An Empirical Study on Configuration-Related Code Weaknesses

Flávio Medeiros
IFAL, Alagoas, Brazil
flavio.medeiros@ifal.edu.br

Márcio Ribeiro
UFAL, Alagoas, Brazil
marcio@ic.ufal.br

Rohit Gheyi
UFCG, Paraíba, Brazil
rohit@dsc.ufcg.edu.br

Larissa Braz
UFCG, Paraíba, Brazil
larissanadja@copin.ufcg.edu.br

Christian Kästner
CMU, Pennsylvania, USA
kaestner@cs.cmu.edu

Sven Apel
Saarland University, Saarland, Germany
apel@cs.uni-saarland.de

Kleber Santos
UFCG, Paraíba, Brazil
kleber@copin.ufcg.edu.br

Abstract

Developers often use the C preprocessor to handle variability and portability. However, many researchers and practitioners criticize the use of preprocessor directives because of their negative effect on code understanding, maintainability, and error proneness. This negative effect may lead to configuration-related code weaknesses, which appear only when we enable or disable certain configuration options. A weakness is a type of mistake in software that, in proper conditions, could contribute to the introduction of vulnerabilities within that software. Configuration-related code weaknesses may be harder to detect and fix than weaknesses that appear in all configurations, because variability increases complexity. To address this problem, we propose a sampling-based white-box technique to detect configuration-related weaknesses in configurable systems. To evaluate our technique, we performed an empirical study with 24 popular highly configurable systems that make heavy use of the C preprocessor, such as Apache Httpd and Libssh. Using our technique, we detected 57 configuration-related weaknesses in 16 systems. In total, we found occurrences of the following five kinds of weaknesses: 30 memory leaks, 10 uninitialized variables, 9 null pointer dereferences, 6 resource leaks, and 2 buffer overflows. The corpus of these weaknesses is a valuable source to better support further research on configuration-related code weaknesses.

Keywords

Configurable Systems, Preprocessors, Code Weaknesses

ACM Reference Format:
Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Larissa Braz, Christian Kästner, Sven Apel, and Kleber Santos. 2020. An Empirical Study on Configuration-Related Code Weaknesses. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422409

SBES '20, October 21–23, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422409

1 Introduction

Almost any substantial software system can be configured to adapt it to the requirements of the user, the target platform and the application scenario [12, 30, 49]. The individual configurations derived this way differ in terms of the features they offer [23], which encapsulate platform-specific code and optional functionalities. Developers often implement configurable systems using preprocessor directives, such as the *#ifdef* macro, declaring parts of the source code as optional. However, despite the widespread use of the C preprocessor, many studies and practitioners[1] criticize the use of preprocessor directives because of their negative effect on code understanding and maintainability [5, 12, 18, 24, 30]. This negative effect may lead to configuration-related weaknesses, which typically occur only in specific configurations. Weakness is a type of mistake in software that, in proper conditions, could contribute to the introduction of vulnerabilities within that software [35]. The Common Weakness Enumeration (CWE) is a project idealized to share information about weaknesses and how to fix them. Malicious hackers search for weaknesses in code and unfixed reports on bug trackers, and they try to get benefits for exploring them, damage the service availability, or sell the information in the black markets [14, 15]. However, finding and fixing configuration-related code weaknesses is a non-trivial task due to the complexity of configurable systems.

There are some tools that may help developers to detect code weaknesses, such as Cppcheck [1] and *FlawFinder* [52]. However, they do not consider the complete configuration space of a given configurable system, and the developers cannot select which configurations to test when using them. For example, the default behavior of Cppcheck is to test all possible configurations, but this is an impractical task for highly configurable systems. Previous work consider other kind of issues in their analysis, such as configuration-related syntax errors [31], configuration-related undeclared and unused identifiers [32], and type and compilation errors [2, 7, 8, 31, 32]. Other works [3, 37] consider code weaknesses in their analysis. They identify code weaknesses in the bug trackers of configurable systems, and study some aspects of them. Still, little effort has been put into studying configuration-related code weaknesses in detail, and into proposing tools to detect and fix them.

[1]For example, *Linux* developers suggest that *"code cluttered with* `#ifdefs` *is difficult to read and maintain. Don't do it."*

In this paper, we propose a sampling-based white-box technique to detect configuration-related weaknesses in configurable systems. It selects a subset of configurations to analyze individually using the *Linear Sampling Algorithm (LSA)* [29]. We use *LSA* in our study because it provides a good balance between the number of selected configurations and bug-detection capabilities [29]. We combine *LSA* with Cppcheck, a static analysis tool that is able to detect different kinds of code weaknesses, including memory leaks, resource leaks, buffer overflows, dereferences of null pointers, and uninitialized variables [1]. In our technique, we replace the default behavior of Cppcheck with the more promising *LSA* sampling approach to scale in practice. To evaluate our technique, we performed an empirical study on 24 popular open-source C configurable systems to better understand configuration-related weaknesses.

We focused on the following kinds of weaknesses: uninitialized variables (CWE-457),[2] null pointer dereferences (CWE-476), resource leaks (CWE-400 and CWE-775), buffer overflows (CWE-120), and memory leaks (CWE-400 and CWE-401). We considered systems used in practice, such as Apache Httpd, Cherokee, and Libssh. These systems are all highly configurable, and they heavily use the C preprocessor. by taking additional kinds of configuration-related weaknesses into account.

Overall, our technique found 57 configuration-related weaknesses in our corpus of subject systems. Specifically, they occur in 16 (67%) of the subject systems. We confirmed all configuration-related code weaknesses by searching for fixes in the corresponding software repositories (i.e., made by actual developers of the configurable system), and by submitting patches to developers (51 and 6 weaknesses, respectively). We excluded from the analysis the weaknesses that we cannot confirm as real ones. In total, 93% of the weaknesses involved one or two configuration options. Our results show that the numbers of configuration-related weaknesses that developers introduce when modifying existing code (51%) and introducing new functionality (49%) are fairly similar. This result differs from our previous findings, according to which developers introduce most configuration-related syntax errors (87%) when modifying existing code [31], and most configuration-related undeclared and unused identifiers (73%) when introducing new functionality [32].

