our tool and Hercules.

7.1 Correctness

Our methodology to verify the correctness of our tool, to answer RQ2.1, was based on testing techniques. We used two separate suites of test cases: a set of manually-written test cases to verify specific transformations, and a collection of real-world software.

We made a suite of manually verified test cases to ensure that our implementation makes correct transformations from compile-time variability to run-time variability, and never regresses from making correct transformations after making changes to the algorithm. The basic transformation rules are supported, and have test cases to verify them, as detailed in table 7.1 below.

Name	Description	Supported
`and_no.c`	`#if 1 && 0` should be removed	●
`assignments.c`	Several assignments split by `#ifdef`s (as in listing 5.3)	●
`assignments_different_comments.c`	Same as above, with differently placed comments	●
`big_no_ifdefs.c`	Code with various kinds of syntax, without `#ifdef`s	●
`blank.c`	Simple `main` function without code other than `return 0;`	●
`define_normal.c`	Testing token replacements with `#define` and `#undef`	●
`if_0_in_func.c`	Remove `#if 0`, replace its corresponsing `#else` by `if (1)`	●
`if_0_outside_func.c`	Remove `#if 0`, place its corresponsing `#else` at top level	●
`if_brace.c`	`#ifdef` around `if` conditional, with braces	●
`if_cplusplus_in_func.c`	`#if __cplusplus` should behave like `#if 0`	●
`if_cplusplus_outside_func.c`	`#if __cplusplus` should behave like `#if 0`	●
`if_defined.c`	Testing `#ifdef` variants (`#if defined(A)`, `#if defined A`)	●
`if_nobrace.c`	`#ifdef` around `if` conditional, without braces	●
`multiline_if.c`	Testing multiline `#if` expression with multiple features	●
`optional_braces.c`	`#ifdef` around opening and closing {} (as in listing 5.7)	●
`prefunc_enums_structs.c`	Testing various `#ifdef`s inside and around enums and structs	●
`simple_elif.c`	Testing an `#ifndef` in combination with an `#elif defined`	●
`simple_nest.c`	Testing nesting of `#ifdef`s	●
`string.c`	Testing compile-time string concatenation with `#ifdef`s	●

Table 7.1: The list of test cases

Of course, these test cases only verify that our tool gives a correct result for the test cases, and cannot give an accurate picture of "all of C". Indeed, due to the amount of time and work needed, we could not implement support for transforming many nontrivial cases. Our manually verified test cases are thus especially useful as a regression testing suite, allowing modification of the algorithm while ensuring that these modifications do not cause regressions, and then allowing an additional test case to be written to verify the new behavior. This process hardens the tool step by step.

Unfortunately, the range of possible C programs is very large, and many transformations are complicated to automate. Therefore, we did not eventually manage to make a perfect tool that can fully automatically transform most real-world software. However, this does mean that our tool could be used to reduce a programmer's workload, by transforming part of the directives where supported, while requiring manual changes for the remaining cases, based on our approach outlined in chapter 5.

Our second testing method, to give a more comprehensive indication of the state of the parser, is to run the tool on real-world software. For this purpose, as detailed in chapter 4, we collected 22 open-source projects. We ran the tool on all projects, and counted the number of files that the tool had, as well as did not have, success parsing and applying its transformations to. The tool could fail to transform files for a variety of reasons, for example due to non-termination of the transformation process, or an unexpected error condition. These problems could be caused either by cases that we deemed out of scope for this project, or by errors in the tool itself (caused by cases which we may have considered in scope, but caused an unexpected error

nonetheless). These files will be flagged to the user for further attention. For the successful files, we sampled five files per project, to manually verify if the transformations that were made are correct. This totals to 110 checked files. The results of running the tool on all projects, as well as the manual review of the samples, can be found in table 7.2.

