Computer Languages, Systems & Structures 54 (2018) 427–450

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

A change-aware per-file analysis to compile configurable systems with `#ifdefs`

Larissa Braz[a,*], Rohit Gheyi[a], Melina Mongiovi[a], Márcio Ribeiro[b], Flávio Medeiros[c], Leopoldo Teixeira[d], Sabrina Souto[e]

[a] Federal University of Campina Grande, Campina Grande, Paraíba, Brazil
[b] Federal University of Alagoas, Maceió, Alagoas, Brazil
[c] Federal Institute of Alagoas, Maceió, Alagoas, Brazil
[d] Federal University of Pernambuco, Recife, Pernambuco, Brazil
[e] State University of Paraíba, Campina Grande, Paraíba, Brazil

ARTICLE INFO

Article history:
Received 16 March 2017
Revised 9 January 2018
Accepted 10 January 2018
Available online 2 February 2018

Keywords:
Compilation
`#ifdef`
Configurable systems
Impact analysis

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. In practice, compiling complete systems may be costly. Therefore, developers use sampling strategies to compile only a subset of the configurations. In our previous work, we propose a change-aware per-file analysis 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 extend our previous work by performing an empirical study to evaluate 7,891 transformations applied to 32 files of configurable systems such as Linux and OpenSSL. CheckConfigMX finds 1,699 compilation errors of 34 types introduced by 155 distinct developers in 756 commits (9.19% of the analyzed transformations). In our study, the tool reduces by at least 50% (an average of 99%) the effort of evaluating the analyzed transformations compared to the exhaustive approach and without considering a feature model. In addition, we also evaluate the effectiveness of CheckConfigMX by using mutation testing. We generate 11,229 mutants by applying eight mutant operators to some evaluated files. CheckConfigMX kills all mutants. Therefore, it may help developers to reduce compilation effort to evaluate fine-grained transformations applied to configurable systems with `#ifdefs`.

1. Introduction

Nowadays, many software systems, such as Linux[1] and Ghostscript,[2] are configurable [1]. Configurability is a way to create a collection of similar software systems from a set of software components using common means of production

* Corresponding author.
E-mail addresses: larissanadja@copin.ufcg.edu.br (L. Braz), rohit@dsc.ufcg.edu.br (R. Gheyi), melina@computacao.ufcg.edu.br (M. Mongiovi), marcio@ic.ufal.br (M. Ribeiro), flavio.medeiros@ifal.edu.br (F. Medeiros), lmt@cin.ufpe.br (L. Teixeira), sabrinadfs@gmail.com (S. Souto).

[1] https://www.linux.com/.
[2] http://www.ghostscript.com/.

https://doi.org/10.1016/j.cl.2018.01.002
1477-8424/

[2]. The configuration options of these systems can be combined, each one with different features and effects on quality attributes, creating a family of similar products.

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 [3] and examples of open source systems documented in detail [4] with a large number of configurations, such as BusyBox[3] and Apache.[4] 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 [5–7]. 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 [8]. This scenario may become even more difficult in undisciplined `#ifdefs` [9].

Most of the available compiler tools, such as GCC,[5] consider only one configuration at a time. A brute force strategy of generating, compiling, and testing all configurations (variants) may be costly and not feasible for most configurable systems due to the high number of potential variants [10]. Therefore, in practice developers only check few configurations of the code or the default one [6]. Abal et al. [11] 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 might remain costly and error-prone.

Variability-aware parsers, such as TypeChef [12], 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. Gazzillo and Grimm [13] state that TypeChef misses several interactions between preprocessor features. Moreover, its parser is limited since developers need to reengineer the grammars with the tool combinators and to correctly employ the various join combinators. They propose SuperC, another variability-aware parser. This parser is faster than TypeChef, but it does not perform type-checking analysis. Medeiros et al. [5] 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). Medeiros et al. [5] analyze 41 releases of 8 configurable systems and find 24 syntax errors. They detect files that contain errors in all commits. Later, Medeiros et al. [6] 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 and 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 to reduce the effort of evaluating configurable systems.

In our previous work [14], we propose a change-aware per-file analysis to compile configurable systems with `#ifdefs` by conducting a per-file analysis. First, we receive a pair of files (original and modified) from a configurable system implemented with `#ifdefs`. Then, we perform a change impact analysis to identify all macros impacted by a code change (hereafter, 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.

In this article, we extend our previous work [14] by including 21 new files of seven different configurable systems in our evaluation. CheckConfigMX evaluates 7,891 transformations applied to six files from BusyBox, five files from Apache HTTPD, and three files from Expat, CVS, M4, OpenSSL, Gzip, Linux, and Gnuplot each. CheckConfigMX can evaluate transformations with any number of impacted macros. However, in our study, we do not focus on transformations with more than four impacted macros. CheckConfigMX detects 1,699 compilation errors in 756 transformations, which we categorize in 34 different kinds of errors, such as *incomplete type definition*. In total, we find 14 new kinds of compilation errors. Moreover, CheckConfigMX reduces by at least 50% (on average 99%) the effort to evaluate the analyzed transformations, compared to the exhaustive approach and without considering a feature model [15] (see Section 4).

Moreover, in this article, we evaluate the effectiveness of CheckConfigMX by analyzing the false negatives of our approach using mutation testing. We apply eight mutant operators, such as *rename* and *change type of declaration*, in 1,380 file versions among the set of files analyzed during our study. In total, we have 11,229 mutants. CheckConfigMX kills all of them. By not considering the transformation applied to the configurable system, the previous approaches may have to analyze a higher number of configurations. The complete results of both studies are available online.[6]

In summary, the main contributions of this article are:

- An extended empirical study to evaluate the effectiveness and effort of our approach to detect compilation errors in configurable systems. We included 21 new files of seven different configurable systems in our experiment; CheckConfigMX evaluates 4,306 new transformations; and finds 14 new kinds of compilation errors (see Section 4).

[3] https://www.busybox.net/.
[4] http://www.apache.org/.
[5] https://gcc.gnu.org/.
[6] http://www.dsc.ufcg.edu.br/~spg/comlan17.

```
1 #ifdef AUTH_MD5 && PAM
2 struct pam_user{
3   const char *name;
4   const char *pw;
5 };
6 #endif
```

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

```
1 #ifdef AUTH_MD5 && PAM
2 struct pam_user{
3   const char *name;
4   const char *pw;
5 };
6 #endif
7 #ifdef PAM
8 struct pam_user user;
9 #endif
```

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

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

- An extended empirical study using mutation testing to evaluate the effectiveness of our approach to detect compilation errors in configurable systems. We apply eight mutation operators to 11,229 file versions of the analyzed systems (Section 5).

We organize the remainder of this article as follows. In Section 2, we show a motivating example. Section 3 describes our change-aware per-file analysis to compile configurable systems with `#ifdefs`. Section 4 details an empirical study that evaluates the effectiveness of our approach in finding compilation errors. Following, Section 5 discusses a second empirical study that evaluates the effectiveness of CheckConfigMX through mutation test. Finally, we present the related work in Section 6, and the concluding remarks in Section 7.

2. Motivating example

In this section, we show an example of a compilation error introduced by a fine-grained transformation applied to BusyBox, an open source software that has many Unix tools compacted in only one executable file. Listing 1a presents part of the `httpd.c` file from BusyBox's repository (commit `d2277e2`), which uses an `#ifdef` macro to declare the `struct pam_user`. This `struct` is declared only when the macros `AUTH_MD5` and `PAM` are enabled. In the next commit, the developer changes the code to use the `struct` when `PAM` is enabled. However, the modified code does not compile when we disable `AUTH_MD5` and enable `PAM`, since in this case it does not declare the struct. The compiler reports the following error message: `variable has incomplete type` "`struct pam_user`." Listing 1b illustrates part of the modified code (commit `7291755`). This kind of error also occurs in other configurable systems (see Section 4.6).

We could use variability-aware tools, such as TypeChef to detect such problems. However, executing it on this specific file is time-consuming (machine time). Since the `httpd.c` file has 18 macros. After more than eight hours of execution, we got a heap space issue. Therefore, variability-aware tools may not scale to analyze this file since they may have a costly setup and compilation process. Moreover, we also execute TypeChef on a toy example based in Listing 1b. Although TypeChef has checkers for some kinds of type errors, but not all. For instance, the tool does not detect incomplete definition of type error. Similarly, the approach proposed by Medeiros et al. [6] also does not detect this compilation error.

Feature models may forbid several configurations due to constraints among interacting directives, such as establishing that two macros cannot be enabled at the same time. However, compiling all configurations of a file, such as `httpd.c`, may be costly even if we have a feature model. As a solution, we can use some sampling algorithms to check whether only some configurations compile. For example, the `most-enabled-disabled` sampling algorithm, which uses only the configuration with all macros enabled and the one with all macros disabled, has an efficient balance between sample size and fault-detection capabilities under different assumptions [16]. However, by using it, we still cannot detect the compilation error above. The error is only exposed by configurations in which `PAM` is enabled and `AUTH_MD5` is disabled. In general, sampling algorithms that are not change-aware 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-aware per-file analysis to compile configurable systems, which analyzes the configurations that may have been impacted by the transformation (see Section 3).