The mains contributions of our study are:

- A technique to detect weaknesses in configurable systems considering five different types of weaknesses (Section 3);
- A dataset with 57 weaknesses that can be used by other researchers and practitioners. This dataset can be used by tool developers to improve their sbug finding tools (Sections 4 and 5).

2 Motivating Example

In this section, we describe an example of a configuration-related code weakness in Gawk,[3] a configurable system implementing a utility tool to rewrite text files based on patterns. Figure 1 presents an excerpt of the Gawk source code. This code contains a configuration option (controlled by macro `I18N`) that implements language internationalization. It is responsible for adapting the tool to specific regions or languages. As this code snippet contains one configuration option, we can generate two different configurations: the first one with macro `I18N` enabled (*Configuration 1*); and another configuration with macro `I18N` disabled (*Configuration 2*).

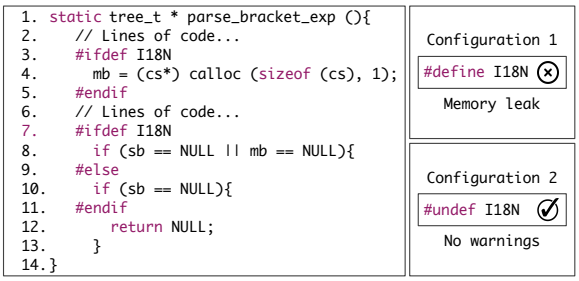

Figure 1: A code snippet of Gawk with a memory leak when we enable macro `I18N`.

When executing the code snippet of Figure 1 with `I18N` enabled (i.e., *Configuration 1*), it may cause a memory leak weakness (CWE-400 and CWE-401): the program allocates memory to variable *mb* at Line 4 but does not deallocate it when returning `NULL` at Line 12. This is a configuration-related weakness, as it occurs in *Configuration 1*, but it does not happen in *Configuration 2*. The consequences of such an issue depend on the application itself. It can be very dangerous if it occurs in a kernel-land process. In that context, a memory leak may lead to serious system stability issues [42]. While easy to spot in this small code excerpt, real configurable systems contain many configuration options, which make the analysis of every configuration infeasible. As a result, real configurable systems, for example, Linux [2], Apache Httpd [31, 32] and GCC [19], contain weaknesses that appear only in specific configurations.

Developers may detect a configuration-related weakness depending on the configurations they check. The majority of C development tools used in practice, including the compilers GCC and Clang, consider only one configuration at a time, that is, they analyze code after preprocessing. Because of this limitation, these tools may not show configuration-related warnings and compilation errors. Although these tools do not detect weaknesses, like the memory leak in the code depicted in Figure 1, developers still use them to detect this kind of bug [37]. Variability-aware tools, such as TypeChef [25], consider the complete configuration space when analyzing configurable systems. However, the time-consuming setup and compilation process of these tools hinder the analysis of large systems. Moreover, these tools also do not consider weaknesses.

Previous studies [2, 3, 7, 8, 18–20, 22, 32] considered configuration-related bugs, such as compilation and syntax errors. Moreover, researchers have discussed the problems of preprocessor directives, their negative influence on the development of tool support, and configuration-related bugs [27, 31, 32]. Other works [3, 37] study some aspects of code weaknesses found in bug trackers of configurable systems. However, little effort had been put into understanding configuration-related code weaknesses in details, and into proposing tools to detect and fix them.

[2]See more details at: https://cwe.mitre.org/data/definitions/*id*.html, where *id* is the number of the CWE.
[3]https://www.gnu.org/software/gawk

3 A Technique to Detect Configuration-Related Code Weaknesses

Next we present an overview of our technique (see Section 3.1). We explain each of its steps in Section 3.2.

3.1 Overview

Figure 2 illustrates our technique. It receives as input the source code of the configurable system, a sampling approach, the configuration-option constraints and build-system information available, and an analysis tool to check the source code. The constraints and build-system information are not required, though, as many C open-source configurable systems do not have these pieces of information available. The technique consists of three steps. First, it uses the *LSA* sampling approach to systematically select configurations (Step 1). Then, it verifies whether the selected configurations are valid ones according to the received build-system information (Step 2). Finally, it runs Cppcheck considering each selected configurations (Step 3). The technique output is the set of detected weaknesses. For each weakness, we perform an extra manual analysis, identifying whether they are configuration-related.

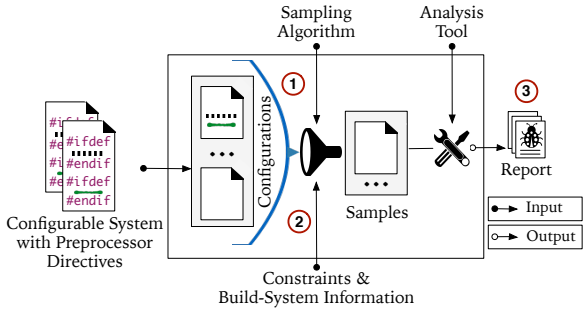

Figure 2: A white-box technique to detect configuration-related code weaknesses based on sampling.

3.2 Technique

To detect configuration-related weaknesses, we need to consider multiple configurations. Checking every configuration individually is often infeasible, because real-world C configurable systems have high numbers of configuration options, leading to configuration spaces of exponential sizes. For instance, Libxml2 has more than 2 thousand configuration options, and the Linux Kernel more than 12 thousand options. To tackle this scalability problem, we use a sampling-based approach to select a subset of configurations to analyze. That is, we preprocess the code to systematically generate some configurations and analyze each selected configuration individually [28, 29, 50].

Our technique starts by selecting each source file of the configurable system individually to perform a per-file analysis to reduce the number of selected configurations. *Step 1* uses a sampling approach to select configurations systematically. In this step, our technique can use different sampling approaches. In this paper, we use *LSA* [32] that combines the following algorithms:

- *One-disabled*: Abal et al. [3] suggested this approach based on 98 configuration-related bugs analyzed in the Linux kernel. It deactivates one configuration option at a time; it requires n configurations per file, where n is the number of configuration options in each source file.
- *One-enabled*: the approach is similar to *one-disabled*, but *one-enabled* activates one configuration option at a time. *One-enabled* also requires n configurations per file, where n is the number of configuration options in each source file.
- *Most-enabled-disabled*: this approach consists of activating all configuration options and then deactivating all options, which require two configurations per file.

