A Change-Centric Approach to Compile Configurable Systems with #ifdefs

Larissa Braz
Federal University of Campina Grande, Brazil
larissanadja@copin.ufcg.edu.br

Rohit Gheyi
Federal University of Campina Grande, Brazil
rohit@dsc.ufcg.edu.br

Melina Mongiovi
Federal University of Campina Grande, Brazil
melina@copin.ufcg.edu.br

Márcio Ribeiro
Federal University of Alagoas, Brazil
marcio@ic.ufal.br

Flávio Medeiros
Federal University of Campina Grande, Brazil
flaviomedeiros@copin.ufcg.edu.br

Leopoldo Teixeira
Federal University of Pernambuco, Brazil
lmt@cin.ufpe.br

Abstract

Configurable systems typically use `#ifdefs` to denote variability. Generating and compiling all configurations may be time-consuming. An alternative consists of using variability-aware parsers, such as TypeChef. However, they may not scale. In practice, compiling the complete systems may be costly. Therefore, developers can use sampling strategies to compile only a subset of the configurations. We propose a change-centric approach to compile configurable systems with `#ifdefs` by analyzing only configurations impacted by a code change (transformation). We implement it in a tool called CHECKCONFIGMX, which reports the new compilation errors introduced by the transformation. We perform an empirical study to evaluate 3,913 transformations applied to the 14 largest files of BusyBox, Apache HTTPD, and Expat configurable systems. CHECKCONFIGMX finds 595 compilation errors of 20 types introduced by 41 developers in 214 commits (5.46% of the analyzed transformations). In our study, it reduces by at least 50% (an average of 99%) the effort of evaluating the analyzed transformations by comparing with the exhaustive approach without considering a feature model. CHECKCONFIGMX may help developers to reduce compilation effort to evaluate fine-grained transformations applied to configurable systems with `#ifdefs`.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...$15.00
http://dx.doi.org/10.1145/2993236.2993250

Keywords Configurable Systems, `#ifdefs`, compilation errors

1. Introduction

Developers often implement configurable systems by using preprocessor conditional directives, such as the *#ifdef* macro, to allow defining parts of the source code as optional. There are reports of industrial systems [2] and examples of open source systems documented in detail [3] with a large number of configurations, such as BusyBox[1] and Apache.[2] However, due to the complexity of dealing with variability in C, developers may introduce compilation errors related to conditional directives when evolving configurable C systems [7, 14, 16]. These compilation errors may appear only in certain configurations, or in different ways in several configurations. Developers believe that configuration-related errors are harder to find and more critical than problems that appear in all configurations [15].

Most of the available compiler tools, such as GCC,[3] consider only one configuration at a time. Generating and compiling all configurations may be costly. A brute force strategy of generating, compiling, and testing all variants is not feasible for most configurable systems due to the high number of potential variants [12]. Therefore, in practice developers only check few configurations of the code or the default one [16]. Abal et al. [1] manually analyze commits of the Linux kernel repository and find a number of configuration-related bugs. They suggest the *one-disabled* sampling algorithm, which deactivates one preprocessor directive at a time. Still, manual analysis of configurable systems with a large number of macros may be costly and error-prone.

[1] `https://www.busybox.net/`
[2] `http://www.apache.org/`
[3] `https://gcc.gnu.org/`

```
#ifdef ENABLE_AUTH_MD5 && ENABLE_PAM
   struct pam_userinfo {
      const char *name;
      const char *pw;
   };
#endif
```

(a) Code snippet of the original `httpd.c` file.

```
#ifdef ENABLE_AUTH_MD5 && ENABLE_PAM
   struct pam_userinfo {
      const char *name;
      const char *pw;
   };
#endif
#ifdef ENABLE_PAM
   struct pam_userinfo userinfo;
#endif
```

(b) Code snippet of the modified `httpd.c` file.

Figure 1: Code snippets of consecutive commits of `httpd.c` file. This transformation introduces an incomplete type definition compilation error.

Variability-aware parsers, such as TypeChef [11], analyze the code by considering the complete configuration space. They generate abstract syntax trees enhanced with all variability information. However, the time-consuming setup and compilation process of these tools hinder the analysis of some projects. Medeiros et al. [14] propose an approach to improve these issues by generating stubs to replace the original types and macros from header files. To detect the configuration-related errors, they check all configurations by parsing the source code using a variability-aware parser, but ignoring file inclusions (`#include` directives). They analyze 41 releases of 8 configurable systems and find 24 syntax errors. Later, Medeiros et al. [16] present a strategy that considers only the header files of the target platform to minimize the setup problems of variability-aware tools. They instantiate it with TypeChef. They detect 16 configuration-related faults (2 undeclared variables and 14 undeclared functions) and 23 warnings related to configurability in 15 configurable systems. However, none of the previous approaches consider code changes in the analysis process to reduce the effort of evaluating configurable systems.

We propose a change-centric approach to compile configurable systems with `#ifdefs`[4] by conducting a per-file analysis. First, we receive a pair of files (original and modified) from a configurable system with `#ifdefs`. Then, we perform a change impact analysis to identify all macros impacted by a code change (transformation). Next, we generate and compile all possible impacted configurations in both versions of the file. We collect the set of compilation errors that appear only in the modified version of the configurable system, and categorize them into distinct errors. Finally, we report the set of different compilation errors and their related configurations to the developers. We implement this approach in an automated tool called CHECKCONFIGMX.

We conduct an empirical study to evaluate our approach. We evaluate 3,913 transformations applied to the six largest files of BusyBox, five largest files of Apache HTTPD, and three largest files of Expat. We detect 595 compilation errors in 214 pairs, which we categorize in 20 different kinds of errors, such as *incomplete type definition*. Moreover, we reduce by at least 50% (an average of 99%) the effort to evaluate the analyzed transformations by comparing with the exhaustive approach without considering a feature model [10]. The complete results are available online.[5] The main contributions of this paper are:

- an automated change-centric approach to compile configurable systems with `#ifdefs` (Section 3); and,
- an empirical study to evaluate the effectiveness and effort of our approach to detect compilation errors in configurable systems (Section 4).