3. A change-aware per-file analysis to compile configurable systems

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

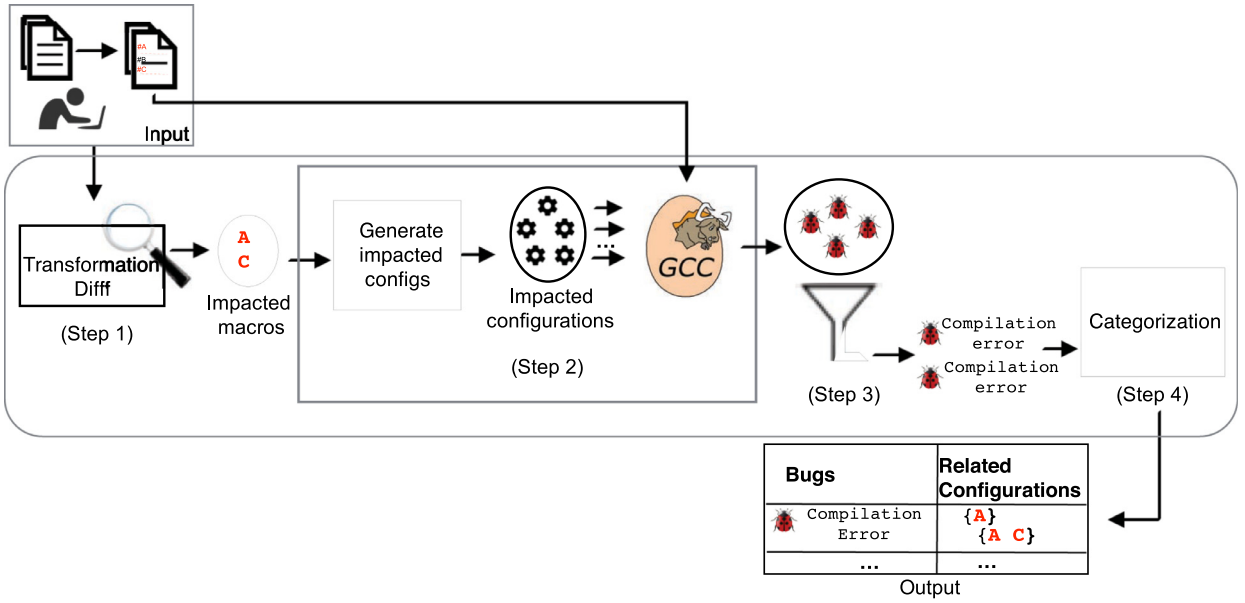

Fig. 1. A change-aware per-file analysis to compile configurable systems with `#ifdefs`. The approach receives two versions of a C configurable system. It performs a change impact analysis to identify the macros impacted by the change (Step 1). Next, it generates all possible impacted configurations 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.

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 textual 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). Fig. 1 illustrates the main steps of our approach.

We implement our approach in a tool called CheckConfigMX, which uses the Git *diff* command to assist our change impact analysis. This command returns the 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 is not part of the modified code but enables its compilation.

Listing 2 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. Listing 2a 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. Listing 2b illustrates part of the modified file (commit 7291755). Our approach compares the original and modified files and identifies the textual diff between them. In this example, the transformation adds the code in lines 2 to 9 of Listing 2b. We consider these code snippets as impacted by the change. The set of directly impacted macros of this example consists of the `M2` and `M3` macros, since they are part of the transformation. Notice that they are nested within 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(){}
3  #endif
4  #ifdef M4 && M5
5  static int httpd(...){}
6  #endif
```

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