LSA selects $2 + 2 * n$ configurations, in which n is the number of configuration options. For instance, *LSA* selects 10 configurations for a source file with 4 distinct configuration options. According to the results of a previous study [29], *LSA* increases the number of detected weaknesses, while reducing analysis effort (i.e., number of selected configurations).

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

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

Previous approaches evidence that sampling is effective to detect configuration-related bugs [2, 19, 31, 32]. However, the analysis of configurable systems, even using sampling, may not scale when we incorporate certain pieces of information, such as header files, build-system information, configuration option constraints, and global analysis [29]. In this sense, to make our technique scalable, we accepted some limiting assumptions. We perform per-file instead of a global analysis to reduce the number of selected configurations (i.e., our technique considers only configuration options within a single source file at a time). The per-file analysis used in our technique may yield some false positives and negatives. For instance, it may miss some weaknesses that may happen in more than one file. We may allocate a memory in one file and deallocate memory in another one. As another example, a file may allocate and deallocate a memory, but in some configurations, another file may change the control flow and not deallocate memory. We used a per-file analysis to make our analysis scalable. By using our technique, we can process an average file with 10-15 preprocessor macros, and about 700 lines of code, in between 7-10 seconds.

4 Study Setup and Results

We evaluated our technique in 24 highly configurable systems, such as Apache Httpd and Libxml2. In this section, we present the setup of our empirical study. First, we present the experiment definition (Section 4.1). Then, we present our subjects (Section 4.2).

Following, we present our experiment planning and instrumentation (Section 4.3). Finally, we describe our experiment main results (Section 4.4).

4.1 Definition

The goal of our experiment consists of analyzing our technique in order to evaluate code with preprocessor directives with respect to find configuration-related weaknesses from the point of view of researchers in the context of repositories of real configurable systems. In particular, we addressed the following research question:

- **RQ_1.** How many configuration-related code weaknesses can our technique detect?

To answer **RQ_1**, we counted the number of configuration-related weaknesses detected in each configurable system.

Answering this question is important to better guide the development of tools to detect and avoid weaknesses in practice. For example, the tools could take more effort to detect certain types of weakness than others. In addition, they could pay special attention to some specific configurations.

4.2 Subjects Selection

Overall, we analyze 24 systems written in C ranging from 20 to 2,126 configuration options. Libxml2 has the highest number of distinct configuration options (2,126) distributed across its source files, while Mpsolve has the lowest number of distinct configuration options (20). The configurable systems are from different domains, such as Web servers, text editors, databases, and games. We select these configurable systems guided by previous works [26, 27], which studied C configurable systems that are statically configurable with the C preprocessor (i.e., systems that use preprocessor conditional directives). We present details of each configurable system in Table 1, It indicates the configurable system name, application domain, lines of code, number of files, number of configuration options, number of configuration analyzed, number of distinct code versions (i.e., number of commits in the repositories), and the number of configuration-related weaknesses detected in our empirical study. Figure 3 presents the confidence interval of the number of options per analyzed configurable system.

4.3 Planning and Instrumentation

For the purpose of our study, we evaluate commits from the master branches of the repositories of each of the 24 systems. Further, we also consider previous versions of the source code using the Git software repositories of the configurable systems. To systematically select configurations, our technique (described in Section 3) uses *LSA* [32]. In Table 1, we present the number of configurations that *LSA* selects for each subject system. As *LSA* selects configurations per-file, the total number of configurations analyzed is the sum of the selected configurations of each source file. *LSA* ensures, in the absence of constraints, *pair-wise* and *three-wise* coverage of configuration options. That is, it selects a superset of configurations in which all combinations of two and three options are analyzed (i.e., *2-way* and *3-way* combinatorial interaction testing). For instance,

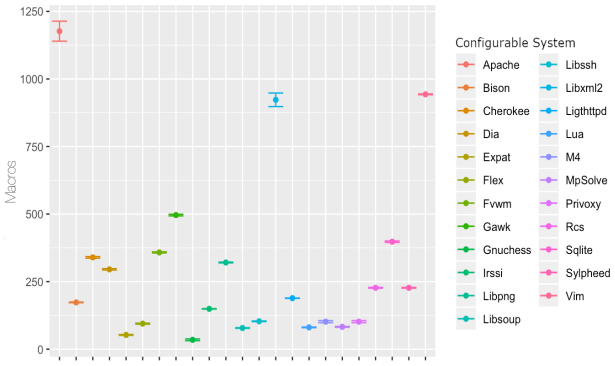

Figure 3: Confidence interval of the number of options per analyzed configurable system.

considering options *A* and *B* in Figure 4, we can see that there is a configuration where *A* and *B* are enabled (*configuration 7*); another configuration in which both *A* and *B* are disabled (*configuration 3*); and other configurations where only *A* or *B* is enabled (for example, *configurations 1* and *2*). The same situation occurs for configuration options *A* and *C*, *A* and *D*, *B* and *C*, and *B* and *D*. We can use the same rationale to see that *LSA* covers all combinations of three configuration options. *Three-wise* coverage is difficult to compute with constraints [9, 28, 29, 39, 40, 47], but *LSA* approximates *3-way* coverage with reasonable computation effort [29].

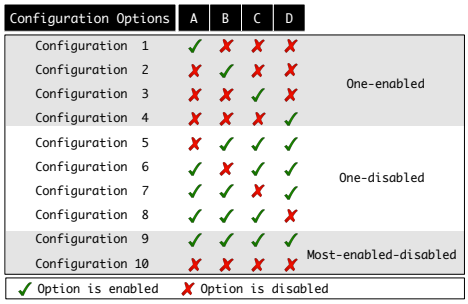

Figure 4: Selecting configurations systematically with *LSA*.