2. Motivating Example

In this section, we show an example of a compilation error introduced by a fine-grained transformation applied to BusyBox. It is an open source software that has many Unix tools compacted in only one executable file. Figure 1a presents part of the `httpd.c` file (commit `d2277e2`), from BusyBox's repository that uses an `#ifdef` macro to declare the `struct pam_userinfo`. This `struct` is declared only when we enable the `ENABLE_AUTH_MD5` and `ENABLE_PAM` macros. In the next commit, the developer adds a code snippet the file that uses this `struct` when the `ENABLE_PAM` macro is enabled. However, the modified code does not compile when we disable `ENABLE_AUTH_MD5` and enable `ENABLE_PAM`. The compiler reports the following error message: `variable has incomplete type` "`struct pam_userinfo`." Figure 1b illustrates part of the modified code (commit `7291755`). This kind of error also occurs in other configurable systems (see Section 4.6).

We attempt to execute TypeChef on the real modified file. However, executing it is time-consuming since the modified `httpd.c` file has 18 macros. Therefore, variability-aware parsers may not scale to analyze this file since they may have a costly setup and compilation process. Moreover, we execute TypeChef on a toy example presented in Figure 1b. The tool does not detect the issue since it does not check this kind of error. Similarly, the approach proposed by Medeiros et al. [16] also does not detect this compilation error.

Compiling all configurations of a file, such as `httpd.c` may be costly even if we have a feature model that might

[4] Hereafter we refer to configurable systems as configurable systems with `#ifdefs`

[5] `http://www.dsc.ufcg.edu.br/~spg/checkconfigmx`

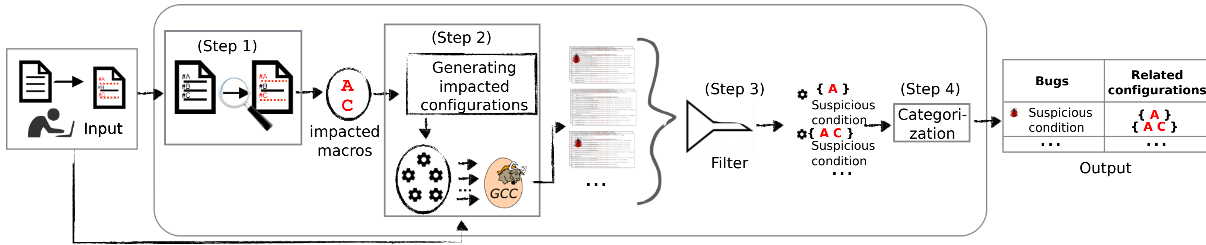

Figure 2: A change-centric approach to compile configurable systems with `#ifdefs`. The approach receives two versions of a C configurable system. It performs a change impact analysis to identify the macros impacted by the change (Step 1). Next, it generates all possible impacted configurations and compiles all of them using GCC (Step 2). Then, it identifies the compilation errors that occur only in the modified version of the system (Step 3). Finally, it categorizes the compilation errors introduced by the transformation into distinct ones (Step 4). The approach reports the result of the categorization.

forbid several configurations. We can use some sampling algorithms to check whether some configurations compile. For example, the `most-enabled-disabled` has an efficient balance between sample size and fault-detection capabilities under different assumptions [17]. However, by using it, we cannot detect this compilation error. The error is only exposed by a configuration which only `ENABLE_PAM` macro is enabled. In general, sampling algorithms that are not change-centric may waste time attempting to compile configurations that are not affected by the transformation, since they may not have new compilation errors. To minimize this problem, we propose a change-centric approach to compile configurable systems, which analyzes only the configurations that may have been impacted by the transformation (see Section 3).

3. A Change-Centric Approach to Compile Configurable Systems

In this section, we describe our approach for compiling configurable systems with `#ifdefs` by using change impact analysis.

3.1 Overview

Our approach receives two versions of a C configurable system file (original and modified). We assume that the system has no compilation errors in the original version and focus on finding the compilation errors introduced by a transformation. In our approach, we consider a transformation as a set of code changes in a configurable system file.

First, our approach performs a change impact analysis that identifies the macros impacted by the transformation (Step 1). It compares the text (diff) between the original and modified files. The set of impacted macros contains all macros that appear in the diff or that can enable the modified code. Next, it generates all possible configurations considering the impacted macros. The approach compiles each selected configuration on both versions of the system using GCC (Step 2). Then, it filters the compilation errors that occur only in the modified version (Step 3). Finally, the approach categorizes the compilation errors by grouping them based on the similarity of their messages (Step 4). Figure 2 illustrates the main steps of our approach.

We implement our approach in a tool called CHECKCONFIGMX. CHECKCONFIGMX uses the Git *diff* command to assist our change impact analysis. This command returns the code changes between the original and the modified files (transformation). CHECKCONFIGMX executes GCC to preprocess and compile the configurable systems.

3.2 Change Impact Analysis

The first step of our approach consists of identifying the set of macros impacted by the change. From two versions of a file, our change impact analysis performs a textual diff to identify the code snippets modified by the transformation. Next, it searches for macros in the textual diff. We consider that a macro is directly impacted by the change when the textual diff contains it. A macro is indirectly impacted when it enables the compilation of the modified code.