```
1  #ifdef M1
2  #ifdef M2 && M3
3  static int tlkr(){}
4  #endif
5  static int check_user(){
6  #ifdef M3
7    struct conv info = {&tlkr};
8  #endif
9  }
10 #endif
11 #ifdef M4 && M5
12 static int httpd(...){}
13 #endif
```

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

Listing 2. Code snippets of consecutive commits of `httpd.c` file. This transformation introduces an undeclared variable compilation error.

Table 1
Set of partial impacted configurations.

Partial impacted configurations			
Config 1	!M1	!M2	!M3
Config 2	M1	!M2	!M3
Config 3	!M1	M2	!M3
Config 4	!M1	!M2	M3
Config 5	!M1	!M2	M3
Config 6	!M1	M2	!M3
Config 7	M1	!M2	!M3
Config 8	M1	M2	M3

3.3. Selecting impacted configurations

In this step, we yield all possible impacted configurations from the set of impacted macros. We only compile the impacted configurations since only they 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. As an example, consider the previous example (Listing 2). 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 set of partial configurations presented by Table 1.

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 it to the set of impacted configurations. Therefore, for the previous example, our approach identifies the set of impacted configurations presented by Table 2.

The number of impacted configurations is directly proportional to the number of impacted macros. After identifying and combining the impacted macros, for each combination we generate other configuration concatenating all non-impacted macros enabled. This process duplicates the set of generated configurations. Our approach analyzes $O(2^{i+1})$ configurations, where i represents the number of impacted macros. In practice, 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 larger 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. When compiling the files, we consider their dependencies defined by `#include` directives.

Table 2
Set of impacted configurations.

Impacted configurations					
Config 1	!M1	!M2	!M3	!M4	!M5
Config 2	M1	!M2	!M3	!M4	!M5
Config 3	!M1	M2	!M3	!M4	!M5
Config 4	!M1	!M2	M3	!M4	!M5
Config 5	!M1	!M2	M3	!M4	!M5
Config 6	!M1	M2	!M3	!M4	!M5
Config 7	M1	!M2	!M3	!M4	!M5
Config 8	M1	M2	M3	!M4	!M5
Config 9	!M1	!M2	!M3	M4	M5
Config 10	M1	!M2	!M3	M4	M5
Config 11	!M1	M2	!M3	M4	M5
Config 12	!M1	!M2	M3	M4	M5
Config 13	!M1	!M2	M3	M4	M5
Config 14	!M1	M2	!M3	M4	M5
Config 15	M1	!M2	!M3	M4	M5
Config 16	M1	M2	M3	M4	M5

Listing 2 presents part of the `httpd.c` file with five macros. However, the original file has 18 macros. We generate only 16 configurations (over 99% reduction when compared to the brute force strategy and without considering feature models) 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 using regular expressions. Each message contains: the error kind, which includes the code element that caused it; and, the line 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 error location (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 pre-existing errors. To use other compilers on Step 2, for example Clang, we have to change the regular expression according to the output message of the new tool.

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 error messages based on their template and ignore the error location (line and column) and the line contents (code statement) to categorize the messages. For example, GCC reports the following message when it tries to compile the program presented in Listing 2b: `httpd.c:11:17: error:` "`use of undeclared variable 'tlkr' {&tlkr}`." We consider the following error message related to the same compilation error: `httpd.c:22:1: error:` "`use of undeclared variable 'tlkr' tlkr = 0`." Notice that the differences are the error line and contents. Finally, if the same compilation error occurs in one or more configurations, we categorize 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. Study I: identifying compilation errors using CheckConfigMX

In this section, we describe the first evaluation of our approach on 7,891 transformations applied to Apache, BusyBox, CVS, Expat, Linux, M4, Gzip, OpenSSL, and Gnuplot. First, we present the study definition (Section 4.1) and planning (Section 4.2). Sections 4.3 and 4.4 present and discuss the results, respectively. Finally, Section 4.5 describes some threats to validity and Section 4.6 summarizes the main findings.

4.1. Definition

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 does CheckConfigMX reduce the effort needed 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 number 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 number 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 CheckConfigMX's frequency 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 (Section 4.2.1) used in the study and its setup (Section 4.2.2).

4.2.1. Subjects selection

For each analyzed file, we obtain the list of commits in the system repository that changed it. Next, we match the consecutive commits as transformations. We analyze transformations of the Git repository history of six BusyBox files, five Apache HTTPD files, and three files from Expat, Linux, CVS, M4, Gzip, and OpenSSL, each (based on the size of their last commits). In addition, we analyze transformations applied to three random files of Gnuplot. CheckConfigMX can evaluate transformations with any number of impacted macros. However, we do not focus on transformations with more than four impacted macros in our study since evaluating them may be costly. Moreover, CheckConfigMX evaluates 96% of all possible transformations of the analyzed files. CheckConfigMX evaluates 4,002 transformations with no impacted macros, 2,585 with one impacted macro, 907 with two impacted macros, and 270 and 139 with three and four impacted macros, respectively. Table 3 details the subjects evaluated in this study.

The analyzed configurable systems are free and open source. Linux Kernel is a Unix-like computer operating system. CheckConfigMX analyzes 1,093 transformations (code changes between commits) applied by 1,168 commits performed to the Linux kernel files from April'05 to February'17. OpenSSL provides a toolkit for the Transport Layer Security and Secure Sockets Layer protocols. It is also a general-purpose cryptography library. CheckConfigMX analyzes 903 transformations applied by 917 commits performed to the OpenSSL files from December'98 to January'17. BusyBox replaces basic functions over 300 common commands of Linux, such as `killall`. CheckConfigMX analyzes 1,687 transformations applied by 1,840 commits performed to the BusyBox files from April'01 to November'15. 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 distributed application execution. CheckConfigMX analyzes all transformations applied by 2,028 commits performed to the Apache files from August'99 to June'16.

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. CheckConfigMX analyzes 146 transformations applied by 216 commits performed to the Expat files from September'00 to February'10. CVS is a version control system, an important component of Source Configuration Management. CheckConfigMX analyzes 453 transformations applied by 455 commits of the CVS files from August'87 to February'17. GNU M4 is an implementation of the traditional Unix macro processor. It has built-in functions for including files, running shell commands, doing arithmetic, among others. CheckConfigMX analyzes 376 transformations applied by 384 commits performed to the M4 files from February'00 to January'17. Gzip (GNU zip) is a compression utility. CheckConfigMX analyzes 82 transformations applied by 83 commits performed to the M4 files from August'93 to January'17. Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. CheckConfigMX analyzes 1,140 transformations applied to the Gnuplot from April'99 to August'17.

4.2.2. Setup

We execute the study on a Mac OSX Yosemite (2.6 GHz, i5 and 8 GB RAM). CheckConfigMX uses GCC 4.2.1 (Apple LLVM 6.0) with the `-ferror-limit=0` parameter, Java 1.8.0_45, and GNU Bash 4.3.33 (x86_64-apple-darwin13.4.0). CheckConfigMX evaluates transformations that impact at most four macros. We use the commands `cat` and `wc -l` to measure the total of lines of code (LOC) of the files.

4.3. Results

CheckConfigMX evaluated a total of 7,891 transformations applied to Apache, BusyBox, CVS, Expat, Linux, M4, Gzip, OpenSSL, and Gnuplot. The modified files of the transformations have a total of 120,243 macros. Our approach identifies that only 7,571 of them as impacted by the transformations. Considering the exhaustive approach of compiling all possible configurations, without considering a feature model, CheckConfigMX reduced the effort to evaluate the analyzed transformations by at least 50%. We observed this through a comparison with the exhaustive approach and without considering a feature model. CheckConfigMX found 2,834 configurations (9.62% of the impacted configurations) with at least one compilation error. It found 22,874 compilation error messages, and categorized them into 2,878 different errors. However, 32.47% of them are false positives (not real compilation errors). For example, we manually found false positives related to missing libraries in old commits. So, our approach found 1,699 real different errors.

CheckConfigMX categorizes the 1,699 bugs found in 34 kinds of compilation errors. The developers of the analyzed systems introduced these compilation errors in a total of 756 transformations. Table 4 shows the total of transformations

Table 3
Configurable systems analyzed in our study. Pairs/First and Last = Date of first and last analyzed commit, respectively; Pairs/Total = Number of pairs; Pairs/Dev. = Developers that performed commits; Average/Diff (LOC) = Average of textual diff between the commits in terms of lines of code.

