A Qualitative Analysis of Variability Weaknesses in Configurable Systems with *#ifdefs*

Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade
Federal University of Campina Grande
{raphael,larissanadja}@copin.ufcg.edu.br
rohit@dsc.ufcg.edu.br
wilkerson@computacao.ufcg.edu.br

Baldoino Fonseca, Márcio Ribeiro
Federal University of Alagoas
{baldoino,marcio}@ic.ufal.br

ABSTRACT

A number of critical configurable systems are implemented using *#ifdefs*, such as Linux. Some tools and strategies are proposed to avoid these directives. However, these systems still have weaknesses, leading to vulnerable code, and may impact millions of users. There is a lack of studies 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. Moreover, few works study the characteristics of weaknesses. To better understand the problem, we conduct two studies. In the first one, we qualitatively analyze 27 variability weaknesses of Apache HTTPD, Linux and OpenSSL reported on their bug trackers. In the second study, we conduct a survey with 110 developers of the previous configurable systems. Overall, our results show evidences that, although developers care about weaknesses, they may not detect some weaknesses reported in the bug trackers, and do not use proper tools to deal with them. They take on median 15 days and 4 discussion messages to solve them. Some weaknesses occur due to two feature interactions, and most of them can be detected by the *all macros enabled* sampling approach.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**; • **Software and its engineering → Preprocessors**; **Software product lines**;

KEYWORDS

Variability Weaknesses, Security, Configurable Systems, Preprocessor, *#ifdefs*, Survey

ACM Reference Format:
Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade and Baldoino Fonseca, Márcio Ribeiro. 2018. A Qualitative Analysis of Variability Weaknesses in Configurable Systems with *#ifdefs*. In *VAMOS 2018: 12th International Workshop on Variability Modelling of Software-Intensive Systems, February 7–9, 2018, Madrid, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3168382

VaMoS'18, February 7-9, Madrid, Spain

ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3168382

1 INTRODUCTION

Large Systems, such as Linux and OpenSSL, use C preprocessor directives, such as *#ifdefs*, to implement variability [22]. However, systems with many features may be difficult to evolve and maintain due to the number of possible configurations [13]. The scenario may become worse when these features interact, and may generate negative effects to the system code, such as introducing weaknesses. Weakness is a type of mistake in software that, in proper conditions, could contribute to the introduction of vulnerabilities within that software.[1] We call it a variability weakness when it occurs in some configurations but not in others [1]. We only consider variability via *#ifdefs*.

Malicious hackers search for weaknesses in code and unfixed reports on bug trackers, and try to get benefits for exploring them, damage the service availability, or sell the weaknesses information in the black markets [7, 8]. In 2017, developers already reported more than 10 thousand vulnerabilities to the National Vulnerability Database (NVD).[2] A vulnerability is a weakness in the system that might be exploited to cause loss or harm [21]. The Common Vulnerabilities and Exposures (CVE) is a project idealized to share information about vulnerabilities and how to fix them using an index enumeration.[3] In addition, the Common Weakness Enumeration (CWE) is a similar project regarding vulnerabilities.[4]

Ferreira et al. [6] perform an empirical study of the Linux Kernel regarding the bad influence of preprocessor directives (*#ifdefs*) in maintainability and comprehension of code. They investigated the relationship between configuration complexity and the occurrence of vulnerabilities according to some metrics. However, they were not concerned with developers' background about weaknesses regarding *#ifdefs*, and how to detect them. Abal et al. [2] classify bugs reported to bug trackers. They classify them with respect to their nature and number of occurrences. Their results show in what ways variability affects and increases the complexity of software bugs. However, they do not discuss about weaknesses. There are some tools proposed that may help developers to detect weaknesses, such as CPPCheck[5] and FlawFinder.[6] They can use sampling approaches to select which configurations to test when using these tools [17]. However, there is a lack of evidences regarding whether developers use the tools.

[1] https://cwe.mitre.org/documents/glossary/index.html#Weakness
[2] https://nvd.nist.gov
[3] http://cve.mitre.org/
[4] https://cwe.mitre.org
[5] http://cppcheck.sourceforge.net/
[6] https://www.dwheeler.com/flawfinder/

In this work, we perform an empirical study to better understand the developers' background in detecting weaknesses, and the tools and strategies they use to perform this activity. We analyze variability weaknesses reported on configurable systems with *#ifdefs*, such as Apache HTTPD, Linux, and OpenSSL. We manually analyzed 27 variability weaknesses of eight kinds. Developers take on median 15 days and 4 discussion messages to fix them. Moreover, we also analyze their priority level, and presence conditions. Furthermore, we conduct a survey with 110 developers of the above mentioned configurable systems. We ask them to select between two code styles (with and without weaknesses), how they detect weaknesses, and their experience. The complete results of both studies are available online.[7]

Developers do not report weaknesses relating them to CWEs. Besides, the majority of developers do not detect some kinds of weaknesses in configurable systems. Furthermore, despite developers care about weaknesses, they do not use proper tools to detect weaknesses in configurable systems. The main contributions of this paper are:

- A qualitatively analysis of 27 variability weaknesses of three real configurable systems (Section 3);
- A survey with 110 developers to better understand their background and how they detect weaknesses (Section 4).