Project	**Successful files**	**Manual review**
Apache HTTP server	377/549 (68.7%)	2/5
axTLS	58/60 (96.7%)	4/5
Busybox	658/744 (88.4%)	4/5
Emacs	373/564 (66.1%)	4/5
GIMP	3211/3240 (99.1%)	5/5
Gnumeric	592/607 (97.5%)	5/5
gnuplot	187/207 (90.3%)	5/5
Irssi	361/361 (100.0%)	5/5
libxml2	173/193 (89.6%)	3/5
lighttpd	174/193 (90.2%)	5/5
Linux	51493/52598 (97.9%)	3/5
mbedTLS	287/365 (78.6%)	1/5
MPSolve	182/183 (99.5%)	4/5
Netdata	446/455 (98.0%)	5/5
NGINX	332/343 (96.8%)	5/5
OpenSSL	1702/1769 (96.2%)	4/5
OpenVPN	245/252 (97.2%)	5/5
Parrot	253/265 (95.5%)	5/5
Redis	503/520 (96.7%)	5/5
SQLite	367/394 (93.1%)	2/5
uClibc-ng	3518/3604 (97.6%)	4/5
Vim	186/259 (71.8%)	2/5

Table 7.2: The findings running the tool on each complete project

As can be seen, when the goal is to transform all features from compile-time variability to run-time variability, most projects require some level of manual intervention. We have found one project where the tool could apply transformations to 100% of files. Many other projects are close, and require only a small number of files to be transformed manually. The lowest success percentage is 66.1%. However, the overall percentage of all files combined is 97.0%. The overall percentage of all files excluding those from Linux – an outlier in the number of files – is 93.8%. The success rate could be further improved by further development of the tool.

When it comes to manual verification of the transformations, the methodology was to compare the original versions of source files (with `#ifdef`s) with the transformed versions (with `if`s as substitutions). In doing this comparison, most transformations that were made seemed probably valid – that is,

not invalid in an apparent way. The files deemed invalid were mainly found to have one of the following problems:

Conditional `#defines`. This was the most common cause of problems, mostly concerning the case described in section 5.2.9, since this case was not supported yet. Often, a token can represent different values depending on a compile-time feature, and the given `#define` can appear multiple times in the source code. Commonly, this difference is about detection of a specific hardware architecture (such as 32-bit or 64-bit) or specific compilers and their features (such as Microsoft Visual C compilers before 2015 requiring `inline` to be substituted by `__inline`). The main focus to fix this issue would thus be to include better recognition for "environment" features such as hardware or compilers, and either process them with the correct values, or exclude those from transformations at all.

Global-scope data structures with varying contents. This case was found in a source file of Vim, and can be found in listing 7.1. As a bonus, this snippet also contains a conditional `#define`.

```
1  struct vimoption
2  {
3      char        *fullname;      // full option name
4      char        *shortname;     // permissible abbreviation
5      long_u      flags;          // see below
6      char_u      *var;           // global option: pointer to variable;
7                                  // window-local option: VAR_WIN;
8                                  // buffer-local option: global value
9      idopt_T     indir;          // global option: PV_NONE;
10                                 // local option: indirect option index
11     char_u      *def_val[2];    // default values for variable (vi and vim)
12 #ifdef FEAT_EVAL
13     sctx_T      script_ctx;     // script context where the option was last set
14 # define SCTX_INIT , {0, 0, 0, 1}
15 #else
16 # define SCTX_INIT
17 #endif
18 };
19
20 // ...
21
22 static struct vimoption options[] =
23 {
24     {"aleph",       "al",    P_NUM|P_VI_DEF|P_CURSWANT,
25 #ifdef FEAT_RIGHTLEFT
26                              (char_u *)&p_aleph, PV_NONE,
27 #else
28                              (char_u *)NULL, PV_NONE,
29 #endif
30                              {
31 #if defined(MSWIN) && !defined(FEAT_GUI_MSWIN)
32                              (char_u *)128L,
33 #else
34                              (char_u *)224L,
35 #endif
36                                                  (char_u *)0L} SCTX_INIT},
37 // ...
38 }
```

Listing 7.1: A sample of the conditional data structure part in the Vim project, `src/optiondefs.h`

What can be seen in this snippet is a struct (`vimoption`) which either has six or seven fields, depending on the feature `FEAT_EVAL`. Then, a static array of this structure is declared, of which one element is shown. Depending on the feature `FEAT_RIGHTLEFT`, the fourth and fifth attribute can be filled in with different values. Trivially shown, there is no possible configuration where these different values are both included in sequence, or where both are completely missing, so in the original code, this struct will always have

the correct number of attributes under any configuration.