Configurable System	File	Pairs				Diff (LOC)	Macros			Impacted Macros		
		First	Last	Total	Dev.		Min	Max	Avg	Min	Max	Avg
BusyBox	ash.c	Jun/06	Sep/15	543	32	103.94	31	75	49.24	0	27	1.41
	httpd.c	Jan/03	Aug/15	247	15	44.35	13	30	15.91	0	13	1.91
	hush.c	Apr/01	Sep/15	696	18	49.35	4	63	29.19	0	19	1.44
	modutils-24.c	Sep/08	Mar/15	24	4	21.50	24	30	24.75	0	6	1.29
	ntpd.c	Nov/09	Aug/15	107	18	48.64	0	12	7	0	6	0.42
	vi.c	Apr/01	Jul/15	223	22	47	13	31	17	0	15	1.64
Apache HTTPD	core.c	Mar/01	Jun/16	577	52	17	7	27	15	0	12	0.17
	event.c	Mar/09	Jun/15	147	19	24.89	24	30	26.27	0	2	0.19
	mod_include.c	Aug/04	Jan/15	353	39	40.53	3	17	5.62	0	4	0.22
	mod_rewrite.c	Aug/99	Jun/16	427	51	17	6	18	31.20	0	4	0.18
	proxy_util.c	Aug/99	Jun/16	524	33	28.18	2	6	3.68	0	5	0.19
Expat	xmlparse.c	Sep/00	Feb/10	166	6	52.88	7	16	9.81	0	5	0.72
	xmltok.c	Aug/00	Jan/09	36	5	19.89	3	8	4.22	0	3	0.72
	xmltok_impl.c	Aug/00	Jun/08	14	3	47.43	4	7	4.21	0	2	0.64
Linux	scrub.c	May/11	Dec/16	233	37	37.66	1	1	1	0	1	1
	slub.c	May/07	Feb/17	706	123	33.38	7	18	12.90	0	7	1.04
	security.c	Apr/05	Aug/16	229	52	18.78	0	7	5.12	0	6	0.44
CVS	apprentice.c	Aug/87	Feb/17	255	4	27.87	0	12	8.06	0	4	1.30
	compress.c	Jan/91	Feb/17	102	5	19.02	0	12	7.77	0	6	1.90
	magic.c	Mar/03	Jul/16	98	5	13.48	7	14	10.39	0	4	1.85
M4	freeze.c	Feb/00	Jan/17	106	4	26.31	0	1	0.25	0	1	0.03
	input.c	Oct/07	Jan/17	125	4	47.64	0	1	1.20	0	2	0.24
	main.c	Oct/07	Jan/17	153	4	15.73	2	5	2.32	0	2	0.10
OpenSSL	s_client.c	Dec/98	Jan/15	233	24	21.98	7	29	15.02	0	21	0.76
	ssl_ciph.c	Dec/98	Jan/17	229	30	27.83	5	18	12.16	0	17	0.55
	t1_lib.c	Dec/98	Jan/17	455	37	40.25	8	8	11.91	0	21	1.44
Gzip	deflate.c	Aug/93	Jan/17	26	4	7.61	8	9	5.26	0	2	0.46
	inflate.c	Aug/93	Jan/17	28	6	7.14	2	4	2.67	0	1	0.07
	util.c	Aug/93	Jan/17	29	4	15.51	2	10	6.24	0	5	0.34
Gnuplot	axis.c	May/00	Aug/17	238	7	25.21	0	3	1.61	0	2	1.07
	set.c	Apr/99	Jun/17	559	9	31.24	3	17	11.24	0	6	1.32
	term.c	Apr/99	Jun/17	331	10	21.97	22	32	28.39	0	27	1.54
				8,219	**686**	**31.28**	**0**	**78**	**12.05**	**0**	**8.06**	**0.83**

that introduced real compilation errors and the total of developers that performed the transformations. The compilation error with the highest occurrence was the use of undeclared variables (41%), followed by no member in `struct` (22%) and incomplete definitions of types (12%). In this study, CheckConfigMX identified the compilation errors listed in Table 5.

The transformations that introduced at least one compilation error have a textual diff average of 12.92 LOC. The largest transformation analyzed by CheckConfigMX, in terms of LOC, occurred 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. CheckConfigMX 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. 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.

Table 4
Total of compilation errors found in configurable system by Steps 3 (filter) and 4 (categorization); Filter/Configs. = Number of impacted configurations with compilation errors; Filter/Total = Number of introduced compilation error messages; Categ./Total = Number of different compilation errors; Errors/Pairs = Number of transformations that introduced compilation errors; Errors/Total = Number of real compilation errors after removing false positives; Errors/Dev. = Number of developers that introduced the errors.

Configurable System	File	Filter		Categ.	Errors		
		Configs.	Total	Total	Pairs	Total	Dev.
BusyBox	ash.c	284	1,689	145	27	89	11
	httpd.c	51	302	110	18	63	
	hush.c	93	255	36	16	31	
	modutils-24.c	33	344	35	8	33	
	ntpd.c	14	22	9	6	5	
	vi.c	48	207	41	14	26	
Apache HTTPD	core.c	165	943	199	79	176	29
	event.c	7	165	17	6	17	
	mod_include.c	85	735	105	26	85	
	mod_rewrite.c	12	302	17	3	17	
	proxy_util.c	42	350	64	8	50	
Expat	xmlparse.c	6	13	5	3	3	1
	xmltok.c	-	-	-	-	-	
	xmltok_impl.c	12	312	159	-	-	
Linux	scrub.c	150	820	166	6	97	68
	slub.c	645	4,281	190	69	101	
	security.c	218	3,018	378	82	199	
CVS	apprentice.c	248	738	46	43	27	4
	compress.c	434	1,222	48	38	44	
	magic.c	244	526	8	28	2	
M4	freeze.c	16	84	26	5	5	4
	input.c	8	34	38	6	9	
	main.c	12	41	14	5	4	
OpenSSL	s_client.c	114	296	72	34	38	28
	ssl_ciph.c	18	258	104	36	55	
	t1_lib.c	683	106	569	104	330	
Gzip	deflate.c	2	2	1	1	1	2
	inflate.c	1	19	1	1	1	
	util.c	-	-	-	-	-	
Gnuplot	axis.c	108	190	38	22	34	8
	set.c	152	7,668	120	27	117	
	term.c	130	418	77	35	40	
		3,109	**22,874**	**2,878**	**756**	**1,699**	**155**

4.4. Discussion

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

Compilation errors:

Similar to our previous work [14], we found that most of the compilation errors that developers introduce are caused by the use of undeclared variables (41%), followed by no member in `struct` (22%) and incomplete definitions of types (12%). The second and third most introduced errors are related to structs. Fig. 2 presents the most common kinds of compilation errors found by CheckConfigMX in our study. Table 5 shows the kinds and total of compilation errors found by our approach. In this study, CheckConfigMX found 14 kinds of compilation errors which were not found in our previous study [14], such as type redefinition, incomplete result type, and argument not valid (see Ids 21–34 in Table 5.)

For example, a transformation in the `xmlparse.c` file from Expat (commits `3370a61` and `6ce9922`), introduced a compilation error related to undeclared variables. Listing 3a shows a code snippet of the original file. It declares a variable

Table 5
Kinds of compilation errors found in Study I.

id	Kinds of Compilation Errors	BusyBox	Apache HTTPD	Expat	Linux	Gzip	M4	OpenSSL	CVS	Gnuplot	Total
1	use of undeclared variable	171	123	2	92	1	6	205	64	38	**702**
2	no member in struct	24	165	-	4	-	-	179	3	-	**375**
3	member reference base type is not a structure or union	-	19	-	-	-	-	-	-	-	**19**
4	incomplete definition of type	16	2		114	-	-	28	1	49	**210**
5	invalid operands to binary expression	12	-	-	-	-	-	3	-	-	**15**
6	too many/few arguments to function call	-	12	-	2	-	-	1	-	-	**15**
7	unknown type name	-	11	-	42	-	-	-	-	-	**53**
8	use of undeclared label	9	-	-	-	-	-	-	-	-	**9**
9	expression is not assignable	6	-	-	-	-	-	-	-	2	**8**
10	conflicting types	-	4	-	75	1	-	-	7	24	**111**
11	subscripted value is not an array, pointer, or vector	4	-	-	-	-	-	-	-	-	**4**
12	there are not pointers to compatible types	2	-	1	-	-	-	-	-	-	**3**
13	reference to local variable declared in enclosing function	-	3	-	-	-	-	-	-	-	**3**
14	address of bit-field requested	-	2	-		-	-	-	-	-	**2**
15	called object type is not a function or function pointer	2	-	-	1	-	-	-	-	20	**23**
16	continue statement not in loop statement	-	1	-	-	-	-	-	-	-	**1**
17	function cannot return array type	-	1	-	-	-	-	-	-	-	**1**
18	non-void function should return a value	-	1	-	-	-	-	-	-	-	**1**
19	not a function or function pointer	-	1	-	-	-	-	-	-	-	**1**
20	type name requires a specifier or qualifier	-	1	-	-	-	-	2	-	-	**3**
21	type redefinition	-	-	-	41	-	6	2	-	49	**98**
22	indirection requires pointer operand	-	-	-	2	-	-	-	-	-	**2**
23	argument not valid	-	-	-	12	-	-	-	-	-	**12**
24	incomplete result type	-	-	-	2	-	-	-	-	-	**2**
25	a parameter list without types is only allowed in a function definition	-	-	-	7	-	-	-	-	-	**7**
26	field must have constant size	-	-	-	1	-	-	-	-	-	**1**
27	variable length array declaration not allowed at file scope	-	-	-	1	-	-	-	-	-	**1**
28	non-static declaration of follows static declaration	-	-	-	1	-	-	2	-	2	**5**
29	function definition not allowed	-	-	-	-	-	-	1	-	-	**1**
30	attribute not allowed	-	-	-	-	-	6	-	-	-	**6**
31	cannot combine with previous declaration specifier	-	-	-	-	-	-	-	-	1	**1**
32	ISO C requires a named parameter	-	-	-	-	-	-	-	-	1	**1**
33	array type has incomplete element type	-	-	-	-	-	-	-	-	1	**1**
34	array initializer must be an initializer list or string literal	-	-	-	-	-	-	-	-	4	**4**
	Total	**246**	**346**	**3**	**397**	**2**	**18**	**423**	**75**	**191**	**1,701**

`isParamEntity` under the `XML_DTD` macro. Listing 3b shows a code snippet of the modified file. In this version of the file, the `XML_StopParser` function uses the `isParamEntity` variable. The modified file with the `XML_DTD` macro disabled does not compile because the `isParamEntity` variable is not defined using this configuration. Developers fixed this error on commit 79aa349 (44 days later). They included the following comment in the code: *"Fixed compile errors when XML_DTD and XML_CONTEXT_BYTES were undefined."* The comment also indicates another compilation error in a file that CheckConfigMX did not analyze, which was caused by another disabled macro.

CheckConfigMX found that a transformation applied to `httpd.c` from BusyBox (commits `d2277e2` and `7291755`) introduced two compilation errors: *incomplete type definition*, and *undeclared variable*. Listings 1 and 2 show code snippets of the original and modified versions of this file. Medeiros et al. [6] also evaluated this code. They extended TypeChef to consider only one configuration of the header files and evaluated the modified file (commit `7291755`). They found only one of the errors (undeclared variable), 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 errors, such as the one in Listing 2. They only consider configurations in which one macro is enabled or disabled. In this example, we had to enable two macros to detect the error.

CheckConfigMX 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. Listing 4a shows a code snippet of the original file. It declares the `conn struct`, which contains the signed int `rvrse` (lines 1 to 3). Listing 4b shows a code snippet of

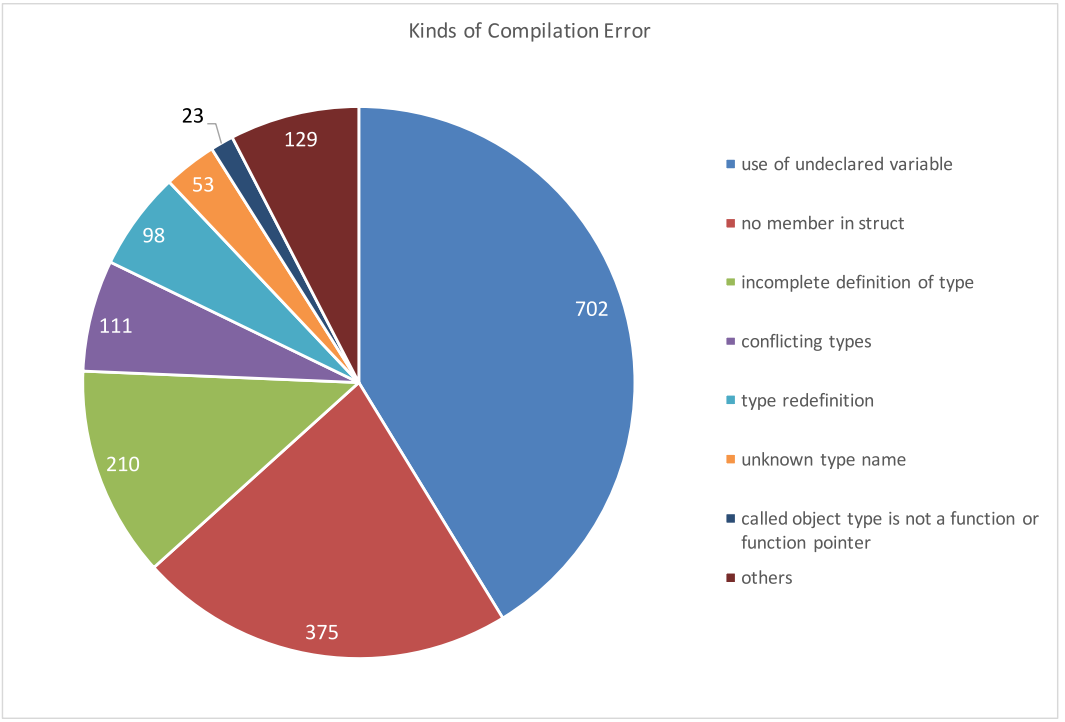

Fig. 2. Most common kinds of compilation errors found by CheckConfigMX in our study.