Figure 3 presents two code snippets of consecutive commits in the `httpd.c` file, from the BusyBox's repository. We rename the macros to simplify the explanation. Figure 3a illustrates part of the original code (commit d2277e2). It declares a function `check_user_passwd` under the `M1` macro. The file also contains other macros, such as `M4` and `M5`. Developers changed this file by adding code under the `M1` macro. Figure 3b illustrates part of the modified file (commit 7291755). Our approach compares the original and modified files and identifies the textual diff between them (transformation). In this example, the transformation adds the code in lines 2 to 4 and 7 to 12 of the modified code (Figure 3b). We consider these code snippets as impacted by the change. The set of directly impacted macros of this example consists only of the `M2` and `M3` macros, since they are part of the code changes (textual diff). Notice that they are nested with the `M1` macro. Therefore, `M1` is indirectly impacted by the change since it enables or disables the impacted code. The complete set of impacted macros is: `{M1, M2, M3}`.

```
1  #ifdef M1
2     static int check_user_passwd() { }
3  #endif
4  #ifdef M4 && M5
5     static int miniHttpd(int server) { }
6  #endif
```

(a) Code snippet of the original `httpd.c` file.

```
1  #ifdef M1
2  #ifdef M2 && M3
3     static int pam_talker() { }
4  #endif
5     static int check_user_passwd() {
6  #ifdef M3
7        struct pam_conv conv_info =
8           { &pam_talker};
9  #endif
10    }
11 #endif
12 #ifdef M4 && M5
13    static int miniHttpd(int server) { }
14 #endif
```

(b) Code snippet of the modified `httpd.c` file.

Figure 3: Code snippets of consecutive commits of `httpd.c` file. This transformation introduces an undeclared variable compilation error.

3.3 Selecting Impacted Configurations

In this step, we yield all possible impacted configurations from the set of macros impacted by a transformation identified by the previous step. We compile only the impacted configurations since only these configurations may have new compilation errors. As a result, we can save time and effort by avoiding compiling non-impacted configurations.

First, we use a combinatorial algorithm to find all combinations of the impacted macros. For example, consider the previous example (Figure 3). The modified file contains the `M1`, `M2`, `M3`, `M4`, and `M5` macros. The algorithm receives as input the impacted macros (`M1`, `M2` and `M3`) and returns the set of configurations that enables their combinations. It returns the following set of configurations: `{M1}, {M1 M2}, {M1 M3}, {M2}, {M2, M3}, {M3}, {M1, M2, M3}, {}`. For simplicity, we represent a configuration by its enabled macros. Therefore, the configuration `{M1}` is equal to `{M1, !M2, !M3, !M4, !M5}`.

Next, for each configuration found by the combinatorial algorithm, we make a copy of it and enable all non-impacted macros. For example, for the configuration `{M1}`, we create a new configuration `{M1, M4, M5}` and add in the set of impacted configurations. Therefore, our approach identifies the following set of impacted configurations for the previous example: `{M1}, {M1, M2}, {M1, M3}, {M2}, {M2, M3}, {M3}, {M1, M2, M3}, {}, {M1, M4, M5}, {M1, M2, M4, M5}, {M1, M3, M4, M5}, {M2, M4, M5}, {M2, M3, M4, M5}, {M3, M4, M5}, {M1, M2, M3, M4, M5}, {M4, M5}`.

The number of impacted configurations is directly proportional to the number of impacted macros. Our approach analyzes $O(2^{i+1})$ configurations, where i is the number of impacted macros. It may have a better performance when analyzing fine-grained transformations applied to highly configurable systems, since they tend to have few impacted macros among a large set of macros.

We compile both file versions for each impacted configuration. Since we identify the impacted configurations based on the impacted macros of the modified version of the file, the original version may not have the same macros. To compile a configuration in the original code, we need to automatically disable the macros that exist only in the modified code.

Figure 3 presents part of the `httpd.c` file with five macros. However, the original file has 18 macros. We generate only 16 configurations (99% of reduction) to compile each version of the file by using our approach. For this example, we find a compilation error when `M1` and `M3` are enabled and `M2` is disabled.

3.4 Filtering and Categorizing Compilation Errors

In the filter step (Step 3), we automatically select the compilation errors that appear only in the modified version of the file. We filter the error messages according to their templates. Each message contains: the kind of the error, which includes the code element that caused it; and, the line of the error (number and contents). Notice that, the transformation may add or remove some lines of code before the error, regarding the code location. Therefore, we filter the messages by removing the location of the error (line and column) since the same compilation error may occur in both versions of a system file (original and modified), but in different lines of code. Our goal consists of identifying only the new compilation errors introduced by the transformation. We do not focus on identifying the pre-existing errors.

In the last step, we automatically categorize the filtered compilation errors messages into distinct ones by analyzing if they are related to the same fault (Step 4). We consider that two error messages are related to the same fault if they contain the same kind of error and elements. We analyze the error messages based on their template and ignore the location of the error (line and column) and the contents of the error line (code statement) to categorize the messages. For example, GCC reports the following message when it tries to compile the program presented in Figure 1b: `httpd.c:11:17: error:` "`use of undeclared variable 'pam_talker' {&pam_talker};`". We consider the following error message is related to the same compilation error of the previous one: `httpd.c:22:1:`

error: “use of undeclared variable ‘pam_talker’ pam_talker = 0;”. Notice that, the differences are the line of the error and the contents of the error. Finally, if the same compilation error occurs in one or more configurations, we classify them as only one error. Our categorization reports to the user the set of compilation errors and their related configurations identified by our approach.

4. Evaluation

In this section, we describe the evaluation of our approach on 3,913 transformations applied to BusyBox, Apache, and Expat files.