Whenever data structures are initialized *within functions*, and such an initialization is split by preprocessor directives, we handle the problem by duplicating the entire initialization, and making it conditional in its entirity. Outside of functions however, it is not possible to make code conditional at run-time. Our automated approach outside of functions is generally to remove the `#ifdef`s and leave the code inside intact, because in many cases where `#ifdef`s are used outside of functions, the "optional" code does not directly affect configurations that do not have that feature enabled. For example, if an entire function is `#ifdef`'d out, that function still needs to be called from within another function, which thus needs `#ifdef`s to prevent calling a non-existent function.

In the case of this file from Vim, however, the result is a struct with eight values, instead of six or seven. Here, a solution to transform this code automatically would become quite complex: either this static initialization needs to be moved to code at runtime, or the array (in this case `options[]`) would need to be duplicated altogether and thus renamed throughout the entire program. We have deemed both methods out of scope due to the complexity involved.

Single-line comments at the end of `#defines`. This highlights a small oversight in our tool, where there can be a single-line comment at the end of a `#define`, which is then copied into code that has more tokens after that replaced identifier. The `//` comment will invalidate the other tokens, which is not intended behavior; normally during preprocessing, the comments should be removed first, then the `#define`d tokens filled in. However, we want to normally keep comments intact in the transformed code. A solution would thus be to either remove these comments specifically, insert a new line, or change the `//` comments into `/*` comments. An example of this problem can be found in Vim, shown in listing 7.2 and listing 7.3.

```
1  // values for vv_flags:
2  #define VV_COMPAT 1//compatible, also used without "v:"
3  #define VV_RO     2//read-only
4  #define VV_RO_SBX 4//read-only in the sandbox
5
6  // ...
7
8  {VV_NAME("count",VAR_NUMBER), VV_COMPAT+VV_RO},
9  {VV_NAME("count1",VAR_NUMBER), VV_RO},
10 {VV_NAME("prevcount",VAR_NUMBER), VV_RO},
11 {VV_NAME("errmsg",VAR_STRING), VV_COMPAT},
```

Listing 7.2: A sample of the comment error in the Vim project, `src/evalvars.h`

```
1  {VV_NAME("count",VAR_NUMBER), 1//compatible, also used without
       "v:"+2//read-only},
2  {VV_NAME("count1",VAR_NUMBER), 2//read-only},
3  {VV_NAME("prevcount",VAR_NUMBER), 2//read-only},
4  {VV_NAME("errmsg",VAR_STRING), 1//compatible, also used without "v:"},
```

Listing 7.3: The result of transforming this code

Minor previously-unknown bugs. A small number of errors were seemingly misidentifications of cases that were supposed to be already supported. An example is one file in Linux and one in BusyBox, where the contents of an `#if 0` are not removed. This would thus require an investigation into the circumstances where this problem occurs, and fix that problem while hardening the test cases as usual.

Overall, our tool could reduce the workload of a user tasked with converting preprocessor directives, and further development of the tool could focus on the above common problems to increase success rates. Yet, the goal of transforming all instances of compile-time variability to run-time variability in large C projects may remain a big undertaking, depending on the complexity of the code and other factors. We therefore believe a *staged configuration* approach ([6]) would be a promising alternative. If a limited number of features is chosen to be transformed, part of the difficulty and complexity is eliminated, which could drastically reduce the amount of errors, and the need for manual intervention. In particular, there could be a practical use for making only a handful of features dynamic, or modifying the code to remove the `#ifdefs` or the features altogether, in a tool-assisted way.

In our experience, creating a tool to transform compile-time variability us-

ing `#ifdef` directives to run-time variability is a much more difficult task than implementing a compiler or interpreter for the language, as the language and its preprocessor system was designed for. Mainly the possibility to intermingle the structure formed by preprocessor directives with the to-be-formed structure of the program code itself, is something that causes many problems. A regular compiler with a preprocessor does not encounter these problems, since all conditionals have only one final value at time of compilation, and the preprocessor can thus trivially simplify away all directives before the syntax of C needs to be considered. It is for this reason that there seems to be so little existing support for handling preprocessor directives in C code in a generalized way, or in any way where the preprocessing directives – or the editing decisions that they make – remain in the code in some form while the C code is also processed.