After selecting configurations for each source file, we use Cppcheck version 1.67 to detect weaknesses in each selected configuration on a per-file basis. We consider weaknesses of the following kinds: to uninitialized variables (CWE-457), null pointer dereferences (CWE-476), resource leaks (CWE-400 and CWE-775), buffer overflows (CWE-120), and memory leaks (CWE-400 and CWE-401). Cppcheck indicates the CWE type in its output [1]. We also use Git version 2.3.2 to retrieve the source code of the commits from the configurable systems. We count the lines of code and the number of files using the Count Lines of Code (CLOC) [10] tool version 1.56.

4.4 Results

Overall, we found some weaknesses in the subject systems. However, a number of them occur in invalid configurations and others are not configuration-related. For the purpose of this study, we

Table 1: Overview of the subject systems, including the total number of configuration-related code weaknesses.

Config. System	Domain	LOC	Files	Opt.	Config.	Vers.	Weaknesses
Apache Httpd	Web server	144,768	362	700	2,894	25,615	5
Bash	Interpreter	96,153	248	757	3,942	130	1
Bison	Parser generator	24,325	129	269	946	5,423	1
Cherokee	Web server	63,109	346	452	1,582	5,748	3
Dia	Diag. software	28,074	132	307	1,550	5,634	4
Expat	XML library	17,103	54	84	418	47	0
Flex	Lexical analyzer	16,501	41	130	366	1,607	0
Fvwm	Window manager	102,301	270	301	1,578	5,439	6
Gawk	GAWK interpreter	43,070	140	714	2,504	1,345	1
Gnuchess	Chess player	9,293	37	39	166	236	0
Irssi	IRC client	51,356	308	157	758	4,130	2
Libpng	PNG library	44,828	61	327	1958	2,188	3
Libsoup	SOUP library	40,061	178	92	448	2,005	0
Libssh	SSH library	28,015	125	115	718	2,915	13
Libxml2	XML library	234,9314	162	2,126	5,932	4,246	1
Lighttpd	Web server	38,847	132	215	906	1,470	3
Lua	Lang. interpreter	14,503	59	145	436	83	1
M4	Macro expander	10,469	26	106	330	953	1
Mpsolve	Math. software	10,278	41	20	136	1,434	0
Privoxy	Proxy server	29,021	67	158	880	63	0
Rcs	Revision control	11,916	28	97	366	915	0
Sqlite	Database system	94,113	134	467	3,322	553	0
Sylpheed	E-mail client	83,528	218	286	1,546	2,733	3
Vim	Text editor	288,654	178	942	8,170	5,720	9
Total		**1,525,220**	**3,476**	**9,006**	**41,416**		**57**

Lines of code; number of source files; number of compile-time configuration options; number of analyzed configurations; number of analyzed versions; and number of detected configuration-related weaknesses.

focused on analyzing the weaknesses that we could confirm by submitting patches to the configurable systems weakness trackers or by identifying fixes on the configurable systems repositories. This way, we found 57 configuration-related code weaknesses in 16 out of our 24 subject configurable systems (67%), as we present in Table 1. We counted in our statistics only weaknesses that developers fixed or weaknesses for which we submitted patches that developers accepted. Developers had previously fixed 51 weaknesses (89%) detected in our study. In total, we submitted 12 patches to fix weaknesses still in the code of 7 configurable systems, and developers accepted and fixed 6 (50%) of them. We found different types of configuration-related weaknesses, as we present in Figure 5: 30 memory leaks (53%), 10 uninitialized variables (18%), 9 dereferences of null pointers (16%), 6 resource leaks (11%), and 2 weaknesses related to buffer overflows (4%).

We further determined the number of configuration options involved in each configuration-related weakness. The majority of weaknesses involve one configuration option: 50 weaknesses (88% of the preprocessor-related weaknesses considered in our

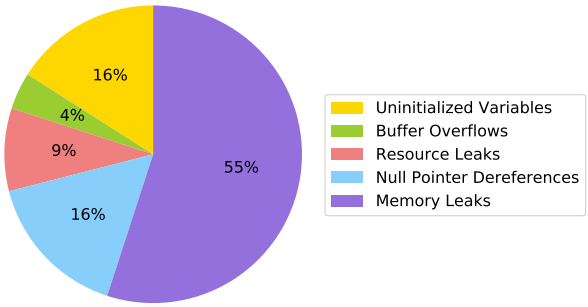

Figure 5: Types of configuration-related weaknesses.

study). Furthermore, we found three weaknesses that involve two configuration options; two weakness that relates to three options; one weakness that involves five configuration options; and one weakness that relates to seven options.

One weakness found by our technique is explained in Section 2. Next, we present an example of a configuration-related uninitialized

variable in Figure 6 for illustration. We found this uninitialized variable in Bash, and this weakness occurs only when we disable options TRACE and REGISTER, and enable option WATCH. In this configuration, variable ubytes is not initialized at Line 5, but used at Line 15. Technically, the value of an uninitialized non-static local variable is indeterminate in C, and accessing it leads to an undefined behavior. This weakness was detected in Bash 4.2 and fixed in Bash 4.4. We present more information about all weaknesses detected in our study at the supplementary website [51].

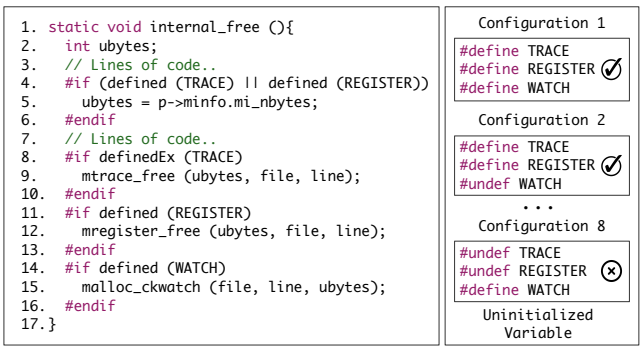

Figure 6: An example of a configuration-related uninitialized variable.

We performed an analysis of correlation to investigate similarities between the projects that we found weaknesses. We could not find a pattern in the 16 systems with weaknesses with respect to domain, LOC, number of configurations, number of versions, number of files, and number of optional features (see Table 1).

5 Discussion

In Section 5.1, we answer the research question. We present the patches submitted to the configurable systems in Section 5.2, and discuss the threats to validity in Section 5.3.