We organize the remainder of this paper as follows. In Section 2, we present a motivating example. Section 3 describes our study to analyze bug reports regarding variability weaknesses related to *#ifdefs*. Section 4 presents a survey to analyze developers experience in detecting weaknesses, tools supporting, and strategies to detect them. Finally, we relate our work to others in Section 5, and present concluding remarks in Section 6.

2 MOTIVATING EXAMPLE

In this section, we describe an example of a high priority variability weakness in Linux (Bug id: `60570`). Figure 1a presents a simplified code snippet of the `printk.c` file of Linux (commit `2f90b68`). It declares and initializes the `print` variable with 1 when `CONFIG_PRINTK_TIME` is enabled. Otherwise, the variable is declared but not initialized. Following, if the variable has the value 1 or if the `buf` pointer is `NULL`, the function will return 0 or 15, respectively. Otherwise, the `do_div` function divides the `ts` variable by 10^9. Next, the `sprintf` function sends formatted output to a char buffer, instead of printing it on console as the `printf` function does. The function receives the buffer (`buf`), the string format ("[%5*lu*.%06*lu*]"), and the inputs (`(long)ts` and `sec/1000`). In this example, the string format received as parameter allows the output to have up to 15 characters.

A Buffer Overflow occurs in Line 10 if the input size is larger than the string format size, which happens if the value of the `ts` variable is greater than 99,999. This way, the data may be written outside the buffer bounds, and may cause the execution of malicious code [11]. It is classified as the CWE-120: Buffer Copy without Checking Size of Input (Classic Buffer Overflow). This kind of weakness is the third most dangerous software error.[8] Buffer Overflow weaknesses are related to some CVEs, such as CVE-2004-0200 and CVE-2001-0050. In addition, it is a variability weakness since it occurs when `CONFIG_PRINTK_TIME` is enabled, but it does not occur when it is disabled.

In total, the real `printk.c` file contains 15 macros. By performing a per-file analysis, we identified that this weakness can be detected by using the *all macros enabled* sampling approach. Figure 1b presents the code fix that removes the weakness.

Previous studies [13, 17, 19] proposed tools and strategies to detect weaknesses. Despite that, developers continue to develop configurable systems containing weaknesses. For instance, developers already reported more than 10 thousand vulnerabilities to the NVD in 2017. Moreover, there is still a lack of studies to verify whether developers use proper tools in practice, and how developers test configurable systems with *#ifdefs* to detect weaknesses. In this work, we study some reported weaknesses (Section 3), and we conduct a survey with developers to evaluate their background and how they detect weaknesses in configurable systems (Section 4).

3 STUDY I: VARIABILITY WEAKNESSES

In this section, we evaluate 27 variability weaknesses reported to bug trackers of real configurable systems. First, we present the study definition (Section 3.1) and planning (Section 3.2). Next, Section 3.3 presents and discusses the results. Finally, Section 3.4 describes some threats to validity.

3.1 Definition

The goal of this study consists of analyzing bug reports for the purpose of studying variability weaknesses with respect to their characteristics from the point of view of C developers in the context of configurable systems with *#ifdefs*. We address the following research questions:

- RQ_1**:** How many kinds of weaknesses occur on the analyzed configurable systems with *#ifdefs*?
  For each analyzed variability weakness, we classify it based on CWEs.
- RQ_2**:** How critical are the analyzed variability weaknesses?
  For each analyzed variability weakness, we identified its priority level based on bug reports.
- RQ_3**:** Which sampling approaches can detect the analyzed variability weaknesses?
  For each analyzed weakness, we identify the sampling approaches that can detect it.
- RQ_4**:** How many weaknesses occur inside *#ifdefs*?
  The analyzed weaknesses occur inside *#ifdefs*, or outside but depend on them to happen. For each analyzed weakness, we identify where it occurs.
- RQ_5**:** How long do variability weaknesses take to be fixed?
  For each analyzed weakness, we count how many days it takes to be fixed in bug reports.
- RQ_6**:** How many discussion messages in the bug reports are necessary before a variability weakness to be fixed?
  For each analyzed weakness, we identify the number of discussion messages until it is fixed.

[7] http://www.dsc.ufcg.edu.br/~spg/vamos18-weakness/
[8] http://cwe.mitre.org/top25/

```
1  //...
2  bool print;
3  #ifdef CONFIG_PRINTK_TIME
4    print = 1;
5  #endif
6  size_t time(u64 ts, char *buf){
7    //...
8    if (!print) return 0;
9    if (!buf) return 15;
10   sec = do_div(ts, 1000000000);
11   return sprintf(buf,"[%5lu.%06lu]",
12                  (long)ts, sec/1000);
13 }
14 //...
```

(a) Code snippet with variability weakness.

```
1  //...
2  bool print;
3  #ifdef CONFIG_PRINTK_TIME
4    print = 1;
5  #endif
6  size_t time(u64 ts, char *buf){
7    //...
8    if (!print) return 0;
9    sec = do_div(ts, 1000000000);
10   if (!buf)
11     return sprintf(NULL,"[%5lu.000000]",
12                    (long)ts);
13   return sprintf(buf, "[%5lu.%06lu]",
14                  (long)ts, sec/1000);
15 }
16 //...
```

(b) Code snippet without variability weakness.

Figure 1: Code snippets of Linux with a Buffer Overflow and the fixed version.