On paper, however, we do find it possible to get far when making hypothetical solutions to determine which transformations need to be made in which situations. This is namely what we have done in chapter 5. Therefore, we believe it is possible that future work can improve on this, given enough time.

7.2 Comparison with Hercules

For completeness, we have also looked into Hercules, and compared our tool against it, from a perspective of practicality to use for real-world software. As the only previously available tool for this problem that we are aware of, this will thus answer RQ2.3. Both tools have their own strengths and tradeoffs, and the tools have a different focus, meaning there are different scenarios where each tool does best.

For controlled, experimental pieces of code, we were able to see that Hercules supports some difficult cases well - it handles some nontrivial cases well that our tool does not. These cases are described in more detail in section 5.2. Compared to our tool, Hercules may have had a 'head start' in supporting complicated code since it was based on TypeChef, which is already a variability-aware parser. However, during our testing, for any code that had a large `#include` such as `<stdio.h>`, transformation would no longer be possible due to the included file needing to be transformed as well, which seemed too large to process within reasonable time. This problem might have been reduced with a fast enough computer, or by preparing special versions of header files that only contain the definitions that are actually used by the source code. The tool also only supports a single file to be supplied at a time, and it is much more strict on code definitions

being complete. Almost all real-world files were therefore hard for us to test on, and prevented us from making a comparison between Hercules and our tool for real-world projects. In comparison, our tool does not need complete code definitions, and leaves `#include` directives as-is, which means it does not suffer from the same problems. It also allows transforming of entire directories, so entire projects can be transformed.

In general, both our own tool, as well as Hercules, are stuck in a stage not far from proof-of-concept. This is telling for the difficulty of making such a tool fully functional – and we find that it is easy to underestimate the task at hand. A similar pattern was visible in other research projects related to preprocessor directives that we studied during this research. For example, a paper on the discipline of preprocessor directives by Liebig et al. ([13]) stated that there was ongoing work to create a tool to convert undisciplined directives to disciplined directives. We have not been able to find a finished version of this tool.

7.3 Performance behavior

RQ2.2 is about the performance behavior of our tool. To measure this, we have looked at the time needed to process all validation projects. The results can be found in table 7.3.

Project	SLOC	SLOC no-error	Total runtime (s)	Runtime no-error (s)	Lines/s	Lines/s no-error
Apache HTTP server	225948	76070	20.6	15.1	10987	5028
axTLS	19219	18304	18.1	3.5	1061	5161
Busybox	187772	107986	413.7	28.7	454	3762
Emacs	369106	47435	68.0	10.5	5428	4512
GIMP	805274	780045	133.5	115.5	6033	6756
Gnumeric	295551	273337	49.6	47.3	5964	5779
gnuplot	106792	66356	39.2	12.7	2726	5209
Irssi	63577	63577	9.6	9.6	6635	6635
libxml2	232194	91395	104.6	21.5	2220	4249
lighttpd	89143	61224	69.8	15.9	1277	3846
Linux	21075159	17716961	8304.6	5418.8	2538	3270
mbedTLS	114679	48643	154.8	13.2	741	3693
MPSolve	27051	26451	14.2	4.2	1903	6332
Netdata	269374	115323	49.9	23.6	5394	4890
NGINX	145674	134726	62.6	21.6	2328	6237
OpenSSL	475036	410771	150.9	94.3	3148	4355
OpenVPN	87873	74300	15.0	12.4	5876	6015
Parrot	116487	104610	30.0	24.1	3886	4341
Redis	156297	139438	44.7	31.9	3497	4365
SQLite	262504	202544	118.1	48.0	2223	4221
uClibc-ng	223275	188797	148.5	48.9	1504	3858
Vim	383980	164043	277.6	58.5	1383	2805

Table 7.3: The performance statistics of running the tool on each complete project

In this table, we list for each project the number of significant lines of code (SLOC), that is the total amount of lines of code that are not whitespace or a comment. We also list the total runtime of our tool for this project (in seconds), as well as a calculated number of lines per second that the tool can thus process.

The total runtime includes files which could not be processed, because they reached the processing timeout. Therefore, every such file induces a ten-second "penalty" to the total runtime, and the more timeouts are reached, the slower the tool seems for the files which are successful. Additionally, the more files that quickly caused a parsing error, the faster the tool seems, because no more time is spent on a file once an error is detected caused by that file. Therefore, to give a second indication, we also counted the total time the tool spent on files that it transformed successfully.