5.1 Research Question

RQ$_1$. How many configuration-related code weaknesses can our technique detect?

In total, we found 57 configuration-related code weaknesses in 16 out of the 24 C configurable systems considered in our study. We present the number of configuration-related weaknesses detected in each configurable system in Table 1. In our study, we found the following kinds of configuration-related weaknesses: memory leaks (30), uninitialized variables (10), dereferences of null pointers (9), resource leaks (6), and buffer overflows (2).

We found that most (93%) of the configuration-related weaknesses gathered in our study involved up two configuration options. In total, 50 of them involved one configuration option, three involved two options, two involved three options, one involved five options, and one involved seven options. Moreover, we found 30 memory leaks involving: one (26), two (2), three (1) and five (1) configuration options. The configuration-related weakness that involved seven configuration options is a resource leak that occurs in the source code of the Lua configurable system. One uninitialized variable that occurs in the source code of Apache Httpd involved two configuration options. One uninitialized variable that occurs in Bash involved three configuration options and we present in Figure 6. The other weaknesses of both resource leak and uninitialized variable kinds involved only one configuration option. All buffer overflows and null point dereference weaknesses also involved only one configuration option.

The kind of weakness that most occur in our study is memory leak (53%). It is an unintentional form of memory consumption whereby the developer fails to free an allocated block of memory when no longer needed. The consequences of such weakness depend on the application itself [42]. This kind of weakness is considered potentially dangerous if it occurs in a long-lived user-land application, as these applications continue to waste memory over time, eventually consuming all RAM resources. It may lead to abnormal system behavior. Furthermore, it is considered dangerous if it occurs in a kernel-land process. In this context, a memory leak may lead to serious system stability issues [42].

We found two buffer overflows in Bison (one) and Vim (one). This weakness occurs when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code [41]. Buffer overflow is classified with high severity [41], and it is considered the third most dangerous software error [33]. Furthermore, an attacker may intentionally trigger a resource leak. When successful, the attacker might be able to launch a denial of service attack by depleting the resource pool. In general, resource leak weaknesses have two common causes: error conditions and other exceptional circumstances; and confusion over which part of the program is responsible for releasing the resource [44]. Using our technique, we found six resource leaks in five configurable systems: Libssh (two), Lua (one), Lighttpd (one), Sylpheed (one) and Libxml2 (one).

In some languages such as C and C++, stack variables are not initialized by default. They generally contain junk data with the contents of stack memory before the function was invoked. An attacker can sometimes control or read these contents [34]. In total, we found ten uninitialized variables in Apache Httpd (three), Dia (two), Cherokee (one), Libssh (one), Sylpheed (one), Fvwm (one) and Bash (one). Moreover, a code may dereference a null pointer, thereby raising a *NullPointerException*. When successful, an attacker might be able to use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks [43]. We found nine null point dereference weaknesses in the source code of five configurable systems: Libssh (four), Irssi (two), Apache Httpd (one), Dia (one), and Fvwm (one).

The C preprocessor is flexible enough to allow developers to embrace any code fragment with preprocessor conditional directives, even a single token such as an opening bracket. This way, developers can introduce preprocessor conditional directives that

do not align with the underlying syntactic structure of the source code [25, 26]. Previous studies used terms such as undisciplined or incomplete preprocessor use to refer to preprocessor conditional directives of this kind [18, 27]. In our motivating example, presented in Figure 1, we illustrate an undisciplined use of the preprocessor. We can see at Lines 7, 9 and 11 that the conditional directives split up parts of the `if` statement. Undisciplined preprocessor use may influence the code quality negatively [5, 12, 30], and might ease the introduction of configuration-related weaknesses in practice. By analyzing the 57 configuration-related code weaknesses considered in our study, we found that 9 configuration-related weaknesses occur in undisciplined preprocessor directives. Some types of weaknesses may occur more often in the presence of undisciplined annotations.

Developers face configuration-related weaknesses in practice. Our results suggest that variability hinders the detection of configuration-related weaknesses.

5.2 Submitting Patches to Fix the Weaknesses

We submitted 12 patches [51]—for each code weakness not already fixed by developers—to 7 configurable systems: Apache Httpd (4), Dia (1), Gawk (1), Lighttpd (2), Libxml2 (1), Sqlite (2), and Sylpheed (1). We submitted these patches by using bug track systems, and e-mail lists. For every system, we read the documentation to identify the guidelines for submitting patches.

We consider that developers accept a patch when they mention that it is a weakness, or keep the patch open after updating some patch information, such as its priority. Conversely, we consider that developers reject the patch when they mention it is not a weakness, or update this information on the patch. Overall, developers accepted 6 (50%) out of 12 patches we submitted to the C configurable systems. We present information about the patches in Table 2, listing the name of the configurable system, file name with the weakness, the type of weakness, and the patch status. Developers already fixed all accepted patches.

Regarding the 6 rejected patches, the Apache Httpd developers rejected four patches: two weaknesses that arise in invalid configurations, and two false positives (in one case the developer stated that the reported null pointer dereference was a false positive, since the virtual storage address begins at 0,[4] and in the other case the Cppcheck did not consider a variable was static and reported a memory leak on it).[5] The Sqlite developers also rejected two patches as they were false positives. The developer mentioned that both reports were false positives because the build system ensures that the specific macros that cause the weaknesses are defined at that part of the code, avoiding any problem related to memory leak or uninitialized variable.

5.3 Threats to Validity

5.3.1 Construct validity The issue of whether the configuration-related code weaknesses detected are real weaknesses threatens construct validity. We addressed this threat by getting feedback from the actual developers, confirming each weakness reported in our statistics in two ways: (1) finding a fix in newer versions of the code; and (2) submitting patches to the configurable systems. Developers accepted and fixed 6 out of the 12 configuration-related code weaknesses reported, and we confirmed 51 weaknesses that developers fixed in newer versions of the source code.

Table 2: Patches submitted to the configurable systems.