4.1 Definition

The goal of our experiment consists of analyzing CHECKCONFIGMX for the purpose of evaluating code changes with respect to finding compilation errors from the point of view of researchers in the context of configurable systems. We address the following research questions:

- RQ$_1$: What kinds of compilation errors does CHECKCONFIGMX find?
  We identify the kinds of compilation errors detected by CHECKCONFIGMX.
- RQ$_2$: How much effort does CHECKCONFIGMX reduce to find compilation errors in terms of analyzed configurations?
  For each analyzed transformation, we compare the number of impacted configurations identified by our approach with the number of possible configurations.
- RQ$_3$: What is the rate of transformations that introduce at least one compilation error?
  We measure the rate of transformations in which CHECKCONFIGMX detects at least one introduced compilation.
- RQ$_4$: What is the rate of compilation errors that occur in transformations with no impacted macros?
  We measure the number of impacted macros and the number of introduced compilation errors.
- RQ$_5$: What is the CHECKCONFIGMX’s rate of false positives?
  For each compilation error identified by CHECKCONFIGMX, we manually analyze if it is a false positive.

4.2 Planning

In this section, we describe the subjects used in the experiment and its instrumentation.

Subjects Selection: We consider a pair of consecutive commits from a repository as a transformation. We analyze transformations of the Git repository history of the six largest files of BusyBox, five largest files of Apache HTTPD, and three largest files of Expat (based on the size of their last commits). We assume that largest files may have more interesting scenarios to analyze and can better evaluate our approach. CHECKCONFIGMX can evaluate transformations with any number of impacted macros. However, we do not focus on pairs with more than four impacted macros in our study since evaluating them may be costly. Still, we evaluate 96.01% of the transformations.

CS	File	Pairs	Dev.	Average	
				Macros	Diff
BusyBox	ash	543	32	49.24	103.94
	httpd	247	15	15.91	44.35
	hush	696	18	29.19	49.35
	modutils-24	24	4	24.75	21.50
	ntpd	107	18	7	48.64
	vi	223	22	17	47
Apache HTTPD	core	577	52	15	17
	event	147	19	26.27	24.89
	mod_include	353	39	5.62	40.53
	mod_rewrite	427	51	31.20	17
	proxy_util	524	33	3.68	28.18
Expat	xmlparse	166	6	9.81	52.88
	xmltok	36	5	4.22	19.89
	xmltok_impl	14	3	4.21	47.43
	Total	**4,084**	**317**	**17.36**	**40.18**

Table 1: Configurable systems analyzed in our study. CS = Configurable System; File = Name of the analyzed file; Pairs = Analyzed pairs; Dev. = Developers that performed commits; Macros = Average macros in the analyzed file; Diff = Average lines of textual diff between the pairs.

BusyBox replaces basic functions over 300 common commands, such as killall. We analyze 1,837 transformations (code changes between commits) applied to the BusyBox files from Apr/01 to Nov/15. The analyzed files have an average of 29.94 distinct macros per commit. The highest number of macros in a file is 58, while the lowest is 1. Apache HTTPD is the core technology of the Apache Software Foundation, responsible for more than a dozen projects involving web-based transmission technologies, data processing, and execution of distributed applications. We analyze 2,023 transformations applied to the Apache files from Aug/99 to Jun/16. The analyzed files have an average of 11 macros per commit. The highest number of macros in a file is 28, while the lowest is 1. Expat is an XML parser library written in C in which an application registers handlers for things the parser might find in the XML document. We analyze 216 transformations applied to the Expat files from Sep/2000 to Feb/10. The analyzed files have an average of 8.49 distinct macros per commit, the highest number of macros in a file is 12, while the lowest is 3. Table 1 details the subjects that we evaluate in this study.

Instrumentation: We execute the experiment on a Mac OSX Yosemite (2.6 GHz, i5 and 8GB RAM). We use GCC 4.2.1 (Apple LLVM 6.0), Java 1.8.0_45, and GNU Bash 4.3.33 (x86_64-apple-darwin13.4.0). We evaluate only the transformations that impact at most four macros.

4.3 Results

CHECKCONFIGMX evaluated a total of 3,913 transformations applied to BusyBox, Apache, and Expat. The modified files of the transformations have a total of 77,364 macros, which our approach identified 3,508 of them as impacted. Considering the exhaustive approach of compiling all possible configurations, without considering a feature model, CHECKCONFIGMX reduced the effort to evaluate the an-

CS	File	Errors		
		Pairs	Total	Dev.
BusyBox	ash	27	89	11
	httpd	18	63	
	hush	16	31	
	modutils-24	8	33	
	ntpd	6	5	
	vi	14	26	
Apache HTTPD	core	79	176	29
	event	6	17	
	mod_include	26	85	
	mod_rewrite	3	17	
	proxy_util	8	50	
Expat	xmlparse	3	3	1
	Total	**214**	**595**	**41**

Table 2: Compilation errors detected by CHECKCONFIGMX; CS = Configurable System; File = Name of the analyzed file; Pairs = Pairs that introduced compilation errors; Dev. = Developers that performed the commits that introduced the errors.

CS	File	Filter		Categorization	
		Configs.	Total	Total	Real
BusyBox	ash	284	1,689	145	89
	httpd	51	302	110	63
	hush	93	255	36	31
	modutils-24	33	344	35	33
	ntpd	14	22	9	5
	vi	48	207	41	26
Apache HTTPD	core	165	943	199	181
	event	7	165	17	17
	mod_include	85	735	105	85
	mod_rewrite	12	302	17	12
	proxy_util	42	350	64	50
Expat	xmlparse	6	13	5	3
	xmltok	-	-	-	-
	xmltok_impl	12	312	159	0
	Total	**852**	**5,639**	**942**	**595**

Table 3: Total of compilation errors found in configurable system by Steps 3 and 4; CS = Configurable System; File = Name of the analyzed file; Filter/Configs. = Impacted configurations with compilation errors; Filter/Total = Introduced compilation error messages; Categorization/Total = Different compilation errors; Categorization/Real = Real compilation errors after a manual analysis removing false positives.

alyzed transformations by at least 50%. We observed this through a comparison with the exhaustive approach without considering a feature model. CHECKCONFIGMX found 776 configurations (6.02% of the impacted configurations) with at least one compilation error. It found 5,639 compilation error messages, which we categorize into 942 different errors. However, 36.83% of them are false positives. For example, we found false positives related to missed libraries in old commits. So, our technique found 595 real different errors.