3.2 Planning

Subjects Selection. We perform an analysis of the bug reports of Apache HTTPD and Linux. In addition, we evaluate reports of the OpenSSL GitHub. Linux is a computer operating system. OpenSSL is an open source project that provides a robust toolkit for the Transport Layer Security and Secure Sockets Layer protocols. It is also a general-purpose cryptography library. Apache HTTPD is the core technology of the Apache Software Foundation. It is responsible for projects involving web-based transmission technologies, data processing, and execution of distributed applications. The analyzed configurable systems are widely used. Therefore, an exploited vulnerability in one of them may affect a large number of users.

Methodology. We search for weakness in bug trackers. First, we used the keywords CWEs and CVEs. However, few developers use them when reporting a weakness. So, we select keywords from the 24 deadly sins that are related to the C programming language. We select two kinds of sins: implementation and cryptographic [10]. For example, integer overflow is the seventh sin, and it is associated to the following CWEs: Incorrect Calculation (CWE-682), Integer Overflow or Wraparound (CWE-190), Integer Underflow (CWE-191), and Integer Coercion Error (CWE-192). We use the description of the CWE[9] to search for variability weaknesses.

We manually classified each bug report yielded by our search using keywords as true positive, false positive or inconclusive. When the keyword is not related to the expected weakness kind, we classified the bug report as false positive. In addition, some weaknesses are not related to variability. We also classified them as false positives. We ignore the build-system for counting the layers of *#ifdefs* in our study. We considered the bug report as inconclusive when we could not relate it to some CWE, or when we could not fully understand the weakness. Table 1 shows the variability weaknesses (true positives) considered in our study.

We analyzed each variability weakness regarding its number of macros, and whether it occurs inside or outside an *#ifdef*. We perform a per file analysis to identify the sampling approach that can detect it. We do not consider feature models. For example, if a weakness occurs inside an *#ifdef* M1 with no nested *#ifdefs*, only M1 needs to be enabled to detect the weakness. Both *all macros enabled* and *one macro enabled* sampling approaches can detect the weakness. However, we reported the sampling approach with less effort in terms of number of configurations. In our example, we report that the *all macros enabled* sampling approach can detect the weakness.

Configurable System	Keywords	File	Weakness	IM	VL	NM	Samp.	#ifdef
Apache	Format String	htdbm.c	Format String	1	1	0	AE	Inside
	Overflow	ab.c	Integer Overflow	1	1	0	AE	Outside
	Weakness	htpasswd.c	Crypto Algorithm	1	1	0	OD	Outside
Linux	Overflow	ring_buffer.c	Integer Overflow	1	1	0	AE	Inside
	Overflow	printk.c	Buffer Overflow	1	2	0	AE	Outside
	Overflow	filemap.c	Buffer Overflow	1	1	0	AE	Outside
	Overflow	output_core.c	Integer Overflow	1	1	0	AE	Inside
	Overflow	hid-picolcd_core.c	Buffer Overflow	1	1	0	AE	Inside
	Strlen	sysfs.c	Integer Overflow	1	1	0	AD	Inside
	Overflow	kobject.h	Buffer Overflow	2	2	1	OD	Inside
	Memory Leak	page-flags.h	Memory Leak	1	1	0	AD	Inside
	Null Pointer	intel_sdvo.c	Null Pointer	1	1	0	AE	Outside
	Null Pointer	auditsc.c	Null Pointer	1	1	0	AE	Inside
	Null Pointer	blk-mq.c	Null Pointer	1	1	0	AE	Outside
	Null Pointer	ray_cs.c	Null Pointer	1	1	0	AE	Outside
	Null Pointer	tcp_ipv4.c	Null Pointer	1	1	0	AE	Outside
	Memory Leak	ipv6_sockglue.c	Memory Leak	1	1	0	AE	Outside
	Overflow	mmap.c	Integer Overflow	2	1	0	AE	Outside
	Overflow	addr.c	Buffer Overflow	1	1	0	AE	Inside
	Overflow	page_alloc.c	Buffer Overflow	1	1	0	AE	Inside
OpenSSL	Null Pointer	statem_clnt.c	Null Pointer	1	2	1	AD	Inside
	Overflow	b_print.c	Format String	2	2	0	OD	Outside
	Attack	encode.c	Integer Underflow	1	1	0	AE	Outside
	Memory Leak	d1_pkt.c	Memory Leak	1	1	0	AD	Outside
	Overflow	srp_lib.c	Buffer Overflow	1	1	0	AD	Inside
	Race Condition	t1_lib.c	Race Condition	1	1	0	AE	Inside
	Overflow	t1_lib_.c	Integer Overflow	1	1	0	AD	Inside

Table 1: Analyzed variability weaknesses. Keyword = Keyword that yields the bug report with the weakness; IM = Number of macros that yield the weakness; VL = Variability Level of the weakness; NM = Number of Nested Macros; Samp. = Sampling Approach that can detect the weakness; AE = All Enabled; AD = All Disabled; OD = One Disabled; *#ifdef* = Whether the weakness occurs inside or outside an *#ifdef*.

[9] http://cwe.mitre.org/data/definitions/682.html

Configurable Systems	Files	LOC (Avg)	Macros		
			Total	Avg	Median
Apache HTTPD	12	893	70	23.33	17
Linux	43	1,145	141	8.29	5
OpenSSL	16	1,438	103	14.71	6
Total	**71**	**3,476**	**314**	**46.33**	**28**

Table 2: Median and total of time and discussion messages to fix a variability weakness per analyzed configurable systems. Time to Fix/Total and Median = Sum and Median of days between the report being created and the weakness being fixed; Discussion Messages/Total and Median = Sum and Median of Discussion Messages in the bug reports.