Configurable System	File	Problem	Status
Dia	test-boundingbox.c	Uninit variable	Fixed
Gawk	regcomp.c	Memory leak	Fixed
Lighttpd	mod_dirlisting.c	Memory leak	Fixed
Lighttpd	mod_dirlisting.c	Resource leak	Fixed
Libxml2	catalog.c	Resource leak	Fixed
Sylpheed	jpilot.c	Resource leak	Fixed
Apache Httpd	ssl_util.c	Null deference	Rejected
Apache Httpd	mpm_winnt.c	Memory leak	Rejected
Apache Httpd	ap_regkey.c	Uninit variable	Rejected
Apache Httpd	ap_regkey.c	Uninit variable	Rejected
Sqlite	os_win.c	Uninit variable	Rejected
Sqlite	test_intarray.c	Memory leak	Rejected

5.3.2 Internal validity Regarding internal validity, the corpus of weaknesses is critical for our empirical study. Creating a representative corpus is difficult, primarily because we have no means of knowing all weaknesses in a given system. This is because there is no comprehensive quality assurance strategy in the first place. By using a tool to detect weaknesses, as we did with Cppcheck, we limit our study to weaknesses that this tool can detect, such as memory leaks, uninitialized variables, and null pointer dereferences. It is important to notice that we selected Cppcheck because it is a tool used in practice, it was mentioned in a number of interviews that we did with developers [30]. Further, we found developers mentioning it in commit messages in open-source projects.

To minimize this threat, we performed a study with the tools *CppCheck*, *Flawfinder*, *Splint*, *GCC*, and *Clang Analyzer*, and we considered 21 bugs reported the variability bugs database [3]. We run the tools by using their default configuration and we checked all possible configurations of the source code, as we used the simplified version of the bugs provided in the variability bugs database (that is, brute force is feasible in the simplified version). The results show that all tools do not detect the majority of bugs: *CppCheck* detected 08 (38%) bugs, *GCC* detected 05 (24%), *Flawfinder* detected 02 (10%) bugs, *Splint* detected 02 bugs (10%) also, and *Clang Analyzer* detected 04 (19%) bugs. In addition, we considered only the configuration-related weaknesses that we fully understand, this way we may miss some real configuration-related weaknesses.

Furthermore, our technique analyzes one file at a time. It does not find weaknesses that span multiple files. So, we may miss some code weaknesses (false negatives). Furthermore, our technique does not check all possible configurations, as we used sampling which checks only a subset of configurations. So, we might miss some weaknesses in configurations that we do not analyze. To minimize

[4] https://bz.apache.org/bugzilla/show_bug.cgi?id=56210
[5] https://bz.apache.org/bugzilla/show_bug.cgi?id=56211

this threat, we used a *LSA* sampling approach aiming at maximizing the number of detected code weaknesses [29].

We did not consider the build-system information of each subject system in our study. Build-system information is inherent difficult to analyze in a sound fashion [11, 38, 46], and most subject systems do not have this information available. In these systems, our results do not consider, for example, configuration-related weaknesses that occur in source files without preprocessor conditional directives that are included conditionally depending on configuration options.

Our technique considered only updated and added files to find configuration-related weaknesses in configurable system repositories, from the second to the last commit, as described in Section 3. However, this technique may lead to false negatives. For instance, developers may update a file *A*, which leads to weaknesses in a different file *B*. In our technique, because only *A* has been modified, we only analyze *A*. However, later, if developers modify *B*, our technique potentially catches the weaknesses. To minize this threat, we submit patches to the open-source systems and find a commit fixing the weaknesses. Thus, we only consider real weaknesses.

5.3.3 External validity We analyzed 24 configurable systems of different domains, sizes, numbers of configuration options, and numbers of developers, and found 57 configuration-related weaknesses in 16 of them. We selected mature C configurable systems used in industrial practice, but we also selected some younger configurable systems with a few developers to consider a broader range of configurable system's characteristics (see Figure 3). The corresponding communities exist for years and are active. This way, we reduce threats related to external validity.

6 Implications for Practice

In this section, we discuss some implications that our results bring to practice. First, our results agree with Muniz et al. [37] findings by showing that developers face configuration-related code weaknesses in practice. Previous works [30, 37] do not propose a technique to detect configuration-related weaknesses in the source code of configurable systems. Current state-of-the-art variability-aware tools need a time-consuming setup, such as TypeChef [25] and SuperC [21], hindering their application in practice. Thus, developers should put effort into the development of variability-aware analysis tools to minimize weaknesses in practice.

Second, our results support the conjecture [37] that most configuration-related code weaknesses involve one or two configuration options (93%). For instance, most of the memory leaks (28 out of 30) detected in our study involved no more than two configuration options. In addition, we found the following number of configuration-related memory leaks that involved up to two configuration options: nine null dereferences, nine uninitialized variables, two buffer overflows, and four resource leaks. This result is similar to previous studies regarding configuration-related bugs [2, 3, 19, 22, 29, 31, 32, 48]. Moreover, the *pair-wise* sampling approach [47] checks all combinations of two configuration options and would detect all configuration-related weaknesses involving one or two configuration options (93% of the code weaknesses considered in our study). So, based on this information, developers can start testing the source code using simple algorithms, and use other time-consuming algorithms, such as *three-wise* and *four-wise* sampling, only when testing the source code before releases.

Third, we found some differences regarding the way developers introduce configuration-related weaknesses. Developers normally introduce most configuration-related syntax errors (87%) when modifying code [31], and most configuration-related undeclared and unused identifiers when introducing (73%) new functionalities [32]. Our study shows that developers introduce configuration-related memory and resource leaks, buffer overflows, null pointer dereferences, and uninitialized variables by introducing new functionalities (51%) and modifying existing code (49%). Hence, instead of using only a particular technique to detect all kinds of configuration-related weaknesses and bugs, our results support the claim that we need different strategies and tools to properly catch them. For instance, we might need lightweight tools that check for configuration-related syntax errors on the fly, but more time-consuming code analysis with global information to detect other configuration-related weaknesses (applying it only when introducing new source files or before submitting new code versions to configurable system repositories).

Fourth, the corpus of weaknesses gathered in our work is a valuable source to further study configuration-related weaknesses, and to test and improve variability-aware analysis tools. Furthermore, we support Muniz et al. [37] findings while analyzing two kinds of weaknesses that they did not consider during their analysis (uninitialized variable and resource leak).