We classify the 595 bugs found in 20 kinds of compilation errors. The developers of the analyzed systems introduced these compilation errors in a total of 214 transformations. Table 2 shows the total of pairs that introduced real compilation errors and the total of developers that performed the commits of the transformations. The kind of error with the highest occurrence was the use of undeclared variables (49.7%), followed by no member in `struct` (31.7%) and incomplete definitions of types (3.1%). Tables 3 and 4 summarize the compilation errors that we have found in this experiment.

The transformations that introduced at least one compilation error have a textual diff average of 12.92 LOC. We found the largest transformation in terms of modified lines of code in the `mod_include.c` of Apache HTTPD (commits `f5858d9` and `e56d601`). This transformation had a textual diff of 1,735 LOC. It introduced five compilation errors on four configurations. We found 20 transformations that introduced errors with a diff of only one LOC. The developers introduced an average of 2.4 compilation errors per transformation. We also found 132 transformations that had no impacted macros, but introduced 307 errors. For example, a transformation applied to `proxy_util.c` of Apache HTTPD (commits `935de30` and `e63bcb2`) had no impacted macros, but it introduced 31 different compilation errors.

4.4 Discussion

In this section, we discuss issues related to the compilation errors found, the change impact analysis and filter steps, false positives, false negatives, and the time to evaluate the transformations.

Compilation Errors: We found that most of the compilation errors that developers introduce are caused by the use of undeclared variables (49.7%), followed by the use of undeclared members of structures (31.7%) and incomplete definition of type (3.1%). The first and third most introduced errors are related to structs. Table 4 shows the kinds and total of compilation errors found by our approach.

We found that a transformation applied to `httpd.c` of BusyBox (commits `d2277e2` and `7291755`) introduced two compilation errors. Figures 1 and 3 show code snippets of the original and modified versions of this file. Medeiros et al. [16] extended TypeChef and evaluated the modified file. They found only one of the errors, as they do not detect compilation errors related to *incomplete type definition*. Moreover, we executed TypeChef on small toy examples with each kind of error found in our study. We found that it does not detect the following kinds of errors: *incomplete type definition*, *unknown type name*, *undeclared label*, *function cannot return array type*, *continue statement not in loop statement*, and *address of bit-field requested*.

The sampling approaches one-enabled and one-disabled do not detect the compilation error introduced by the transformation presented in Figure 3. They compile only the configurations in which one macro is enabled or disabled. In this example, we had to enable two macros to detect the error.

We found a transformation applied to `core.c` of Apache HTTPD (commits `dfd16b1` and `a677072`) that introduced a compilation error related to a bit-field requested address. Figure 4a shows a code snippet of the original file. It declares the `struct conn_rec`, which contains the signed int `double_reverse` (lines 1 to 3). Figure 4b shows a code snippet of the modified file. It tries to access the address of `double_reverse`, which is a bit-field of size two (line 9). This attempt of access causes a compilation error because the

Kinds of Compilation Errors	BusyBox	Apache HTTPD	Expat	Total
use of undeclared variable	171	123	2	296
no member in struct	24	165	-	189
member reference base type is not a structure or union	-	19	-	19
incomplete definition of type	16	2		18
invalid operands to binary expression	12	-	-	12
too few arguments to function call	-	12	-	12
unknown type name	-	11	-	11
use of undeclared label	9	-	-	9
expression is not assignable	6	-	-	6
conflicting types	-	4	-	4
subscripted value is not an array, pointer, or vector	4	-	-	4
are not pointers to compatible types	2	-	1	3
reference to local variable declared in enclosing function	-	3	-	3
address of bit-field requested	-	2	-	2
called object type is not a function or function pointer	2	-	-	2
continue statement not in loop statement	-	1	-	1
function cannot return array type	-	1	-	1
non-void function should return a value	-	1	-	1
not a function or function pointer	-	1	-	1
type name requires a specifier or qualifier	-	1	-	1
Total	**246**	**346**	**3**	**595**

Table 4: Kinds of compilation errors found in our study.

memory can be accessed byte by byte (8 bits) only. The modified version of the file contains 5,596 LOC and 14 macros. The transformation has 131 LOC of textual diff and impacts no macro. Developers fixed this error 44 days later after 3 more commits performed on this file. The number of macros in a file may increase the code complexity, which may lead developers to postpone the compilation of the configurations. TypeChef does not detect this kind of error.

Change Impact Analysis: We found 163 transformations (3.99%) that impacted more than four macros. Among them, 66 pairs have five impacted macros, 22 pairs have six impacted macros, and 20 pairs have seven impacted macros. We found a pair of `ash.c` file of BusyBox with 54 impacted macros (commits `cb81e64` and `c470f44`). We found that 132 of the transformations that introcuded compilation errors do not have impacted macros, and 52 of them have one impacted macro. Together, they represent 77.77% of the transformations that introduced at least one compilation error. We could detect them by combining the one-enabled and all-disabled-enabled sampling algorithms. However, in our study, the one-enabled approach would require compiling an average of 20 configurations for each transformation with at most one impacted macro, while CHECKCONFIGMX compiled an average of 3.42 configurations for each one.

The repositories of the analyzed systems have 4,076 commits and we analyzed 3,913 (96.01%) pairs of them. The goal of our approach consists of reducing the effort of compiling all possible configurations by compiling only the impacted ones. Therefore, it is useful to analyze fine-grained transformations. We consider that a fine-grained transformation typically impacts a small number of macros and consequently, a small number of configurations. Nevertheless, CHECKCONFIGMX can also help developers to evaluate large files with several impacted macros.