```
1 #ifdef XML_DTD
2 #define isParamEntity(...)
3 #endif
```

(a) Code snippet of the original `xmlparse.c` file (commit 3370a61).

```
1 #ifdef XML_DTD
2 #define isParamEntity(...)
3 #endif
4 int XML_StopParser(...){
5    if (isParamEntity){}
6 }
```

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

Listing 3. Code snippets of consecutive commits of `xmlparse.c` file of Expat, a transformation that introduced an undeclared variable compilation error.

the modified file. It tries to access the address of `rvrse`, which is a bit-field of size two (line 9). This attempt causes a compilation error because the memory can only be accessed byte by byte. 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: CheckConfigMX found 328 transformations (3.99%) that impacted more than four macros. Among them, 91 transformations have five impacted macros, 36 transformations have six impacted macros, and 30 transformations have seven impacted macros. CheckConfigMX found a transformation of `ash.c` file from BusyBox with 54 impacted macros (commits `cb81e64` and `c470f44`). CheckConfigMX found that 253 of the transformations that introduced compilation errors do not have impacted macros, and 286 of them have one impacted macro. Together, they represent 71.29% of the transformations that introduced at least one compilation error. We could detect them by combining the one-enabled, all-disabled, and all-enabled sampling algorithms. However, in our study, the one-enabled approach would require compiling an average of 12.04 configurations for each transformation with at most one impacted macro, while CheckConfigMX compiled an average of 3.58 configurations for each one.

The repositories of the analyzed systems have 8,219 transformations and CheckConfigMX analyzed 7,891 (96%) 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

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

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

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

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

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

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. CheckConfigMX compiles $O(2^{i+1})$ configurations, where i represents the number of impacted macros. For example, for the purpose of testing our approach in transformations that involve more than four impacted macros, CheckConfigMX 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 by the transformation. CheckConfigMX generated and compiled 256 configurations, and none of them introduced compilation errors. CheckConfigMX took 65.62 s to evaluate this transformation. Step 2 spent 58.63 s to compile the configurations, which is almost 30 times higher than the CheckConfigMX's average time of this step during our study. We can similarly evaluate the other transformations under this scenario. Developers should indicate the maximum number of macros that they would like to consider.

CheckConfigMX found 97 compilation errors introduced by transformations applied to the `scrub.c` file from Linux. This file has no macro in any version of the file in the repository. Therefore, CheckConfigMX also compiles files without `#ifdef` blocks and finds compilation errors on them. *Filter:* After compiling all configurations of each transformation, CheckConfigMX performed the filter step and found 22,874 compilation error messages in 2,834 configurations of 756 transformations. CheckConfigMX found 720 new messages in a transformation applied to `ash.c` from BusyBox (commits `59f351c` and `92e13c2`). This number is much higher than the average compilation errors messages (7.29) of the analyzed transformations with errors. CheckConfigMX categorized them into 10 compilation errors, of which 1 is false positive. We attempted to manually 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.

Categorization: CheckConfigMX groups messages based on patterns into distinct bugs by analyzing the type of the error and the elements that caused it. Our goal is to avoid reporting duplicated faults, since the same error may occur more than once in the same configuration or in different configurations of the same file. CheckConfigMX reduced the number of compilation errors to 2,878 (12.58%) distinct ones. For example, our filter found the following error messages for the transformation applied in the `mod_include.c` of Apache HTTPD (commits fa863b3 and 3565103):

1. `no member named 're_result' in 'struct ssi_internal_ctx'`
   `if (!ctx->intern->re_result || !ctx->intern->re_string)`
2. `no member named 're_result' in 'struct ssi_internal_ctx'`
   `apr_size_t len = (*ctx->intern->re_result)[idx]`
3. `no member named 're_result' in 'struct ssi_internal_ctx'`
   `(*ctx->intern->re_result)[idx]`

CheckConfigMX considers that these compilation errors are related to the same fault. CheckConfigMX categorized them into a compilation bug of the type *no member in struct* related to the `structssi_internal_ctx`.

False positives: We manually investigated all results of our automated categorization. We found that 817 (32.47%) of the compilation errors found by Step 3 were false positives. Some problems, such as missing libraries and compilers incom-

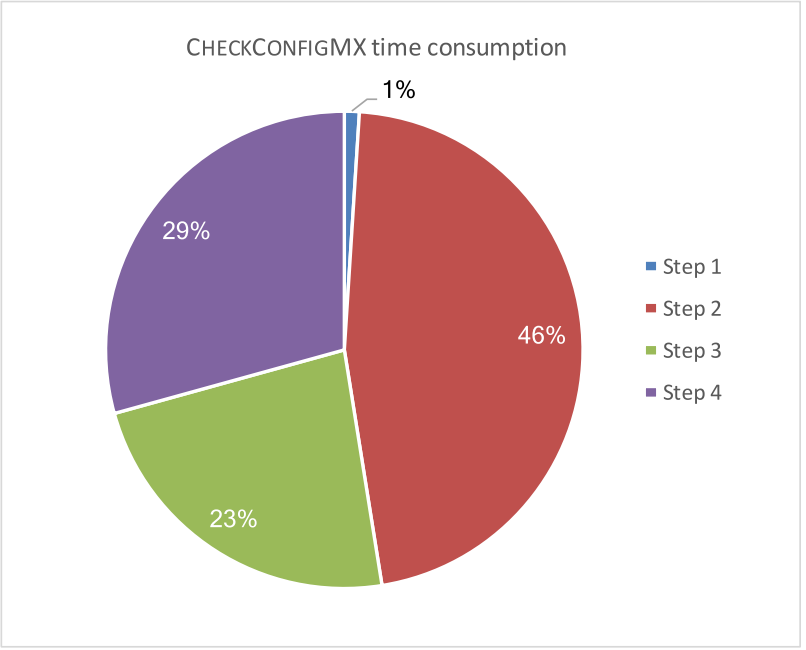

Fig. 3. CHECKCONFIGMX time consumption in our study. Step 1 = change impact analysis; Step 2 = compilation; Step 3 = filter; and, Step 4 = categorization.

patibility may cause these errors. The subjects that CHECKCONFIGMX 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 study and their dependencies, instead of the whole system. This may lead 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 appear in a global analysis since the compilation scope would include the file that defines the variables. We evaluate the false positive rate in more detail in Section 5.

CHECKCONFIGMX evaluated the repository history of all analyzed files of the configurable systems, which led us to analyze old commits. For example, CHECKCONFIGMX analyzed commits between August'99 and October'16. Therefore, as CHECKCONFIGMX is manipulating old files, some data may be unavailable nowadays. Also, CHECKCONFIGMX compiled the files on a modern compiler, while the developers may have compiled some of them on old compilers, that considered old versions of C. Therefore, this issue may lead to false positives.

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 file of a configurable system with `#ifdefs`. In a few seconds, CHECKCONFIGMX can report to them the set of possible compilation errors introduced by the transformation.

Some files take an above average amount of time to be evaluated. For example, the average time to evaluate the transformations applied to `modutils-24.c` from BusyBox and `mod_include.c` of Apache was 9 s. CHECKCONFIGMX found that `modutils-24.c` has four transformations with 16 or 32 impacted configurations. Moreover, CHECKCONFIGMX 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` from BusyBox (commits `68d5cb5` and `3eab24e`). Our approach took 20.28 s to evaluate it. This transformation impacted 32 configurations, in which our approach took 14 s 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). CHECKCONFIGMX uses GCC to compile them and it took on average 2.90 s 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 0.09 s to identify the impacted macros. Finally, it took on average 1.46 s and 1.80 s to perform the steps of filter and categorization, respectively. Fig. 3

Table 6
Averages of the runtime execution of CheckConfigMX; Time (s) = Average times for: Impact Analysis (Step 1); GCC (Step 2); Filter (Step 3); Categ. (Step 4); and, Total.

Configurable System	File	Time (s)				
		Impact Analysis	GCC	Filter	Categ.	Total
BusyBox	ash.c	0.02	2.02	0.69	2.49	5.23
	httpd.c	0.05	2.13	1.21	1.35	4.75
	hush.c	0.05	2.62	1.54	1.66	5.88
	modutils-24.c	0.05	3.72	2.03	3.29	9.10
	ntpd.c	0.04	1.91	1.27	2.51	5.74
	vi.c	0.05	1.77	1.13	2.68	5.63
Apache HTTPD	core.c	0.05	2.50	1.68	1.26	5.49
	event.c	0.11	2.81	2.13	1.63	6.69
	mod_include.c	0.30	4.30	2.44	1.77	9.82
	mod_rewrite.c	0.07	1.77	1.19	1.30	4.34
	proxy_util.c	0.06	3.13	2.10	2.53	7.84
Expat	xmlparse.c	0.06	1.56	1.03	1.30	3.96
	xmltok.c	0.04	1.76	1.16	1.74	4.71
	xmltok_impl.c	0.06	1.44	1.14	1.29	3.94
Linux	scrub.c	0.21	4.14	1.39	1.39	6.41
	slub.c	0.28	10.07	1.74	1.19	13.32
	security.c	0.57	9.77	1.87	1.40	13.61
CVS	apprentice.c	0.06	2.65	1.81	1.49	6.01
	compress.c	0.08	3.29	1.49	1.56	6.42
	magic.c	0.03	2.43	1.52	1.52	5.50
M4	freeze.c	0.04	1.72	1.34	1.35	4.46
	input.c	0.06	1.79	1.40	1.47	4.72
	main.c	0.03	1.62	0.91	1.16	3.72
OpenSSL	s_client.c	0.07	2.53	1.34	1.23	5.16
	ssl_ciph.c	0.11	3.00	1.62	1.60	6.33
	t1_lib.c	0.06	3.12	1.31	1.21	5.69
Gzip	deflate.c	0.03	2.06	1.53	1.38	5.00
	inflate.c	0.05	1.69	1.26	1.29	4.29
	util.c	0.05	1.85	1.45	1.40	4.75
Gnuplot	axis.c	0.05	2.09	1.29	3.03	6.49
	set.c	0.09	3.09	1.39	3.83	8.38
	term.c	0.04	2.42	1.28	3.31	7.06
	Average	**0.09**	**2.90**	**1.46**	**1.80**	**6.26**

presents the time consumption percentage of each step of CheckConfigMX in our study, and Table 6 shows the average time for analyzing all transformations.

4.5. Threats to validity

A feature model [15] has a significant role in the compilation of a Software Product Line [17,18], as each feature may impose specific constraints. 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 in these experiments. However, false

positives may not occur when using our approach in practice because developers may have a properly configured environment to compile the system. Moreover, we could minimize this threat by using our approach in a real environment, with all libraries and compilers available. The implementation of compilers may vary according to the operating system [19]. CheckConfigMX found different compilation errors by executing the same GCC version in Ubuntu 14.04 and Mac OS X Yosemite on `httpd.c` from BusyBox. Each compiler detected errors that the other did not detect. The internal platform dependencies of GCC may have bugs [19]. In addition, the different compilation errors may be caused by system-specific macros. CheckConfigMX 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 syntactic structure of the C programming language and code dependencies, which may cause false positives or negatives. For example, if a developer removes a variable that is used under a block `#ifdef M1 && !M2`, CheckConfigMX does not try to compile, and detect it if these macros are not impacted. In addition, we do not consider numbers in `#ifdefs` statements. For example, we handle an `#ifdef M1 < 1` by enabling or disabling the macro, but we do not assigning numerical values to it. Although our change impact analysis is simple, CheckConfigMX 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 1,699 errors introduced by the transformations. Table 5 presents all 34 kinds of compilation errors that CheckConfigMX found in 756 (9.19%) transformations applied to Apache, BusyBox, CVS, Expat, Linux, M4, Gzip, OpenSSL, and Gnuplot files. In our previous work [14], CheckConfigMX did not find 14 of the 34 kinds of compilation errors found in this study, such as type redefinition and incomplete result type. CheckConfigMX found 91 errors of these kinds in four configurable systems: M4, OpenSSL, CVS, and Gnuplot (see Ids 21–34 in Table 5).
- RQ_2: How much does CheckConfigMX reduce the effort needed 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 and without considering a feature model (see Table 3).
- RQ_3: What is the number of transformations that introduce at least one compilation error?
  CheckConfigMX found that 9.19% of the transformations of all analyzed files introduced at least one compilation error. The tool found 169 transformations that introduced one compilation error; 134 transformations that introduced two compilation errors; 34 transformations that introduced three compilation errors; 102 transformations that introduced four compilation errors; and, 401 transformations that introduced at least five compilation errors (see Table 4).
- RQ_4: What is the number of compilation errors that occur in transformations with no impacted macros?
  CheckConfigMX found 845 compilation errors in 253 transformations of all analyzed files with no impacted macros. This indicates that 49.73% of the compilation errors would be identified by the all-disabled-enabled approach. CheckConfigMX found the following number of transformations that introduced compilation errors: 286 with one impacted macro; 146 with two impacted macros; 50 with three impacted macros; and, 21 with four impacted macros.
- RQ_5: What is CheckConfigMX's frequency of false positives?
  CheckConfigMX found that 817 (32.47%) of the compilation errors that Step 4 found are false positives. Using our approach in a controlled environment would solve these issues (see Table 4).

5. Study II: mutation testing

Mutation testing has shown its value as a promising technique to assess the effectiveness of test cases and has been applied to many programming languages [20,21], such as C [22]. Many researchers apply mutation testing to evaluate configurable systems [20,23]. The analysis of Just et al. [24] indicates a statistically significant correlation between mutant detection and real fault detection. Daran and Thévenod-Fosse [25] perform a study that explored the relationship between mutants and real faults. They found that when the subject was exercised with generated test suites, the errors (incorrect internal state) and failures (incorrect output) produced by mutants were similar to those produced by real faults.

In this section, we describe our second evaluation using mutation testing to analyze the effectiveness of CheckConfigMX. We evaluate the tool with 11,229 mutants generated after we apply eight mutation operators in commit files of Apache, BusyBox, CVS, Expat, Linux, Gzip, M4, and OpenSSL. First, we present the definition (Section 5.1) and planning (Section 5.2). Next, Section 5.3 discusses the study results. Finally, Section 5.4 describes some threats to validity and Section 5.5 summarizes the main findings.

5.1. Definition

We address the following research question:

```
1 #ifdef M1
2 int x = 0;
3 int y = x + 1;
4 #endif M1
```

(a) MO1 - Code snippet of the original configurable system.

```
1 #ifdef M1
2 int z = 0;
3 int y = x + 1;
4 #endif M1
```

(b) MO1 - Code snippet of the mutant configurable system.

Listing 5. Configurable system with `#ifdefs` before and after the application of the mutation operator MO1.

```
1 void function1(){}
2 void function2(){
3 #ifdef M2
4   function1();
5 #endif
6 }
```

(a) MO2 - Code snippet of the original configurable system.