3.3 Results and Discussion

The keywords that returned more bug reports are: Memory Leak (293), Overflow (274), Strlen (124), and Attack (102). Some keywords returned a low number of bug reports, such as: CWE (3) and Weakness (18). In total, we analyzed 27 variability weaknesses. The weakness kind with most occurrences is Buffer Overflow (seven). We also identified 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). In total, 17 weaknesses of four kinds occur in Linux (6 Buffer Overflows, 5 Null Pointers, 4 Integer Overflows, 2 Memory Leaks). Moreover, seven out of the eight kinds of weaknesses occur in OpenSLL. The Risky Cryptographic Algorithm weakness only occurs in Apache HTTPD, in which Format String and Integer Overflow also occur (once each).

Malicious agents can negatively explore weaknesses [3]. For example, a malicious agent that seeks to gain partial or total control of a host at Internet can exploit a Buffer Overflow to insert extra data encoded with specific instructions in a buffer. The extra data can flood the buffers, causing an overage and compromising the integrity of the original data. The implemented data has now infiltrated most buffers, giving the agent the capability to access other files on the network, manipulate programming or delete important data.

Observation 1: Most analyzed variability weaknesses are Buffer Overflows and Null Pointers.

We identified each analyzed variability weakness whether it occurs inside or outside an *#ifdef*. In total, 14 (51.85%) of them occur inside *#ifdefs*, while 13 (48.15%) occur outside these clausules. Moreover, 14.82% of them are inside two nested *#ifdefs*. Furthermore, the variability weaknesses involve one (89.65%) or two (10.35%) macros.

The statement where the weakness itself occurs may be inside or outside an *#ifdef*. Figure 2 presents a code snippet with a Division-by-Zero weakness. The weakness occurs outside an *#ifdef* (Line 7). However, it happens when the `M1` macro is disabled. All variability weaknesses that occur in Apache involved only one macro. One of them occurred inside an *#ifdef*, while the other two weaknesses occurred outside.

```
1 #ifdef M1
2   int var1 = 10;
3 #else
4   int var1 = 0;
5 #endif
6 int main(){
7   int var2 = 10/var1;
8 }
```

Figure 2: Code snippet of a Division-by-Zero variability weakness that occurs outside *#ifdefs*.

Observation 2: The analyzed variability weaknesses involve up to two macros.

In addition, we identified the sampling approaches that can detect the analyzed variability weaknesses. In total, 19 (70.37%) of the analyzed variability weaknesses can be detected by the *all macros enabled* sampling approach, 5 (18.51%) by the *all macro disabled* sampling approach, and 3 (11.12%) by the *one macro disabled* sampling approach. The Buffer Overflow weakness presented in Figure 1a can be detected by the *all macros enabled* sampling approach, while the Division-by-Zero weakness presented in Figure 2 can be detected by the *all macro disabled* sampling approach. Table 1 presents the reported sampling approaches that can detect the analyzed weaknesses (see Column Sampling).

Some tools can detect some kinds of weaknesses. For example, CppCheck can detect Buffer Overflow, Integer Overflow, Null Pointer, and Memory Leak. However, developers have to identify the configurations to be tested. By default, it analyzes the code using the *one enabled* sampling approach.

Observation 3: The *all macros enabled* sampling approach can detect most analyzed variability weaknesses.

The analyzed variability weaknesses are identified as high, normal, and low priority in their bug reports. In total, 40.74% of them have high priority, and 59.25% have normal priority. The developers did not indicate the priority level of a weakness in Linux. Furthermore, none of them have low priority. It seems that developers recognize the importance of this security problems.

The analyzed variability weaknesses took on median 4 discussion messages and, 15 days to be fixed. In some cases, developers took 720 days to fix a Buffer Overflow in the Linux project. As another example, some high priority Memory Leaks took more than 60 days to be fixed in the Linux project. The system was vulnerable during these days. This may happen due to the complexity of detecting and fixing weaknesses related to *#ifdefs*, specially when they involve a number of macros and are inside nested *#ifdefs*. Developers must quickly fix some bugs to avoid the system to be exposed.

Observation 4: No analyzed variability weakness has low priority.

3.4 Threats to Validity

We analyzed a subset of the returned bug reports in the search due to time constraints. Analyzing different bug reports may change our observations. As future work, we intend to analyze more bug reports and kinds of weaknesses. We could not analyze some bug reports, since we did not understand them. Analyzing them also may change our observations. Furthermore, we analyzed three configurable systems in different domains, but we intend to analyze other systems as future work. Finally, developers report few weaknesses using the CWE and CVE terminology. We manually analyzed each bug report and related them to CWEs. It is an error-prone activity since it depends on our knowledge.

4 STUDY II: SURVEY

In this section, we survey developers that contribute to the configurable systems considered in Study I. First, we present the study definition (Section 4.1) and planning (Section 4.2). Next, Section 4.3 presents and discusses the results. Finally, Section 4.4 describes some threats to validity.

4.1 Definition

The goal of this study consists on analyzing developers' background for the purpose of evaluating their background in weaknesses on configurable systems with *#ifdefs*, and the tools and strategies used with respect to identifying weaknesses from the point of view of C developers in the context of configurable systems with *#ifdefs*. We address the following research questions:

- RQ_7: Which tools do developers use to detect weaknesses in configurable systems?
- RQ_8: Which sampling approaches do developers use to detect weaknesses in configurable systems?
- RQ_9: How many developers identified weaknesses in survey's code snippets?
  We asked developers to choose between code with and without weakness (Format String, Integer Overflow, Null Pointer Dereference, and Buffer Overflow) reported in bug trackers of configurable systems.