We focused on analyzing transformations that impacted at most four macros since compiling more than 32 configurations may be costly. For example, for the purpose of testing our approach in transformations that involve more than four impacted macros, we analyzed one of them applied to `event.c` of Apache HTTPD (commits `11da82` and `273b7aa`). The modified file has 27 macros, but only 7 of them are impacted. We generated and compiled 256 configurations, and none of them introduced compilation errors. CHECKCONFIGMX took 65.62s to evaluate this transformation. Step 2 spent 58.63s to compile the configurations, which is almost 30 times higher than the CHECKCONFIGMX's average time of this step during our experiment. We can similarly evaluate the other transformations under this scenario. Developers should indicate the maximum number of macros that they would like to consider.

Filter: After compiling all configurations of each transformation, we performed the filter step and found 5,639 compilation error messages in 776 configurations of 214 transformations. We found 720 new messages in a transformation applied to `ash.c` of BusyBox (commits `59f351c` and `92e13c2`). This number is much higher than the average compilation errors messages (7.29) of the analyzed transformations with errors. We categorized them into 10 compilation errors, of which 1 is false positive. We attempted to execute TypeChef in the modified `ash.c` file (92e13c2) of this transformation. However, we got a heap space error. Similarly, we could not execute TypeChef in other files due to heap space and missing libraries problems.

False Positives: We manually investigated all results of our automated categorization. We found that 347 (36.83%) of the compilation errors found by Step 3 were false positives. Some problems, such as missing libraries and compilers incompatibility may cause these errors. The subjects that we analyzed are more than 10 years old and some of the required libraries and compilers are not available anymore. Therefore, using our approach in a controlled environment, with all required libraries and compilers available, would minimize these issues.

We implemented our approach as a per-file analysis. We considered only the files analyzed in this experiment and their dependencies, instead of the whole system. This may have led to false positives. For example, we found that all of the 159 compilation errors detected by CHECKCONFIGMX in two transformations of the `xmltok_impl.c` file of Expat are false positives. The errors are related to the use of undeclared variables. At first, CHECKCONFIGMX considered them as real errors since the file has no `include headers`. However, we investigated the commits and found that `xmlparse.c` includes `ascii.h` and `xmltok_impl.c`. Moreover, `ascii.h` defines the variables used in the `xmltok_impl.c`, which caused the compilation errors. Therefore, these errors do not

```
1  struct conn_rec {
2     signed int double_reverse:2;
3  };
4  void do_double_reverse (conn_rec *conn){...}
5  char * ap_get_remote_host(conn_rec *conn, ...)
6  {
7     if (the.) {
8        do_double_reverse(conn)
9      }
10 }
11 // This file contains 31 blocks of #ifdefs and #
       ifs
12 // with 14 different macros
```

(a) Code snippet of the original `core.c` file.

```
1  struct conn_rec {
2     signed int double_reverse:2;
3  };
4  void do_double_reverse (int *double_reverse)
      {...}
5  char * ap_get_remote_host(conn_rec *conn){
6     if (...) {
7        do_double_reverse(&conn->double_reverse)
8     }
9  }
10 // This file contains 31 blocks of #ifdefs and #
       ifs
11 // with 14 different macros
```

(b) Code snippet of the modified `core.c` file.

Figure 4: Code snippets of consecutive commits of `core.c` file. This transformation introduces a bit-field requested address compilation error.

appear in a global analysis since the compilation scope would include the file that define the variables.

False Negatives: We investigated whether our approach may have false negatives by using mutation testing. We manually applied 70 mutations of 7 kinds in the last 10 versions of the six largest BusyBox's repository files. We manually checked that they are not equivalent mutants. We used the mutation score to measure the CHECKCONFIGMX's rate of false negatives since mutation testing results can be used as a reference for new test cases or to measure quality on existing tests [9]. We used the following mutation operators: remove semicolon; remove declaration; change pointer operator; duplicate declaration; change declaration type; remove field; and, remove return type. We included some defects that previous approaches [1, 17] found, but we did not find in our study, such as remove semicolon. Previous approaches [1, 16, 17] found bugs in real configurable systems caused by changes similar to these mutation operators.

CHECKCONFIGMX killed all mutants. This result increases our confidence that our approach may not report false negatives. We also measured the effort reduction to evaluate the transformations by using our tool instead of an exhaustive approach, such as TypeChef. CHECKCONFIGMX identified 70 impacted macros, among the set of 1,287 macros. It evaluated a median of 2 impacted configurations per transformation. The median of all possible configurations per transformation, without considering a feature model, is 8,388,608. So, our approach reduced by at least 50% (an average of 99%) the effort to evaluate the transformations when compared with the exhaustive approach. Moreover, CHECKCONFIGMX may be useful to reduce the number of compiled configurations (effort) in a configurable system by comparing with sampling approaches. For example, in this study CHECKCONFIGMX compiled around five configurations on average. The analyzed transformations applied to `ash.c` of BusyBox had up to four impacted macros and CHECKCONFIGMX compiled 32 configurations in the worst cases. The `ash.c` has around 49 macros on average (Table 1). Therefore, the one-enabled sampling approach compiles 49 configurations to evaluate this file.

Our set of mutation operators may not be representative of all the errors that actually happen in practice. However, they were useful to show that our approach detected all errors introduced by the mutations.

Time: We measured the total time of each step performed by CHECKCONFIGMX during this study. CHECKCONFIGMX took on average 6s to evaluate each transformation applied to the systems. Developers can execute CHECKCONFIGMX whenever they modify a of a configurable system with `#ifdefs`. In a few seconds, CHECKCONFIGMX can report to them the set of possible compilation errors introduced by the code changes.