```
1 void function2(){
2 #ifdef M2
3   function1();
4 #endif
5 }
```

(b) MO2 - Code snippet of the mutant configurable system.

Listing 6. Configurable system with `#ifdefs` before and after the application of the mutation operator MO2.

- RQ_6: What is CheckConfigMX's mutation score?
  We execute CheckConfigMX in the transformations (original and mutant codes) to measure the amount of killed mutants. In addition, we identify the mutation score of CheckConfigMX by the ratio between the total of killed mutants by the total of mutants.

5.2. Planning

We apply mutation operators to at most 50 versions of the files presented in Section 4.2.1, according to the amount of versions in their repositories. We use the same setup as described in Section 4.2.2. Mutation testing results can be used as a reference for new test cases or to measure confidence on existing tests [24]. Therefore, we use the mutation score to measure the effectiveness of CheckConfigMX. We manually apply the eight kinds of mutation operators [26], such as the use of undeclared variables and the division of expressions by zero [27], and generate different mutants for each application. Previous approaches found similar bugs in real configurable systems [20,23]. We consider that CheckConfigMX kills a mutant when it detects all compilation errors introduced by the mutant operators. Next, we describe the mutation operators used in the study.

5.2.1. MO1 - rename variable

The *rename variable* operator replaces the name of any variable for a new one, it may cause a compilation error of duplication or the use of an undeclared variable or function. For example, Listing 5a declares a variable `x`. The mutation operator renames it, generating the mutant presented in Listing 5b. The mutant has an undeclared variable at Line 3 when `M1` is enabled.

5.2.2. MO2 - remove declaration of function or variable

The *remove declaration of function or variable* operator is used to check whether removing the declaration of a variable or a function yields a compilation error of undeclared variable or function. For example, Listing 6a declares two functions, where the second calls the first one. The mutation operator removes the declaration of the first function, generating the mutant in Listing 6b. The mutant presents the compilation error when `M2` is enabled.

5.2.3. MO3 - change pointer operator

The *change pointer operator* swaps * and & operators in both directions, which may cause a compilation error. For example, Listing 7a uses pointer operators. The mutation operator changes the pointer operator at Line 4, generating the mutant in Listing 7b. The mutant presents the compilation error ‘‘`indirection requires pointer operand`’’ when `M3` is enabled, since `x` is not a pointer in this scope.

```
1 #ifdef M3
2 void function1(){
3   int  x;
4   int *p = &x;
5 }
6 #endif M3
```

(a) MO3 - Code snippet of the original configurable system.

```
1 #ifdef M3
2 void function1(){
3   int  x;
4   int *p = *x;
5 }
6 #endif M3
```

(b) MO3 - Code snippet of the mutant configurable system.

Listing 7. Configurable system with `#ifdefs` before and after the application of the mutation operator MO3.

```
1 int main(){
2 #ifdef M4
3   int x;
4 #endif M4
5 }
```

(a) MO4 - Code snippet of the original configurable system.

```
1 int main(){
2 #ifdef M4
3   int x;
4   int x;
5 #endif M4
6 }
```

(b) MO4 - Code snippet of the mutant configurable system.

Listing 8. Configurable system with `#ifdefs` before and after the application of the mutation operator MO4.

```
1 #ifdef M5
2 int x;
3 int y = x;
4 #endif M5
```

(a) MO5 - Code snippet of the original configurable system.

```
1 #ifdef M5
2 int x;
3 int y [1] = x;
4 #endif M5
```

(b) MO5 - Code snippet of the mutant configurable system.

Listing 9. Configurable system with `#ifdefs` before and after the application of the mutation operator MO5.

5.2.4. MO4 - duplicate declaration of variable or

Function This mutation operator duplicates the declaration of a variable or a function and yields a compilation error indicating the duplication. For example, Listing 8a has a variable declaration at Line 3. The mutation operator duplicates the declaration of this variable, generating the mutant in Listing 8b. The mutant does not compile when `M4` is enabled due to a duplicated declaration compilation error at Line 6.

5.2.5. MO5 - change type of variable declaration

This mutation operator changes the type of an `int` variable to `array` and yields a compilation error when the variable is used as if it still has its old type. Listing 9a has a variable `y` of the type `int` declared at Line 3. Variable `x` of type `int` is assigned to `y` (Line 4). After changing the type of `y` from `int` to `array`, the mutant code (Listing 9b) does not compile when `M5` is enabled. The mutant presents the compilation error ‘‘`array initializer must be an initializer list or wide string literal.`’’

5.2.6. MO6 - remove variable of statement

The *remove variable of statement* operator removes one variable from a statement, instead of removing it completely as MO3 does. For example, consider the statement at Line 2 of Listing 10a. The mutation operator removes part of this statement, generating the mutant in Listing 10b. The mutant presents a compilation error when `M6` is enabled, since the compiler expects an expression instead of the semicolon after the operator `+`.

```
1 #ifdef M6
2 int x = 1 + 1;
3 #endif M6
```

(a) MO6 - Code snippet of the original configurable system.

```
1 #ifdef M6
2 int x = 1 + ;
3 #endif M6
```

(b) MO6 - Code snippet of the mutant configurable system.

Listing 10. Configurable system with `#ifdefs` before and after the application of the mutation operator MO6.

```
1 #ifdef M7
2 int function1(){
3   return 1;
4 }
5 void function2(){
6   int x = function1();
7 }
8 #endif M7
```

(a) MO7 - Code snippet of the original configurable system.

```
1 #ifdef M7
2 void function1(){
3   return 1;
4 }
5 void function2(){
6   int x = function1();
7 }
8 #endif M7
```

(b) MO7 - Code snippet of the mutant configurable system.

Listing 11. Configurable system with `#ifdefs` before and after the application of the mutation operator MO7.

```
1 int main(){
2 #ifdef M8
3   int x;
4 #endif M8
5 }
```

(a) MO8 - Code snippet of the original configurable system.

```
1 int main(){
2 #ifdef M8
3   int x
4 #endif M8
5 }
```

(b) MO8 - Code snippet of the mutant configurable system.

Listing 12. Configurable system with `#ifdefs` before and after the application of the mutation operator MO8.

5.2.7. MO7 - change type of return

This mutation operator swaps the return type of a `void` function to `int` and vice versa. Consider Listing 11a, the return type of the `function1` is `int` at Line 2. The mutation operator changes the return type of `function1` to `void`, generating the mutant in Listing 11b. The mutant does not compile when `M7` is enabled. It presents two compilation errors: ‘‘`void function 'function1' should not return a value`’’; and, ‘‘`initializing 'int' with an expression of incompatible type 'void'.`’’

5.2.8. MO8 - remove semicolon

The *remove semicolon* operator yields a compilation error indicating that the punctuation is expected at the end of the statement. Listing 12a declares a variable `x` at Line 3. The mutation operator removes the semicolon at the end of this line, generating the mutant in Listing 12b. The mutant presents the compilation error when `M8` is enabled.

5.3. Results and discussion

CheckConfigMX evaluated 11,229 mutants. It identified an impacted macro in each one. We identified 27 equivalent mutants generated by MO1 and do not consider them in our analysis. CheckConfigMX reduced the analysis effort average from $3.17x10^{12}$ possible configurations to 4 analyzed configurations per mutant (99% of reduction). Table 7 summarizes our findings.

We investigated whether our approach may have false negatives by using mutation testing. We manually applied the eight kinds of the mutation operators in at most 50 versions of all files considered in Study I (Section 4.2.1). In some files,

Table 7
Results of the CheckConfigMX's analysis of configurable systems during the mutation testing study; Version = Total of analyzed versions per file; MO*i* = Number of mutants generated by mutant operator *i*; Score = Mutation Score.

Configurable System	File	Versions	MO1	MO2	MO3	MO4	MO5	MO6	MO7	MO8
BusyBox	ash.c	50	50	50	50	50	50	50	50	50
	httpd.c	50	46	50	50	50	50	50	50	50
	hush.c	50	50	50	50	50	50	50	50	50
	modutils-24.c	24	24	24	24	24	24	24	24	24
	ntpd.c	50	50	50	50	50	50	50	50	50
	vi.c	50	50	50	50	50	50	50	50	50
Apache HTTPD	core.c	50	50	50	50	50	50	50	50	50
	event.c	50	50	50	50	50	50	50	50	50
	mod_include.c	50	41	50	50	50	50	50	50	50
	mod_rewrite.c	50	44	50	50	50	50	50	50	50
	proxy_util.c	50	46	50	50	50	50	50	50	50
Expat	xmlparse.c	50	50	50	50	50	50	50	50	50
	xmltok.c	36	34	36	36	36	36	36	36	36
	xmltok_impl.c	14	14	14	14	14	14	14	14	14
Linux	scrub.c	50	50	50	50	50	50	50	50	50
	slub.c	50	50	50	50	50	50	50	50	50
	security.c	50	48	50	50	50	50	50	50	50
CVS	apprentice.c	50	50	50	50	50	50	50	50	50
	compress.c	50	50	50	50	50	50	50	50	50
	magic.c	50	50	50	50	50	50	50	50	50
M4	freeze.c	50	50	50	50	50	50	50	50	50
	input.c	50	50	50	50	50	50	50	50	50
	main.c	50	50	50	50	50	50	50	50	50
OpenSSL	s_client.c	50	50	50	50	50	50	50	50	50
	ssl_ciph.c	50	50	50	50	50	50	50	50	50
	t1_lib.c	50	50	50	50	50	50	50	50	50
Gzip	deflate.c	26	26	26	26	26	26	26	26	26
	inflate.c	28	28	28	28	28	28	28	28	28
	util.c	29	29	29	29	29	29	29	29	29
Gnuplot	axis.c	50	50	50	50	50	50	50	50	50
	set.c	50	50	50	50	50	50	50	50	50
	term.c	50	50	50	50	50	50	50	50	50
	Total	**1,407**	**1,380**	**1,407**	**1,407**	**1,407**	**1,407**	**1,407**	**1,407**	**1,407**
	Score	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

we found more than one place to apply the mutant operator. We randomly chose one of them. The mutation operators MO3 and MO7 can be applied in both directions. We generated almost the same number of mutants for each direction. 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 [24]. Previous approaches [6,11,16] found bugs in real configurable systems caused by changes similar to the mutation operators MO1–MO8. However, we did not find compilation errors related to MO3, MO6–MO8 in Study I.

CheckConfigMX killed all 11,229 mutants generated using MO1–MO8. As input, the tool received the original file version from the configurable system and the mutant file, and evaluated whether the transformation introduced compilation errors. We manually identified the 27 equivalent mutants. The mutation operator *rename variable* yields equivalent mutants when the new name is not duplicated or if the variable or function is never used. For example, we applied this operator to a commit of the `security.c` (commit `7089db`) file from Linux, Listing 13a presents the original code. Notice that there is the definition of the `security_init` function. We renamed the function to `renamed_security_init` impacting the

```
1 #ifdef CONFIG_AUDIT
2 int security(...){
3   return call_int_hook(...);
4 }
5 #endif
```

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

```
1 #ifdef CONFIG_AUDIT
2 int renamed_security(...){
3   return call_int_hook(...);
4 }
5 #endif
```

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

Listing 13. Code snippets of a commit of the `security.c` file and its mutant code after we applied a rename.