The numbers of lines per second are visualized in figure 7.4 below.

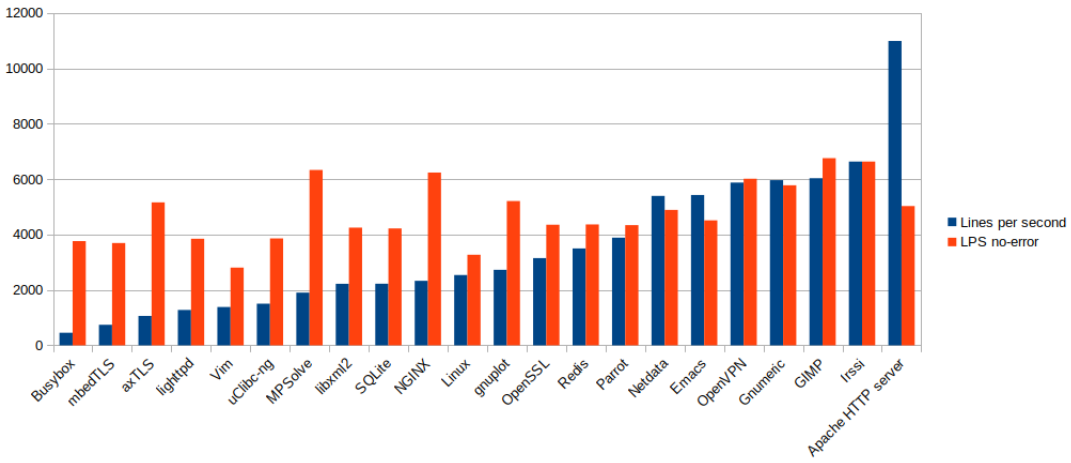

Figure 7.4: The number of lines per second for each project, with the blue bars counting all lines and the red bars only counting lines in successful files

What we can see is that there is a significant difference in the total runtimes. In lines per second, the Apache HTTP server was processed at the fastest rate, while Busybox was processed at the slowest rate. Looking at only successful files however, it becomes clear that 'erroneous' files heavily impacted these numbers: for Apache, 76,070 lines were processed in 15.1 seconds, but the tool took an additional 5.5 seconds for 149,878 unsuccessful lines, suggesting that unsuccessful files had high numbers of lines. For Busybox, 107,986 lines were processed in 28.7 seconds, but an additional 79,786 lines took 385 seconds, hinting at a high number of timeouts.

Looking at the speed of the tool for only successful files, the numbers are much closer together. The lowest number of lines per second is 2,805, and the highest is 6,756, which is 2.4 times as much. We believe the fluctuation in processing speed of successful files between projects can conceivably be explained by other factors that may differ between projects, such as the number of individual files and the nesting levels of `#ifdef`s.

Overall, we think our tool has acceptable performance behavior. The longest total processing time for a single project is 2 hours and 18 minutes, which – perhaps understandably – corresponds to the largest project we tested our tool on by a large factor, Linux. We will also note that these tests were run on a Dell Latitude E7250 from 2015 with an Intel Core i5-5300U processor, not a particularly high-end computer, so results will likely improve further on a more modern system. This means that, for transforming a project as large as Linux on a PC with moderate performance characteristics, one would have to wait two hours, but the tool can be left unattended and will

finish in a time that should not cause any user too many issues if planned ahead. The next longest runtime, Busybox, took less than seven minutes in total.

7.4 Additional insights

In our previous research [11], we discovered erroneous files across several projects. We categorized these errors into two categories: **testing code**, and **errors in production code**. The former category included files which were not meant for compilation, but rather for testing software that processes C/C++ code in some way, for example syntax highlighters. This kind of testing code is not always valid – on the contrary, we have shown purposeful errors that were clearly used to test whether software handled or reported such errors correctly. The second error category consisted of legitimate mistakes that had gone unnoticed, presumably since they would only generate warning messages in compilers.