For some files, the average time was higher than for all files. For example, the average time to evaluate the transformations applied to `modutils-24.c` of BusyBox and `mod_include.c` of Apache was 9s. We found that `modutils-24.c` has four transformations with 16 or 32 impacted configurations. Moreover, we found compilation errors in some of them. Therefore, the number of analyzed configurations increased the time to filter the errors. The time of the filter step is smaller in other transformations with 16 or 32 impacted configurations since CHECKCONFIGMX did not find compilation errors introduced by them. The most time-consuming analysis was in a transformation applied to `hush.c` of BusyBox (commits `68d5cb5` and `3eab24e`). Our approach took 20.28s to evaluate it. This transformation impacted 32 configurations, in which our approach took 14s to compile them all. The average time to evaluate all transformations applied to `hush.c` file was not affected by this transformation because it has a total of 696 transformations.

In general, the most time-consuming step is compiling the configurations (Step 2). We use GCC to compile them and it took on average 2.5s to compile all impacted configurations of the analyzed transformations. On the other hand, the fastest step is the change impact analysis (Step 1). Our approach took on average 0.07s to identify the impacted macros. Finally, it took on average 1.5s and 1.9s to perform the steps of filter and categorization, respectively. Table 5 shows the average time for analyzing all transformations.

CS	File	Time (s)				
		Impact Analysis	GCC	Filter	Categ.	Total
BusyBox	ash	0.02	2.02	0.69	2.49	5.23
	httpd	0.05	2.13	1.21	1.35	4.75
	hush	0.05	2.62	1.54	1.66	5.88
	modutils-24	0.05	3.72	2.03	3.29	9.10
	ntpd	0.04	1.91	1.27	2.51	5.74
	vi	0.05	1.77	1.13	2.68	5.63
Apache HTTPD	core	0.05	2.50	1.68	1.26	5.49
	event	0.11	2.81	2.13	1.63	6.69
	mod_include	0.30	4.30	2.44	1.77	9.82
	mod_rewrite	0.07	1.77	1.19	1.30	4.34
	proxy_util	0.06	3.13	2.10	2.53	7.84
Expat	xmlparse	0.06	1.56	1.03	1.30	3.96
	xmltok	0.04	1.76	1.16	1.74	4.71
	xmltok_impl	0.06	1.44	1.14	1.29	3.94
	Average	**0.07**	**2.52**	**1.54**	**1.91**	**6.06**

Table 5: Averages of the runtime execution of CHECKCONFIGMX; CS = Configurable System; File = The analyzed file; Time (s) = Average times for: Impact Analysis (Step 1); GCC (Step 2); Filter (Step 3); Categ. (Step 4); and, Total.

4.5 Threats to Validity

A feature model [10] has a significant role in the compilation of a Software Product Line [6, 19], as each feature may require specific configurations. However, we did not consider them in our evaluation. By considering feature models we may have fewer compilation errors. Still, other configurable systems may have similar scenarios in practice and CHECKCONFIGMX can be useful to evaluate them.

CHECKCONFIGMX compiles one file per analysis. Therefore, we may have false positives and negatives, since a local change may have global impact. We can adjust our approach to consider the whole system. Since we evaluated old commits, we did not identify all required dependencies, such as external libraries. Moreover, the developers may have used an older version of a C compiler. These issues may lead our approach to detect some false positives. We could minimize this threat by using our approach in a controlled environment, with all libraries and compilers available. The implementation of compilers may vary according to the operating system [22]. We found different compilation errors by executing the same GCC version in Ubuntu 14.04 and Mac OS X Yosemite on `httpd.c` of BusyBox. Each compiler detected errors that the other did not detect. The internal platform dependencies of GCC may have bugs [22]. We may have detected some compilation errors in this study that the developers using another operating system cannot detect.

Our change impact analysis does not consider the C programming language structures, which may cause false positives or negatives. For example, if a developer removes a variable that is used under a macro, CHECKCONFIGMX will not try to compile and detect it if this macro is not impacted. Although our change impact analysis is simple, we detected a number of compilation errors. We can use other approaches to change impact analysis to improve this step.

4.6 Answers to the Research Questions

Next, we summarize the answers of our research questions.

- RQ_1**:** What kinds of compilation errors does CHECKCONFIGMX find?
  CHECKCONFIGMX found 595 errors introduced by the transformations. Table 4 presents all 20 kinds of compilation errors that we found in 214 (5.46%) transformations of BusyBox, Apache HTTPD, and Expat files.
- RQ_2**:** How much effort does CHECKCONFIGMX reduce to find compilation errors in terms of analyzed configurations?
  CHECKCONFIGMX reduced by at least 50% (an average of 99%) the effort to evaluate the analyzed transformations by comparing with the exhaustive approach without considering a feature model.
- RQ_3**:** What is the rate of transformations that introduce at least one compilation error?
  CHECKCONFIGMX found that 5.46% of the analyzed transformations introduced at least one compilation error. The tool found 113 transformations that introduced one compilation error; 34 transformations that introduced two compilation errors; 17 transformations that introduced three compilation errors; 14 transformations that introduced four compilation errors; and, 35 transformations that introduced at least five compilation errors.
- RQ_4**:** What is the rate of compilation errors that occur in transformations with no impacted macros?
  CHECKCONFIGMX found 307 compilation errors in 132 transformations with no impacted macros. This indicates that 55.86% of the compilation errors would be identified by the all-disabled-enabled approach. CHECKCONFIGMX found the following number of transformations that introduced compilation errors: 48 with one impacted macro; 27 with two impacted macros; 11 with three impacted macros; and, 8 with four impacted macros.
- RQ_5**:** What is the CHECKCONFIGMX's rate of false positives?
  CHECKCONFIGMX found that 347 (36.83%) of the compilation errors that Step 4 found are false positives. Using our approach in a controlled environment would solve these issues.

5. Related Work

Kästner et al. [11] propose TypeChef, a variability-aware parser, which analyzes all possible configurations and performs type checking. It parses code with conditional compilation, using SAT solvers for decisions during the parsing process. By using the abstract syntax tree enhanced with all variability information, they can search for configuration-related bugs in all configurations. Such tools analyze the complete configuration space, considering file inclusion and macro expansions Our approach differs from those approaches in

the sense that we focus only on compiling the configurations impacted by the change.