7 Related Work

Muniz et al. [37] conducted two studies to regarding the perception of developers of configurable systems with #ifdefs related to weaknesses, and the strategies and tools they use to identify and remove them. In the first one, they manually analyzed 27 configuration-related weaknesses of Apache Httpd, Linux and OpenSSL reported on their bug trackers. They found weaknesses of the following kinds: null pointer (six), integer overflow (six), memory leak (three), format string (two), race condition (one), risky cryptographic algorithm (one), and integer underflow (one). Most of their occurrences are null pointer and integer overflow. Furthermore, Muniz et al. [37] findings showed that all their 27 analyzed weaknesses involved up to two configuration options, this result is similar to ours. We found that 94.5% of the configuration related code weaknesses gathered in our study involved up to two configuration options. In their second study, Muniz et al. [37] conducted a survey with 110 developers of the previous configurable systems. They found that some developers do not check any configurations to detect weaknesses, and some senior developers could not identify integer and buffer overflows. We found two buffer overflows using our technique. In addition, they found that some developers do not use proper tools to detect weaknesses in configurable systems with #ifdefs. Our study results show evidences that our technique can detect configuration-related weaknesses in configurable systems.

Mordahl et al. [36] presented an empirical study of real-world variability bugs. To achieve this goal, they built a framework that simulates variability-aware by integrating four static analysis tools

(CBMC, Clang, Infer and Cppcheck) and applying this approach in three configurable systems (axTLS, Toybox and BusyBox). Our work uses the LSA algorithm in conjunction with Cppcheck on 24 configurable systems to identify weaknesses.

Abal et al. [3] manually studied 98 variability bugs of four C/C++ configurable systems, such as Linux and Apache. Sixty-two bugs are weaknesses (CWEs 078, 120, 125, 190, 197, 401, 403, 416, 440, 457, 476, 561, 563, 617, 675, 685, 764, 843). For each of the weaknesses, they analyzed relevant variability properties and summarized them into a self-contained C99 program with the same variability properties. These simplified weaknesses aid understanding the real weakness and constitute a publicly available benchmark for analysis tools. Also, they created simplified patches, and single-function versions of the weaknesses for evaluation of prototype and intraprocedural analyses. In addition, they provided the VBDb (Variability Bugs Database) containing the corpus of 98 variability bugs (some of them weaknesses). In our work, we propose a technique to automatically detect configuration-related weaknesses. It receives a link to a repository of a C/C++ system, and automatically evaluates each of their commits. As a result, we found 57 configuration-related weaknesses. We confirmed each configuration-related weakness by searching for fixes in the corresponding software repositories (i.e., made by actual developers of the configurable system), and by submitting patches to developers. In total, we submitted 12 patches, and developers accepted 50% of them. They already fixed all accepted patches. Our technique detects some resource leaks that they do not find (CWEs 400 and 775).

Ferreira et al. [13] conducted an empirical study of the Linux Kernel to analyze whether `#ifdefs` influence the occurrence of vulnerabilities. They investigated the relationship between configuration complexity and the occurrence of vulnerabilities according to some metrics. They counted the number of `#ifdefs` that appear inside a function. They considered how many distinct configuration options are used within a function. They analyzed the vulnerability history of functions, by checking whether a certain function has been touched by developers to fix past vulnerabilities. Their analysis revealed that vulnerable functions have, on average, 3.04 times more `#ifdefs` internally than non-vulnerable functions. Moreover, vulnerable functions have on average 4.2 times more configuration options internally than non-vulnerable functions. Vulnerable functions have fewer configuration options, and have, on average, a 1.3 times more outgoing function calls than non-vulnerable functions. Our work complements their work [13] by studying 57 configuration-related weaknesses in 16 configurable systems.

Halin et al. [22] evaluated all possible configurations (26,000+) of one version of JHipster, a popular code generator for web applications. They used a cluster of 80 machines during 4 nights for a total of 4,376 hours (182 days) CPU time. They found that over 35% of the configuration failed, and identified 6 feature interactions that caused 99% of these failures. The one-enabled, one-disabled, and most-enabled-disabled can detect 3.15, 2.34, and 0.67 out of 6 faults in their study, respectively. We observed differences regarding the way developers introduce the configuration-related weaknesses, providing insights for tool developers. Palix et al. [45] performed studies to detect faults in several versions of the Linux Kernel using Coccinelle. A previous study [6] presented the tool named Plug to detect memory leaks in C and C++ programs. However, these tools are not variability-aware. In our technique, we combined Cppcheck [1] with *LSA*.

Some researchers studied the way developers use the C preprocessor in practice [4, 5, 12]. Liebig et al. [27] analyzed 40 configurable systems and warned that developers can introduce subtle syntax errors, for example, by annotating a closing bracket but not the opening one. They classify this kind of preprocessor use as undisciplined. Other researchers proposed similar classifications, such as incomplete [16–18] and unstructured [12] preprocessor use. According to Liebig et al. [27], undisciplined preprocessor use makes up 15.6% of the total number of preprocessor directives. Medeiros et al. [30] interviewed 40 developers and performed a survey with 202 developers to understand why the C preprocessor is still widely used in practice despite the strong criticism the preprocessor receives in academia. They found that the majority of developers believe that undisciplined preprocessor usage influences code quality, maintainability and error-proneness negatively. All these studies discussed the C preprocessor and its problems, such as bugs, inconsistencies, and code quality. However, none of these studies took into consideration configuration-related code weaknesses. In our work, we performed an empirical study of 24 popular highly configurable systems using the C preprocessor at compile time to answer a number of research questions related to the number of configuration-related weaknesses.

8 Concluding Remarks

In this paper, we propose a sampling-based white-box technique to detect configuration-related weaknesses in the source code of configurable systems. To evaluate our technique, we performed an empirical study on 24 highly configurable systems. In particular, we answered one research question related to the frequency of configuration-related weaknesses.In summary, we found 57 distinct configuration-related code weaknesses in 16 out of 24 configurable systems considered in our study. Configuration-related code weaknesses may happen due to the complexity of detecting and fixing weaknesses related to preprocessor directives, especially when they involve a number of configuration options. However, it is important to quickly fix weaknesses, especially the ones with high severity, avoiding exposing the system to malicious attacks.