It was easy to analyze the errors we encountered in our previous research, because our parser needed to operate only on preprocessor directives and comments, and could ignore most other syntax and language constructs. Due to the complexity of our current project goal, there are more files that we are not yet able to transform to the fullest extent, and it is therefore harder to determine which errors are caused by the incompleteness of our own tool, and which errors are genuine mistakes in real-world source code that has gone unnoticed during all compilations for any particular reason.

For RQ2.4, we were curious to see how our new tool handled the same files that our old tool reported as errors. Due to C++ being out of scope for this project, two C files remain: `util/check-format-test-positives.c` from OpenSSL, and `test/manual/etags/c-src/h.h` from Emacs. Interestingly, both files can be transformed without errors by our new tool. This can be explained by the fact that our new tool works in a different way, and is designed to work on source code that is well-formed. Due to the higher complexity of the new tool, it has less rigorous error checking in place to catch invalid source files. If a source file truly has invalid preprocessor directives, then it should also raise an error during normal compilation of either the original source files or the transformed versions.

For our new tool, it is more difficult to pinpoint additional insights about real-world projects in the same way, since unsuccessful transformations are more often caused by the tool not handling more complex situations. From validating the correctness of a random selection of files in projects, our

impression was that `#ifdef`s more commonly appear outside of or around functions than inside them.

7.5 Threats to validity

For the evaluation of our tool, we have used twenty two open-source projects and nineteen groups of regression test cases. Of course, this provides us with a vast amount of code, especially with the inclusion of the Linux kernel with its 21 million lines of code. However, given the popularity of the C language, there would always be room to test a wider range of projects. It is also left to be wondered if the practical choice of exclusively open-source projects has had an unforeseen influence on the results compared to if closed-source projects had been included – which might conceivably have different working cultures and different 'ways of programming'.

We have looked at the projects as they were at this point in time. Future changes to these programs, or future revisions of the C language itself, could create new incompatibilities and introduce new cases that need novel solutions to be made.

Our validation of correctness is based on testing. Proof of correctness could have been further improved with a formal mathematical proof, which was considered out of scope for this project. Regardless of whether there is a formal mathematical proof, there could be implementation bugs which only manifest themselves in certain circumstances.

As we will also highlight in the following chapter, it is very difficult to achieve the level of perfection needed to automatically transform sizeable programs in their entirity. Therefore, we have not been able to verify exhaustively what percentage of our transformed files can fully be compiled, do not introduce additional compiler warnings outside of our controlled test cases, and pass existing test suites belonging to the projects themselves.

Chapter 8

Discussion and future work

We have experienced that this is a more challenging problem to solve than originally expected. While high success rates sound good, the posed problem is one that, unfortunately, asks for a level of near-perfection. Any single error in a part of a program can cause the whole program to fail to compile. Most attempts to transform entire codebases will therefore still require some level of manual intervention to fix cases that were hard for us to automate. Still, we believe that our tool and approach could lighten the workload of any person wishing to make such transformations, and we hope it will enable further research and development to continue where we left off.

One improvement we would propose is for the tool to support a form of *staged configuration* ([6]), whereby the user can choose any given subset of features to transform. All other features would then be left as-is. Such an approach would make it much easier to convert an entire program to (partial) run-time variability, since it becomes much less overwhelming for a user to check which transformations were made, and for them to verify that these transformations are valid and work as expected. By transforming only a small number of features, the error rate would go down significantly, and any specific errors would be easy to recognize and can be corrected quickly. A program could thus be transformed gradually by iteratively feeding it through the tool, prioritizing the features that could benefit the most from being transformed to run-time variability first.

One problem that would ideally be solved in a tool like this is for it to keep the original code style intact as much as possible, and to mimic the existing style when inserting new code. From the perspective of transformation correctness and the workings of the code this is irrelevant, but from a practical perspective this can be a big roadblock to anyone attempting to use any

transformation tool to contribute back to the original codebase, since software maintainers often impose the requirement that the style of entire files not be suddenly changed. Our tool keeps track of which line each token was originally on, so it can attempt to keep the numbers of blank lines between each statement the same. There are still other consistency gains to be made, such as the difference between indentation with spaces or tabs, or the presence of spaces between specific pairs of tokens. One way to solve this is by using an automatic formatting tool after running our transformation tool, but this requires extra setup and attention to detail as to what the style rules of the specific project are. If the project follows the rules inconsistently, or follows certain unwritten rules, this may still require manual changes.