Medeiros et al. [14] propose an approach to improve the TypeChef's scalability problem. They generate stubs to replace the original types and macros from header files. They found 24 configuration-related syntax errors in 40 configurable systems. However, their approach also analyzes all possible configurations. Although we do not identify syntax errors in our study, we can detect them, as we show in the mutation testing analysis. Moreover, our approach compiles only the impacted configurations.

Later, Medeiros et al. [16] conduct a study to better understand undeclared/unused variables and functions related to configurability. They propose a strategy that considers only one configuration of the header files to scale their study and minimize possible setup problems. They implement checkers for each kind of error that they want to detect. They detected 2 undeclared variables, 14 undeclared functions and 23 warnings related to configurability. We found 20 kinds of errors, in which one of them was related to *an undeclared variable*. A transformation applied to `xmlparser.c` of Expat introduced this kind of error. Although they have also evaluate the file (commit `79aa349`), they do not detect this compilation error as we do, since manually analyzing a number of configurations and error messages may be error prone. Our approach compiles only impacted configurations. Moreover, we can adapt Steps 2 and 3 of our approach to evaluate the warning messages that they consider.

Previous studies have proposed various strategies for dealing with configuration spaces [1, 13, 21]. Medeiros et al. [17] compare 10 sampling algorithms regarding their fault-detection capability and sample sets size. They found that algorithms with the largest sample sizes detected the most faults and the statement-coverage algorithm detected the lowest number of faults. They also found that simple algorithms with small sample sets, such as most-enabled-disabled, are the most efficient in most contexts. However, these approaches may not evaluate some configurations that are affected by the change and may not detect some errors introduced by them. For example, we found compilation errors that the most-enabled-disabled cannot detect (Section 2).

Qu et al. [20] present an approach that selects configurations for regression testing using slicing-based code change impact analysis [4]. They evaluated two open source C systems and a large industrial C/C++ system. The approach discarded up to 60% of the configurations as redundant. Different from them, we analyze only the macros impacted by a transformation, as our approach aims at reducing the effort of compiling configurable systems with `#ifdefs` by using change impact analysis. Although our change impact analysis is simpler, we found 595 compilation errors introduced by 214 transformations applied to configurable systems.

In the context of the C language, there are some studies proposing tools that perform static and dynamic analysis to find bugs, such as memory leaks. Evans [8] presents Splint to statically check C programs for security vulnerabilities and coding mistakes. Novark et al. [5] propose Plug to detect memory leaks in C and C++ programs. Nethercote and Seward [18] present Valgrind, a framework for building dynamic analysis tools. They can detect, for example, threading bugs. Current, we use GCC in our approach to detect compilation errors. We can replace GCC (Step 2) to some of those tools to detect other kinds of bugs.

6. Conclusions

In this work, we propose a change-centric approach to compile configurable systems with `#ifdefs`. From two versions of a file, it identifies the macros impacted by the change. Next, our approach selects the set of impacted configurations and compiles each one in both versions of the file. It reports a set of compilation errors that occur only in the modified file. We implemented our approach in an automated tool called CHECKCONFIGMX.

We analyzed 3,913 transformations applied to 14 files of the Git repositories of BusyBox, Apache HTTPD and Expat. We found 595 compilation errors of 20 kinds introduced by 214 transformations. Most of them are related to *undeclared variables* (49.74%) and *no member in struct* (31.76%). We manually analyzed each error and found that 36.83% of them are false positives. In general, missing libraries and compilers incompatibility caused these errors. Using our approach in a controlled environment, with all required libraries and compilers available, would solve these issues. CHECKCONFIGMX reduced by at least 50% the effort to evaluate the analyzed transformations by comparing with the exhaustive approach without considering a feature model.

By not considering the impact of the transformations applied to a configurable system, exhaustive approaches may have to analyze all possible configurations. The evaluation results show evidence that CHECKCONFIGMX can be useful in analyzing code changes with up to four impacted macros applied to files with a large number of macros. Still, for small files with few macros our tool may have little gain by comparing with exhaustive approaches. A developer can perform changes in the code and use CHECKCONFIGMX to assess whether they introduced compilation errors. Usually, it takes a few seconds to analyze fine-grained transformations. CHECKCONFIGMX reports only the new introduced compilation errors.

Acknowledgments

We would like to thank Christian Kästner, Sven Apel and the anonymous reviewers. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4, 477943/2013-6 and 460883/2014-3, and CAPES grant 175956.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 421–432, 2014.

[2] T. Berger, R. Rublack, D. Nair, J. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*, pages 1–7, 2013.

[3] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, pages 1611–1640, 2013.

[4] S. Bohner and A. Robert. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996.

[5] M. Bond and K. McKinley. Tolerating memory leaks. In G. E. Harris, editor, *Proceedings of the 14th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 109–126, 2008.

[6] P. Clements and L. Northrop. *Software product lines: Practices and patterns*. Addison-Wesley, 2009.

[7] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, pages 1146–1170, 2002.

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

[9] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 654–665, 2014.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[11] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 17th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 805–824, 2011.

[12] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *Transactions on Software Engineering and Methodology*, 21(3):1–39, 2012.

[13] D. Kuhn, D. Wallace, and A. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

[14] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, pages 75–84, 2013.

[15] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, pages 495–518, 2015.

[16] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proceedings of the 15th International Conference on Generative Programming: Concepts and Experiences*, pages 35–44, 2015.

[17] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 643–654, 2016.

[18] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th Programming Language Design and Implementation*, pages 89–100, 2007.

[19] K. Pohl, G. Bockle, and F. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[20] X. Qu, M. Acharya, and B. Robinson. Configuration selection using code change impact analysis for regression testing. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 129–138. IEEE Computer Society, 2012.

[21] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, pages 1–5, 2011.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, pages 283–294, 2011.