```
1 #ifdef !M1
2 int x = 0;
3 #endif
4 #ifdef !M2 && M3
5 int y = x + 1;
6 #endif
```

(a) Code snippet of the original file.

```
1 #ifdef M1
2
3 #endif
4 #ifdef !M2 && M3
5 int y = x + 1;
6 #endif
```

(b) Code snippet of the modified file.

Listing 14. Code snippets of a transformation that introduces an undeclared variable compilation error.

`CONFIG_AUDIT` macro. However, as this function is never called in this file, the mutation does not cause a compilation error. We do not consider equivalent mutants in our analysis.

The average of all possible configurations per transformation, without considering a feature model, is $3.17x10^{12}$. Our approach reduced the effort to evaluate the transformations by at least 50% (an average of 99%) when compared to the exhaustive approach. Moreover, our experiment results show evidence that CheckConfigMX may be useful to reduce the number of compiled configurations (effort) in a configurable system by comparing with sampling approaches.

5.4. Threats to validity

The mutation operators applied during the study were selected based on Agrawal et al. [22]. This selection may have resulted in a non-representative set of mutation operators that may cause only compilation errors that the tool is able to detect. Furthermore, CheckConfigMX has some limitations. There are some compilation errors that the tool does not detect (false negatives). For example, it does not detect the compilation error introduced by the transformation in Listing 14. CheckConfigMX does not compile a configuration with `M2` disabled and `M3` enabled due to its simple change impact analysis. However, we did not find any similar scenarios in our evaluation using mutation operators. Moreover, applying other kinds of mutation operators may lead to different results. In Study I, we did not detect compilation errors related to the mutation operators MO3, MO6-MO8. Study II indicates that CheckConfigMX may detect them if they happen in a configurable system.

5.5. Research question answer

From the evaluation results, we made the following observations:

- RQ_6: What is CheckConfigMX's mutation score?
  CheckConfigMX had a mutation score of 1 indicating that the tool killed all 11,229 mutants generated by eight kinds of mutation operators applied to at most 50 versions of each 29 files of configurable systems (see Table 7). CheckConfigMX detect compilation errors related to the mutation operators MO3, MO6-MO8 that were not detect in Study I.

6. Related work

Researchers have analyzed configuration-related errors in configurable systems [6,16] through the analysis of software repositories [5,11]. They found that most of the configuration-related bugs involve only a few macros. Abal et al. [11] manually analyzed commits of the Linux Kernel software repository to study configuration-related bugs. They suggested the *one-disabled* sampling algorithm, which deactivates one preprocessor directive at a time; it requires *n* configurations per

file, where n is the number of preprocessor macros in each source file. However, such approach may miss some bugs. Moreover, they stated that a family-based automated analysis tool that scales for the Linux kernel did not exist. Later, Tartler et al. [28] proposed a technique to identify a set of configurations that maximizes the Configuration Coverage (CC) of a configurable system with respect to statement coverage. They found a number of configuration-related bugs in the Linux Kernel project by using the Undertaker tool that implements the approach. However, different from our approach, none of the previous studies analyze the impacted configurations. Moreover, we use CheckConfigMX in fine-grained transformations applied to Linux Kernel files.

Kästner et al. [12] proposed TypeChef, a variability-aware parser, which analyzes all possible configurations and performs type checking. alt 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. Gazzillo and Grimm [13] presented SuperC, another variability-aware parser. It is faster than TypeChef, but it does not perform type checking. Our approach differs from those approaches in the sense that we focus only on compiling the configurations impacted by the change. In addition, CheckConfigMX requires two versions of a configurable system file, while TypeChef and SuperC require only one version.

The time-consuming setup of variability-aware tools hinders the analysis of several projects. An average file in the Linux Kernel, for example, includes over 300 header files [12]. Furthermore, incorporating header files increases the number of preprocessor macros per file significantly. Medeiros et al. [5] proposed an approach to improve the TypeChef's scalability problem. They generated stubs to replace the original types and macros from header files, which became undefined after replacing the `#include` directives. They found 24 configuration-related syntax errors in 40 configurable systems. However, their approach also analyzed all possible configurations. Although we do not identify syntax errors in Study I, we can detect them, as we show in Study II by using the mutation operator *remove semicolon* (MO8). Moreover, our approach compiles only the impacted configurations.

Later, Medeiros et al. [6] conducted a study to better understand undeclared/unused variables and functions related to configurability. They proposed a strategy that considers only one configuration of the header files to scale their study and minimize possible setup problems. They implemented 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. CheckConfigMX 34 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 also evaluated the file (commit `79aa349`), they do not detected this compilation error as we do, since manually analyzing a number of configurations and error messages may be error prone. Our approach compiles only the impacted configurations. Moreover, we can adapt Steps 2 and 3 of our approach to evaluate the warning messages that they consider.

Previous studies proposed various strategies for dealing with configuration spaces [11,28,29]. Medeiros et al. [16] compared 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 find compilation errors that the most-enabled-disabled cannot detect (Section 3).

Qu et al. [30] presented an approach that selects configurations for regression testing using slicing-based code change impact analysis [31]. 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 simple, our study results show evidence that CheckConfigMX can be useful to reduce the effort to detect compilation errors in configurable systems with `#ifdef` directives.

In the context of the C language, some studies proposed tools that perform static and dynamic analysis to find bugs, such as memory leaks. Evans [32] presented Splint to statically check C programs for security vulnerabilities and coding mistakes. Bond and McKinley [33] proposed Plug to detect memory leaks in C and C++ programs. Nethercote and Seward [34] presented 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) with some of those tools to detect other kinds of bugs.

Feature models are often used to express the intended variability of a configurable system [15,35]. Previous research proposed tools that identify anomalies in these models, such as FeatureIDE [36], TVL [37], FAMILIAR [38], and pure::variants [39]. Kowal et al. [40] proposed an algorithm, implemented in FeatureIDE, for explaining different anomalies in feature models. They evaluate several large-scale feature models including an industrial one. In all cases, the explanation length stays acceptable at the cost of a doubled computation time for the complete process. These approaches analyze the complete configuration space. CheckConfigMX, on the other hand, compiles only the impacted configurations. In addition, these approaches investigate anomalies in feature models of configurable systems, while our approach detects compilation errors in the source code of such configurable systems. We can adapt CheckConfigMX to consider the feature models of the configurable systems in Step 2, when generating the configurations to be compiled.

Law and Rothermel [41] presented an approach based on static and dynamic partitioning and recursive algorithms of calls graphs to identify methods impacted by a change. Different from our approach, their analysis estimates the impact before applying the transformations. Ren et al. [42] proposed Chianti that decomposes Java changes into atomic ones and generates a dependency graph. The tool indicates the test cases that are impacted by the change. Zhang et al. [43] proposed FaultTracer, a change impact analysis tool that improves Chianti to refine the dependencies between atomic changes, and adds more rules to calculate the impact of a change. Mongiovi et al. [44] presented Safira that analyzes two versions of a Java/AspectJ program and identify the methods impacted by the change. It decomposes the transformation into a set of small-grained transformations and formalizes a set of laws to calculate the impact of each small-grained transformations separately. Different from the previous approaches, our change impact analysis does not consider the structures of the language. We perform textual comparison to identify the macros impacted by the change. Moreover, we evaluate whether a transformation applied to a configurable system with `#ifdef` introduces compilation errors.

Thum et al. [20] compared model checking and theorem proving for product-line verification by using mutation testing to measure their effectiveness, performance, and efficiency. They applied nine types of mutation operators, such as swap the logical values `true` and `false`, to 318 feature modules and feature-oriented contracts (Java, JML, and Sum). However, they considered the whole configurable system to perform model checking and theorem proving, while CheckConfigMX considers only the possible impacted configurations to detect introduced compilation errors.

Later, Al-Hajjaji et al. [45] proposed a set of mutation operators for software with preprocessor-based variability. The operators are divided into six main fault types: feature dependencies; feature-definition; insufficient-mapping; unnecessary mapping; feature interaction; and, single feature. Their operators focused on pre-processor operators, such as completely removing a `#ifdef` block and replacing a `#ifdef` directive with `#ifndef` directive. Although we only consider mutation operators to source code, we can use their operators to further evaluate our approach and check whether CheckConfigMX detects compilation errors caused by pre-processor mutant operators.

7. Conclusions

We propose a change-aware per-file analysis to compile configurable systems with `#ifdefs` [14]. 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. In this article, we extend our previous work [14] by including 21 new files of seven different configurable systems in our evaluation.

We used CheckConfigMX to compile 7,891 transformations applied to 32 files of real configurable systems with `#ifdefs`, such as Linux. Besides effort reduction, our results show that CheckConfigMX is useful to detect compilation errors introduced by fine-grained transformations applied to configurable systems. In total, the tool found 1,699 compilation errors of 34 kinds; 14 of these kinds were not detected in our previous work [14]. Most of the compilation errors are related to *undeclared variables* (41%) and *no member in struct* (22%).