4.2 Planning

The survey contains two parts. In the first one, developers choose the code style (C code snippets with or without variability weaknesses) that they prefer. We select five weaknesses in bug trackers of Apache HTTPD, Linux, and OpenSSL. All of them are fixed. One code snippet contains a weakness and the other one shows how developers fixed them. Figure 1 presents code snippets similar to one question of the survey. We include one question related to CWE-120 Buffer Copy without Checking Size of Input (rank 3), two questions related to CWE-131 Incorrect Calculation of Buffer Size (rank 20), one question related to CWE-134 Uncontrolled Format String (rank 23), and one question related to CWE-190 Integer Overflow or Wraparound (rank 24). The Top 25 Most Dangerous Software Errors indicates their rank.[10]

In the second part of the survey, developers answer questions about their experience, tools, and configurations used to detect weaknesses. They classify their experience in a scale from none (zero) to high (five). Moreover, they indicate the tools and sampling approaches used to detect weaknesses. We provide the following tool options: CppCheck, GCC,[11] FlawFinder, RATS,[12] Clang,[13] and other – where they can indicate other tools.

4.3 Results and Discussion

We sent emails to 4,079 developers of Apache HTTPD, Linux, and OpenSLL, and 110 of them answered our survey, achieving a response rate of 2.69%. Many developers have experience in detecting weaknesses (see Figure 4b), and 83.40% of them have more than three years of experience with C preprocessor directives, such as *#ifdefs* (see Figure 4c). In total, only 4.54% of developers do not care about not introducing weaknesses in commits (see Figure 4a).

The majority of them chose code snippets without the Format String (see Figure 3a), Integer Overflow (see Figures 3b and 3e), and Null Pointer Dereference (see Figure 3b) weaknesses. However, some developers prefer code snippets containing the Buffer Overflow weakness (see Figure 3d). Most developers that incorrectly answer this question care about weaknesses in configurable systems, and 32.72% of them have more than five years of experience.

Three developers (3.60%), that care about detecting weaknesses and have more than five years of experience, chose the code snippet containing Format String. Moreover, some developers (37.80%) chose the code snippet containing an Integer Overflow, or liked both code snippets. Twenty-nine of them (26.36%) declared to have more than five years of experience. One developer with 3-5 years of experience preferred a code snippet containing the Null Pointer Dereference weakness.

> **Observation 5**: A number of senior developers could not detect Integer and Buffer Overflows.

Many developers check the *all macros enabled* (21.90%) and *all macros disabled* (14.70%) sampling approaches to verify the presence of the weaknesses. However, many developers (24.70%) do not check any configuration (see Figure 4e). In total, 25 out of 75 developers that have more than five years of experience working with *#ifdefs* stated they use *all macros enabled* sampling approach. In total, four developers have less than a year of experience. Two of them do not test any configuration, one use the *all enabled* and *all disabled* sampling approach, and one uses Checksec,[14] a bash script to check the properties of executables.

> **Observation 6**: Some developers do not check any configuration to detect weaknesses.

Some developers (11.50%) do not use any tool to detect weaknesses in the code (see Figure 4d). The most used tools are GCC (38.80%), CLang (22.80%), and CPPCheck (10.50%). Some of the tools are not variability-aware. Moreover, some tools do not associate bugs to CWEs, and cannot detect some of the weaknesses considered in our survey. Most developers with less than a year