For a simpler and more achieveable goal, we have only focused on C in this study. Most of our results could also be applicable other programming languages that have `#ifdef` directives, particularly C++. An extension of the scope to C++ would thus further increase the practical applicability of this project.

Application of artificial intelligence and machine learning might also be rewarding when solving this problem. For example, supervised machine learning could involve presenting a model with examples of transformed compile-time variability, and reinforcement learning could involve rewarding a model for making code transformations that have equivalent behavior with the least amount of code changes possible – a learning process which could conceivably be automated. Generative artificial intelligence may also be used without the need for additional purpose-specific training. It will probably still take a long time before the output of AI-transformed source code can be trusted to be free of errors, but developments are being made quickly in recent times, so it may end up outperforming existing tools if applied well.

Chapter 9

Conclusion

In this research we have studied the transformation of C code from its usual compile-time variability form, containing `#ifdef` preprocessor directives, to a run-time variability form. We have given a definition of nontrivial cases for such transformations as well as an approach for transforming these cases programmatically, and we have developed a tool that aims to implement this approach.

We have identified the general challenges to overcome when transforming C code from compile-time variability to run-time variability. This includes to what extent compile-time variability can and should be transformed to equivalent `if` statements or requires other solutions, such as removing the `#ifdef` directives without replacement or leaving certain `#ifdef` directives in place. In particular, we have compiled a list of twelve cases, that we have each given specific transformation approaches for. We have had partial success implementing these approaches into a tool. About half of the cases we identified are fully or partially implemented, and correctness is verified with test cases. We have also tested our tool on the source code of 22 open-source projects, including Linux. These tests identify a somewhat promising success rate, with an overall percentage of 97.0% of all files that the tool could transform without running into issues. One project even had a success rate of 100%. Manual verification of a sample taken from these transformed files reveals that transformations in 87 out of 110 files were apparently valid, with varying success rates between different projects. Our tool seems to have acceptable performance characteristics. We have also compared our tool against Hercules, the only previously available tool for this problem we are aware of. Both tools have their strengths and tradeoffs, and we found that both tools are not yet ready for unattended transformations of arbitrary code at scale.

Bibliography

[1] pycparser. https://github.com/eliben/pycparser. Accessed: 2023-08-28.

[2] Typechef. https://github.com/ckaestne/TypeChef. Accessed: 2023-08-28.

[3] ISO/IEC 9899:1999. Programming languages – C. Standard, International Organization for Standardization, Geneva, CH, September 2007.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg, 2013.

[6] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software process: improvement and practice*, 10(2):143–169, 2005.

[7] Aleksandar S. Dimovski. Lifted termination analysis by abstract interpretation and its applications. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2021, page 96–109, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Paul Gazzillo and Robert Grimm. Superc: Parsing all of c by taming the preprocessor. *SIGPLAN Not.*, 47(6):323–334, jun 2012.

[9] Alexandru Iosif-Lazar, Ahmad Al-Sibahi, Aleksandar Dimovski, Juha Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. Experiences from designing and validating a software modernization transformation (e). pages 597–607, 11 2015.

[10] Eric Knauss. Constructive master's thesis work in industry: guidelines for applying design science research. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 110–121. IEEE, 2021.

[11] David Korsman. Tool support for detecting and analysing higher-order feature interactions in open-source software projects, February 2022. Research internship report, Radboud University.

[12] David Korsman, Carlos Diego N. Damasceno, and Daniel Strüber. A tool for analysing higher-order feature interactions in preprocessor annotations in c and c++ projects. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '22, page 70–73, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, page 191–202, New York, NY, USA, 2011. Association for Computing Machinery.

[14] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 483–494, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines, 2010.

[16] Mathijs T. W. Schuts, Rodin T. A. Aarssen, Paul M. Tielemans, and Jurgen J. Vinju. Large-scale semi-automated migration of legacy c/c++ test code. *Software: Practice and Experience*, 52(7):1543–1580, 2022.

[17] Alexander von Rhein. *Analysis Strategies for Configurable Systems*. PhD thesis, Universität Passau, 2016.

[18] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1, Part 2):125–145, 2016. Formal Methods for Software Product Line Engineering.