Configurable systems potentially have an exponentially large set of configurations. Analyzing all of these configurations might be infeasible. The studies presented here demonstrate that the approach of focusing analysis on configurations that have been impacted as the result of a source code change dramatically reduces the computational effort (over 99% on average) and still finds many compilation errors. CheckConfigMX took only few seconds to analyze the impacted configurations of each transformation applied to highly configurable systems. An additional advantage of this approach is that it can employ the developer's preferred compiler, thereby ensuring that the errors reported by the approach are ones that the developer would encounter in practice.

Tools and techniques to help developers manage the complexity of configurable systems and reduce the errors in them are becoming increasingly important as time moves forward. Focusing analysis on just configurations that have been impacted by a source code change is an efficient and effective technique that can make a wide variety of existing program analysis techniques more practical in the context of configurable systems, including compilation (as demonstrated in this study). This work gives more evidence of the benefits of performing change-centric analysis [44,46]. We suggest that developers can use the CheckConfigMX after performing fine-grained transformations on code to quickly detect whether they whintroduced new compilation errors. For coarse-grained transformations impacting a number of macros, CheckConfigMX may take longer and in some cases may not be capable of processing it. CheckConfigMX may yield false positives, since it performs a per-file analysis. A local change may have global impact [44]. Developers may manually analyze and remove them. In general, the tool yields few messages for each fine-grained transformation analyzed.

As future work, we plan to adjust the tool to evaluate entire configurable systems by analyzing and compiling all their files. We also aim at analyzing more configurable systems. Moreover, CheckConfigMX may detect false positives as it does not consider feature models when identifying impacted configurations. We aim at considering them to compile only valid configurations. We also intend to compare CheckConfigMX with other tools, such as TypeChef [12] and SuperC [13]. Furthermore, we intend to implement and propose more mutation operators to automatically apply them on the source code, and evaluate whether CheckConfigMX can detect them.

Acknowledgments

We would like to thank Christian Kästner and Sven Apel. This work was partially funded by CNPq grants 308380/2016-9, 477943/2013-6, 460883/2014-3, 465614/2014-0, 306610/2013-2, 307190/2015-3, and 409335/2016-9, FAPEAL PPGs 14/2016, CAPES grants 175956 and 117875, FACEPE grant APQ-0570-1.03/14.

References

[1] Siegmund N, Grebhahn A, Apel S, Kästner C. Performance-influence models for highly configurable systems. In: Proceedings of the tenth joint meeting on foundations of software engineering; 2015. p. 284–94.
[2] Donohoe P. Introduction to software product lines. In: Proceedings of the thirteenth international software product line conference; 2009. p. 305.
[3] Berger T, Rublack R, Nair D, Atlee J, Becker M, Czarnecki K, Wasowski A. A survey of variability modeling in industrial practice. In: Proceedings of the seventh international workshop on variability modelling of software-intensive systems; 2013. p. 1–7.
[4] Berger T, She S, Lotufo R, Wasowski A, Czarnecki K. A study of variability models and languages in the systems software domain. IEEE Trans. Softw. Eng. 2013;39(12):1611–40.
[5] Medeiros F, Ribeiro M, Gheyi R. Investigating preprocessor-based syntax errors. In: Proceedings of the twelfth international conference on generative programming: concepts & experiences; 2013. p. 75–84.
[6] Medeiros F, Rodrigues I, Ribeiro M, Teixeira L, Gheyi R. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In: Proceedings of the fifteenth international conference on generative programming: concepts and experiences; 2015. p. 35–44.
[7] Ernst M, Badros G, Notkin D. An empirical analysis of C preprocessor use. IEEE Trans. Softw. Eng. 2002;28(12):1146–70.
[8] Medeiros F, Kästner C, Ribeiro M, Nadi S, Gheyi R. The love/hate relationship with the C preprocessor: an interview study. In: Proceedings of the twenty-ninth European conference on object-oriented programming; 2015. p. 495–518.
[9] Malaquias R, Ribeiro M, Bonifácio R, Monteiro E, Medeiros F, Garcia A, Gheyi R. The discipline of preprocessor-based annotations does # ifdef tag n't # endif matter. In: Proceedings of the twenty-fifth international conference on program comprehension; 2017 pp. xx–yy.
[10] Kästner C, Apel S, Thüm T, Saake G. Type checking annotation-based product lines. Trans. Softw. Eng. Methodol. 2012;21(3):1–39.
[11] Abal I, Brabrand C, Wasowski A. 42 variability bugs in the Linux kernel: a qualitative analysis. In: Proceedings of the twenty-ninth international conference on automated software engineering; 2014. p. 421–32.
[12] Kästner C, Giarrusso P, Rendel T, Erdweg S, Ostermann K, Berger T. Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of the seventeenth international conference on object oriented programming systems languages and applications; 2011. p. 805–24.
[13] Gazzillo P, Grimm R. SuperC: parsing all of C by taming the preprocessor. In: Proceedings of the thirty-third conference on programming language design and implementation; 2012. p. 323–34.
[14] Braz L, Gheyi R, Mongiovi M, Ribeiro M, Medeiros F, Teixeira L. A change-centric approach to compile configurable systems with #ifdefs. In: Proceedings of the fifteenth international conference on generative programming: concepts & experiences; 2016. p. 109–19.
[15] Kang K, Cohen S, Hess J, Novak W, Peterson A. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. Carnegie-Mellon University Software Engineering Institute; 1990.
[16] Medeiros F, Kästner C, Ribeiro M, Gheyi R, Apel S. A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the thirty-eighth international conference on software engineering; 2016. p. 643–54.
[17] Clements P, Northrop L. Software product lines: practices and patterns. Addison-Wesley; 2009.
[18] Pohl K, Bockle G, Linden F. Software product line engineering: foundations, principles and techniques. Springer-Verlag; 2005.
[19] Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proceedings of the thirty-second conference on programming language design and implementation; 2011. p. 283–94.
[20] Thüm T, Meinicke J, Benduhn F, Hentschel M, von Rhein A, Saake G. Potential synergies of theorem proving and model checking for software product lines. In: Proceedings of the eighteenth international software product line conference; 2014. p. 177–86.
[21] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 2011;37(5):649–78.
[22] Agrawal H, DeMillo R, Hathaway R, Hsu W, Hsu W. Design of mutant operators for the C programming language. Tech. Rep. Purdue University; 1989.
[23] Reuling D, Bürdek J, Rotärmel S, Lochau M, Kelter U. Fault-based product-line testing: effective sample generation based on feature-diagram mutation. In: Proceedings of the nineteenth international software product line conference; 2015. p. 131–40.
[24] Just R, Jalali D, Inozemtseva L, Ernst M, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing?. In: Proceedings of the twenty-second international symposium on foundations of software engineering; 2014. p. 654–65.
[25] Daran M, Thévenod-Fosse P. Software error analysis: a real case study involving real faults and mutations. In: Proceedings of the fifth international symposium on software testing and analysis; 1996. p. 158–71.
[26] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. IEEE Comput. 1978;11(4):34–41.
[27] Frankl P, Weiss S, Hu C. All-uses vs mutation testing: an experimental comparison of effectiveness. J. Syst. Softw. 1997;38(3):235–53.
[28] Tartler R, Lohmann D, Dietrich C, Egger C, Sincero J. Configuration coverage in the analysis of large-scale system software. In: Proceedings of the sixth workshop on programming languages and operating systems; 2011. p. 1–5.
[29] Kuhn D, Wallace D, Gallo A. Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. 2004;30:418–21.
[30] Qu X, Acharya M, Robinson B. Configuration selection using code change impact analysis for regression testing. In: Proceedings of the twenty-eighth IEEE international conference on software maintenance. IEEE Computer Society; 2012. p. 129–38.
[31] Bohner S, Robert A. Software change impact analysis. Wiley-IEEE Computer Society Press; 1996.
[32] Evans D. Static detection of dynamic memory errors. In: Proceedings of the conference on programming language design and implementation; 1996. p. 44–53.
[33] Bond M, McKinley K. Tolerating memory leaks. In: Harris GE, editor. Proceedings of the fourteenth international conference on object oriented programming systems languages and applications; 2008. p. 109–26.
[34] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the twenty-eighth programming language design and implementation; 2007. p. 89–100.
[35] Benavides D, Segura S, Ruiz-Cortés A. Automated analysis of feature models 20 years later: a literature review. Inf. Syst. 2010;35(6):615–36.
[36] Meinicke J, Thüm T, Schröter R, Krieter S, Benduhn F, Saake G, et al. FeatureIDE: taming the preprocessor wilderness. In: Proceedings of the thirty-eight international conference on software engineering companion; 2016. p. 629–32.
[37] Classen A, Boucher Q, Heymans P. A text-based approach to feature modelling: syntax and semantics of TVL. Sci. Comput. Program. 2011;76(12):1130–43.
[38] Acher M, Collet P, Lahire P, France R. FAMILIAR: a domain-specific language for large scale management of feature models. Sci. Comput. Program. 2013;78(6):657–81.
[39] Beuche D. Modeling and building software product lines with pure: variants. In: Proceedings of the twelfth international software product line conference; 2008. p. 358.
[40] Kowal M, Ananieva S, Thüm T. Explaining anomalies in feature models. In: Proceedings of the fifteenth ACM SIGPLAN international conference on generative programming: concepts and experiences; 2016. p. 132–43.

[41] Law J, Rothermel G. Whole program path-based dynamic impact analysis. In: Proceedings of the twenty-fifth international conference on software engineering. IEEE Computer Society; 2003. p. 308–18.
[42] Ren X, Ryder B, Stoerzer M, Tip F. Chianti: a change impact analysis tool for Java programs. In: Proceedings of the twenty-seventh international conference on software engineering; 2005. p. 664–5.
[43] Zhang L, Kim M, Khurshid S. FaultTracer: a change impact and regression fault analysis tool for evolving Java programs. In: Proceedings of the twentieth international symposium on the foundations of software engineering; 2012. p. 40.
[44] Mongiovi M, Gheyi R, Soares G, Teixeira L, Borba P. Making refactoring safer through impact analysis. Sci. Comput. Program. 2014;93:39–64.
[45] Al-Hajjaji M, Benduhn F, Thüm T, Leich T, Saake G. Mutation operators for preprocessor-based variability. In: Proceedings of the tenth international workshop on variability modelling of software-intensive systems; 2016. p. 81–8.
[46] Wloka J, Hoest E, Ryder BG. Tool support for change-centric test development. IEEE Softw. 2010;27(3):66–71.