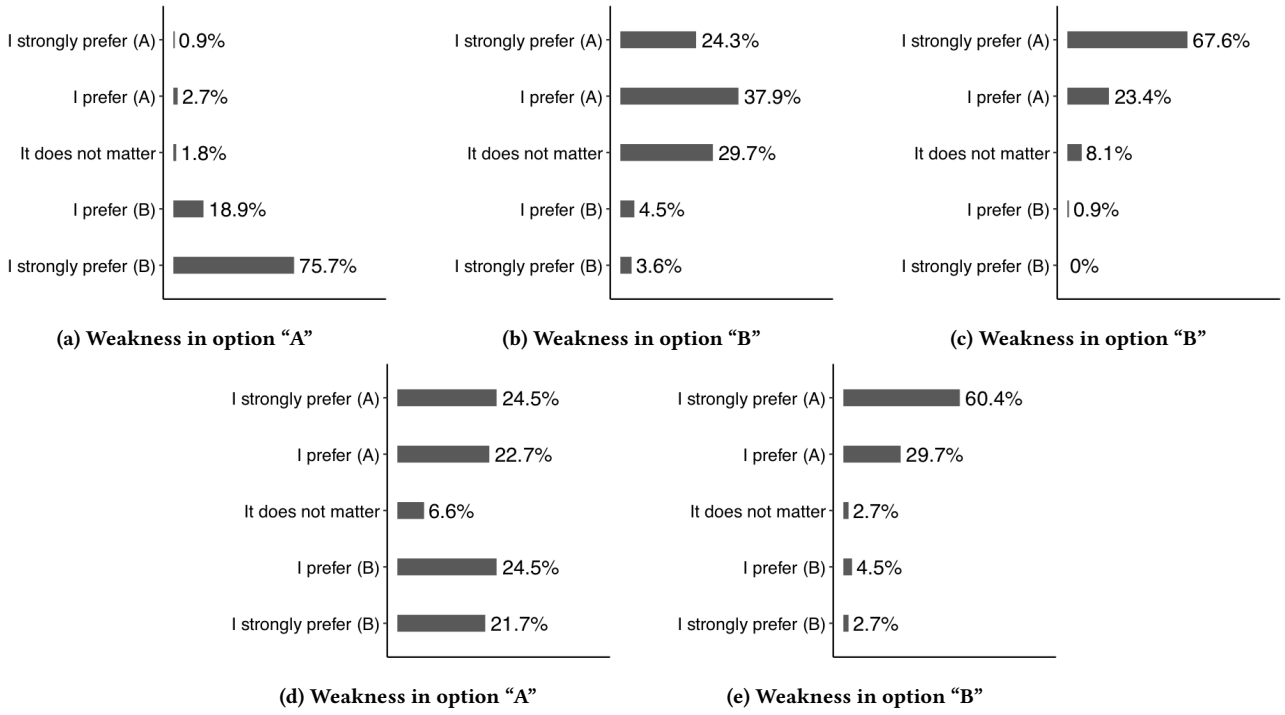
[10] http://cwe.mitre.org/top25/

[11] https://gcc.gnu.org/

[12] https://github.com/andrew-d/rough-auditing-tool-for-security

[13] https://clang.llvm.org/

[14] http://www.trapkit.de/tools/checksec.html



(a) Weakness in option "A" **(b) Weakness in option "B"** **(c) Weakness in option "B"**

(d) Weakness in option "A" **(e) Weakness in option "B"**

Figure 3: Results of the questions about code style preference regarding weaknesses: (a) Format String, (b) Integer Overflow, (c) Null Pointer Dereference, (d) Buffer Overflow, (e) Integer Overflow.

of experience use GCC and Clang to detect weaknesses in configurable systems. Some developers (56.63%) have more than five years of experience and also use GCC and Clang for this task. Despite that developers care about weaknesses, a number of them do not use proper tools to detect weaknesses in configurable systems. There are better tools than compilers to detect weaknesses, such as CppCheck and FlawFinder.

Observation 7: Some developers do not use proper tools to detect weaknesses in configurable systems with *#ifdefs*.

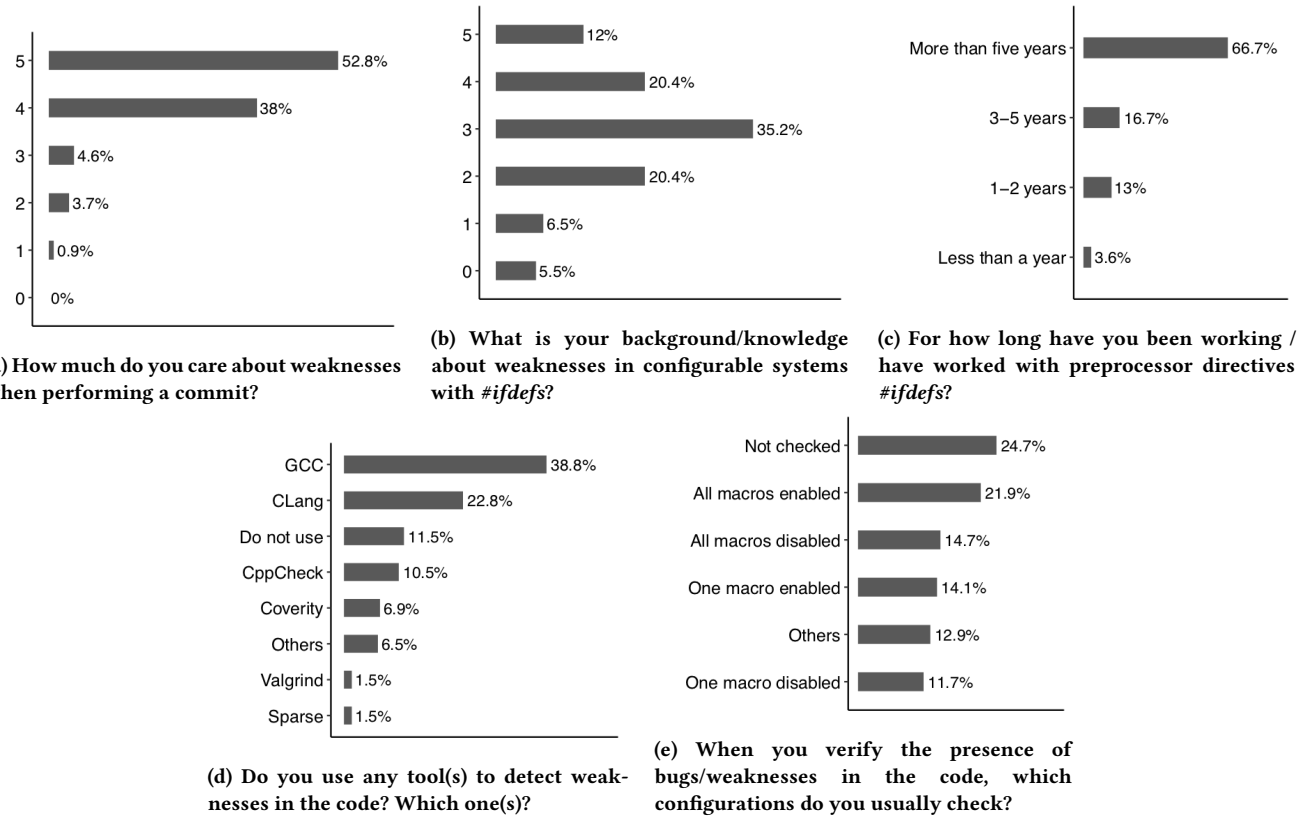
4.4 Threats to Validity

Developers may misunderstand the code snippets in our survey, since we did not include all files. Although the questions contain real weaknesses of configurable systems, the results of this study can only be interpreted in the context of the considered code snippets. We intend to survey more developers as future work.