Our findings are important to support developers to understand these weaknesses, develop variability-aware analysis tools, minimize such weaknesses in practice, and improve software quality. As future work, we intend to extend this empirical study by considering different kinds of weaknesses (other CWEs). Further, we intend to analyze all files at the same time to try to identify weaknesses involving more preprocessor macros.

Acknowledgments

We would like to thank the anonymous reviewers. This work was partially supported by CNPq and CAPES grants.

References

[1] 2020. Cppcheck Design. http://cppcheck.sourceforge.net/.

[2] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the International Conference on Automated Software Engineering*. 421–432.

[3] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Transactions on Software Engineering and Methodology* 26, 3 (2018), 10:1–10:34.

[4] Ira D. Baxter. 1992. Design maintenance systems. *Commun. ACM* 35, 4 (1992), 73–89.

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

[6] Michael D. Bond and Kathryn S McKinley. 2008. Tolerating memory leaks. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications*. 109–126.

[7] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, and Leopoldo Teixeira. 2016. A Change-centric Approach to Compile Configurable Systems with #Ifdefs. In *Proceedings of the 15th International Conference on Generative Programming: Concepts & Experiences*. 109–119.

[8] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, Leopoldo Teixeira, and Sabrina Souto. 2018. A change-aware per-file analysis to compile configurable systems with #ifdefs. *Computer Languages, Systems & Structures* 54 (2018), 427–450.

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

[10] Al Danial. 2020. CLOC. http://cloc.sourceforge.net/.

[11] Christian Dietrich, Reinhard Tartler, Wolfgang Schroder-Preikschat, and Daniel Lohmann. 2012. A robust approach for variability extraction from the Linux build system. In *Proceedings of the Software Product-Line Conference*. 21–30.

[12] Michael Ernst, Greg Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *Transactions on Software Engineering* 28, 12 (2002), 1146–1170.

[13] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs influence the occurrence of vulnerabilities? An empirical study of the Linux kernel. In *Proceedings of the International Systems and Software Product Line Conference*. 65–73.

[14] Matthew Finifter, Devdatta Akhawe, and David Wagner. 2013. An empirical study of vulnerability rewards programs. In *Proceedings of the USENIX Conference on Security*. 273–288.

[15] Stefan Frei, Dominik Schatzmann, Bernhard Plattner, and Brian Trammell. 2010. *Modeling the security ecosystem - the dynamics of (In)security*. Springer US, 79–106.

[16] Alejandra Garrido and Ralph Johnson. 2002. Challenges of Refactoring C Programs. In *Proceedings of the International Workshop on Principles of Software Evolution*. 6–14.

[17] Alejandra Garrido and Ralph Johnson. 2003. Refactoring C with Conditional Compilation. In *Proceedings of the International Conference on Automated Software Engineering*. 323–326.

[18] Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. In *Proceedings of the International Conference on Software Maintenance*. 379–388.

[19] Brady Garvin and Myra Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the International Symposium on Software Reliability Engineering*. 90–99.

[20] Brady Garvin, Myra Cohen, and Matthew Dwyer. 2011. Using Feature Locality: Can We Leverage History to Avoid Failures During Reconfiguration?. In *Proceedings of the Workshop on Assurances for Self-adaptive Systems*.

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

[22] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2017. Test them all, is it worth it? A ground truth comparison of configuration sampling strategies. *arXiv preprint arXiv:1710.07980* (2017).

[23] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis Feasibility Study*. Technical Report. Carnegie Mellon University.

[24] Christian Kastner and Sven Apel. 2009. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.

[25] Christian Kastner, Paolo Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications*. 805–824.

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

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

[28] Jorg Liebig, Alexander von Rhein, Christian Kastner, Sven Apel, Jens Dorre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 81–91.

[29] Flávio Medeiros, Christian Kastner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering*. 643–654.

[30] Flávio Medeiros, Christian Kastner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming*. 999–1022.

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

[32] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. 2015. An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*. 35–44.

[33] Mitre. 2019. Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[34] Mitre. 2020. Uninitialized Variable. https://cwe.mitre.org/data/definitions/457.html.

[35] Mitre. 2020. Weaknesses. https://cwe.mitre.org/documents/glossary/index.html#Weakness.

[36] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the Foundations of Software Engineering*. 50–61.

[37] Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade, Baldoino Fonseca, and Márcio Ribeiro. 2018. A Qualitative Analysis of Variability Weaknesses in Configurable Systems with #Ifdefs. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. 51–58.

[38] Sarah Nadi and Richard Holt. 2014. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process* 26, 8 (2014), 730–746.

[39] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *Computing Surveys* 43, 2 (2011), 11:1–11:29.

[40] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Lecture Notes in Computer Science, Vol. 6287. 196–210.

[41] OWASP. 2020. Buffer Overflow. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

[42] OWASP. 2020. Memory Leak. https://owasp.org/www-community/vulnerabilities/Memory_leak.

[43] OWASP. 2020. Null Pointer Dereference. https://owasp.org/www-community/vulnerabilities/Null_Dereference.

[44] OWASP. 2020. Resource Leak. https://owasp.org/www-community/vulnerabilities/Unreleased_Resource.

[45] Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 305–318.

[46] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. 2013. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proceedings of the International Software Product Line Conference*. 91–100.

[47] Gilles Perrouin, Sagar Sen, and Jacques Klein. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Product Lines. In *Proceeding of the International Conference on Software Testing, Verification and Validation*. 459–468.

[48] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In *Proceedings of the International Conference on Software Engineering*. 632–642.

[49] Henry Spencer and Geoff Collyer. 1992. Ifdef Considered Harmful, or Portability Experience with C News. In *Proceendings of the USENIX Annual Technical Conference*. USENIX Association.

[50] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schroder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In *USENIX Annual Technical Conference*. 421–432.

[51] Our Team. 2020. Supplementary website. https://sbesweaknesses.github.io/.

[52] David Wheeler. 2020. FlawFinder. https://www.dwheeler.com/flawfinder/.