5 RELATED WORK

Ferreira et al. [6] 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 [6] by studying 27 variability weaknesses reported in bug trackers of Linux, Apache HTTPD, and OpenSSL. We analyze some metrics that are not analyzed by them, such as number of days to fix a bug, priority level, number of nested *#ifdefs*. In addition, we also performed a survey with developers of the configurable systems. We identified that some of them (11.5%) do not use any tool to detect weaknesses, and some developers cannot detect Buffer and Integer Overflows involving *#ifdefs*.

Chowdhury and Zulkernine [5] aimed at verifying the relationship between complexity metrics and the vulnerability occurrences by studying releases and more than four years of vulnerability history of Mozilla Firefox. They found that complexity, coupling, and lack of cohesion metrics positively correlate to the number of vulnerabilities at a statistically significant level. Neuhaus et al. [20] performed a study to verify if components that shared similar sets of function calls are more propitious to be vulnerable. We used different kinds of metrics in our study and performed a survey with developers different from their work [5, 20]. We only analyze variability weaknesses that involved up to two macros, and some of them (17.25%) occur inside two nested *#ifdefs*. We identified



(a) How much do you care about weaknesses when performing a commit?

(b) What is your background/knowledge about weaknesses in configurable systems with *#ifdefs*?

(c) For how long have you been working / have worked with preprocessor directives *#ifdefs*?

(d) Do you use any tool(s) to detect weaknesses in the code? Which one(s)?

(e) When you verify the presence of bugs/weaknesses in the code, which configurations do you usually check?

Figure 4: Results of our survey about developers' background and tools to detect weaknesses.

that most of the analyzed weaknesses can be detected by the *all macros enabled* sampling approach (70.37%), but only 21.90% of the developers use this approach when testing the systems codes.

In the context of patches to fix vulnerabilities, Li and Paxson [16] used the NVD to perform a large-scale empirical study of security patches. They studied more than 4,000 bug fixes for over 3,000 vulnerabilities that affected a diverse set of 682 open-source software projects. Besides that, they characterized the security fixes in comparison to other non-security bug fixes, exploring the complexity of different types of patches and they impact on code bases. Different from them, besides considering variability, we also considered some C code metrics in our analysis. We conducted a survey with developers to better understand how they detect weaknesses.

Abal et al. [2] proposed a qualitative study of variability bugs of the Linux Kernel, Apache, Marlin, and BusyBox that provides insights about the nature, and number of occurrences. They investigated in what ways variability affects and increases the complexity of software bugs. They studied 25 kinds of bugs. In total, 19 of them are related to CWEs, such as Memory Leak and Null Pointer. Different from us, Abal et al. [2] did not investigate Format String, Race Condition, Risky Cryptographic Algorithm, and Integer Underflow weaknesses. We also considered some information form bug trackers. In addition, we conducted a survey with developers of these systems. We identified that 52.8% of them care about weaknesses when performing a commit. However, almost 62% of them use GCC and Clang to detect weaknesses. which are compiler tools and do not detect weaknesses.

Medeiros et al. [18] interviewed 40 developers, and conducted a survey with 202 developers to understand why they still use C preprocessor directives despite the strong preprocessor criticism in academia. In our work, we conducted a survey with developers to verify their experience in detecting weaknesses in configurable systems. Moreover, we asked them the tools and strategies used to detect weaknesses. Our results show that developers do not use appropriate tools to detect weaknesses regarding *#ifdefs*. A number of them could not detect Integer and Buffer overflows in our questions.

Previous studies [4, 12, 15, 23] have proposed strategies to deal with configuration-related faults. Braz et al. [4] proposed CheckConfigMX, a change-centric approach to compile configurable systems with *#ifdefs*. It performs a per-file impact analysis and identifies the configurations impacted by code changes. CheckConfigMX uses GCC to compile only the impacted configurations. They found 595 compilation errors of 20 kinds on the repository history of BusyBox, Apache HTTPD, and Expat. The reduced the effort of compiling this configurable systems by at least 50% (on average 99%) in terms of analyzed configurations – without considering feature models. Different from us, they investigated only compilation errors, and did not detect variability weaknesses. In our study, we investigated the analyzed weaknesses characteristics, such as their priority level.

Furthermore, we can extend CheckConfigMX to identify variability weaknesses using FlawFinder, for example.

Kästner et al. [14] proposed *TypeChef*, a variability-aware parser that uses abstract syntax trees to verify all possible configurations at once. However, TypeChef does not detect weaknesses on configurable systems with `#ifdefs`. In this work, we conducted a survey with developers of the analyzed configurable systems regarding the tools used to detect vulnerability weaknesses in configurable systems with *#ifdefs*. Most developers use GCC (38.8%) and Clang (22.8%) to perform this activity. However, there are better tools to detect them, such as CppCheck and Valgrind. In addition, 6.5% of them do not use any tool to detect vulnerability weaknesses in these systems.

Medeiros et al. [17] presented a study to understand the trade-off between effort and fault-detection capabilities comparing 10 sampling approaches regarding the size of the samples and their capability of fault detection. Their results show that simple algorithms with small sample sets, such as *most-enabled-disabled*, are the most efficient in most contexts. In this work, we analyzed characteristics of 27 vulnerability weaknesses reported to bug trackers. Similar to their findings, our results show that the *all macros enabled* sampling approach detects most analyzed weaknesses (70.37%), while the *one macro disabled* sampling approach detects 11.12% of them.

Garvin and Cohen [9] conducted an exploratory study on hundreds of faults from the open source systems GCC and Firefox, to revisit the notion of a feature interaction fault. Only 3 out of 28 faults are related to features interactions. We identified some feature interactions that yield some weaknesses.

6 CONCLUSIONS

We qualitatively analyzed 27 variability weaknesses of eight kinds. The most frequent weakness is Buffer Overflow (seven), followed by Null Pointer (six). The variability weaknesses involve up to two macros, and 51.85% of them occur inside *#ifdefs*. Moreover, most of them can be detected by the *all macros enabled* sampling approach. Developers took on median 15 days and 4 discussion messages to fix them.

Overall, our results show evidences that, although developers care about weaknesses, they cannot detect some weaknesses reported in the bug trackers. Some tools, such as CPPCheck and FlawFinder, may help developers to detect some weaknesses. However, most developers do not use proper tools to identify them. Moreover, developers do not report weaknesses relating to CWEs. It seems developers do not know them. CWE and CVE projects may help them improving their knowledge in detecting weaknesses. Finally, it is important to quickly fix some kinds of critical weaknesses, avoiding exposing the system to malicious attacks.

As future work, we intend to analyze more bug reports and kinds of weaknesses. In addition, we aim at analyzing more configurable systems in different domains and using different variability mechanisms. Furthermore, we intend to interview some developers to better understand how they detect weaknesses. Finally, we will investigate a subset of metrics that can predict weaknesses in configurable systems.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers. This work was partially supported by INES, funded by CNPq grants 308380/2016-9, 460883/2014-3, 465614/2014-0, 306610/2013-2, and 409335/2016-9, FAPEAL PPGs 14/2016, FAPEAL grants 60030 000435/2017 and 60030 1201/2016, CAPES grants 175956 and 117875, and DEVASSES.

REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th International Conference on Automated Software Engineering*. 421–432.
[2] I. Abal, J. Melo, S. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski. 2017. Variability bugs in highly-configurable systems: a qualitative analysis. *Transactions on Software Engineering and Methodology* (2017).
[3] C. Anley, J. Koziol, F. Linder, and G. Richarte. 2007. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* John Wiley & Sons, Inc.
[4] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira. 2016. A change-centric approach to compile configurable systems with *#ifdefs*. In *Proceedings of the 15th International Conference on Generative Programming: Concepts and Experiences*. 109–119.
[5] I. Chowdhury and M. Zulkernine. 2010. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?. In *Proceedings of the 25th Symposium on Applied Computing*. 1963–1969.
[6] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel. 2016. Do #ifdefs influence the occurrence of vulnerabilities? An empirical study of the Linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*. 65–73.
[7] M. Finifter, D. Akhawe, and D. Wagner. 2013. An empirical study of vulnerability rewards programs. In *Proceedings of the 22nd USENIX Conference on Security*. 273–288.
[8] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. 2010. *Modeling the security ecosystem - the dynamics of (In)security.* Springer US, 79–106.
[9] B. Garvin and M. Cohen. 2011. Feature interaction faults revisited: an exploratory study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering*. 90–99.
[10] M. Howard, D. LeBlanc, and J. Viega. 2010. *24 deadly sins of software security: programming flaws and how to fix them.* McGraw-Hill, Inc.
[11] J. Hulse. 2012. Buffer overflows: anatomy of an exploit. (2012). https://nvd.nist.gov
[12] M. Johansen, O. Haugen, and F. Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference*. 46–55.
[13] C. Kästner, S. Apel, T. Thüm, and G. Saake. 2012. Type checking annotation-based product lines. *Transactions on Software Engineering and Methodology* 21, 3 (2012), 1–39.
[14] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 805–824.
[15] D. Kuhn, D. Wallace, and A. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.
[16] F. Li and V. Paxson. 2017. A large-scale empirical study of security patches. In *ACM Conference on Computer and Communications Security*. 2201–2215.
[17] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. 643–654.
[18] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. 2015. The love/hate relationship with the C preprocessor: an interview study. In *Proceedings of the 29th European Conference on Object-Oriented Programming*. 495–518.
[19] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca. 2017. Discipline matters: refactoring of preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering* (2017).
[20] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th Conference on Computer and Communications Security*. 529–540.
[21] C. Pfleeger and S. Pfleeger. 2002. *Security in computing.* Prentice Hall Professional Technical Reference.
[22] H. Spencer and G. Collyer. 1992. `#ifdef` considered harmful, or portability experience with C news. In *USENIX Summer*. USENIX Association, 185–197.
[23] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. 2011. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. 1–5.