A Comparison of 10 Sampling Algorithms for Configurable Systems

Flávio Medeiros
Fed. Univ. of Campina Grande
Paraíba, Brazil

Christian Kästner
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Márcio Ribeiro
Federal University of Alagoas
Maceió, Alagoas, Brazil

Rohit Gheyi
Fed. Univ. of Campina Grande
Paraíba, Brazil

Sven Apel
Universität Passau
Passau, Germany

ABSTRACT

Almost every software system provides configuration options to tailor the system to the target platform and application scenario. Often, this configurability renders the analysis of every individual system configuration infeasible. To address this problem, researchers have proposed a diverse set of sampling algorithms. We present a comparative study of 10 state-of-the-art sampling algorithms regarding their fault-detection capability and size of sample sets. The former is important to improve software quality and the latter to reduce the time of analysis. In a nutshell, we found that sampling algorithms with larger sample sets are able to detect higher numbers of faults, but simple algorithms with small sample sets, such as *most-enabled-disabled*, are the most efficient in most contexts. Furthermore, we observed that the limiting assumptions made in previous work influence the number of detected faults, the size of sample sets, and the ranking of algorithms. Finally, we have identified a number of technical challenges when trying to avoid the limiting assumptions, which questions the practicality of certain sampling algorithms.

1. INTRODUCTION

Many software systems can be configured to different hardware platforms, operating systems, and requirements [48]. However, the variability that is inherent to configurable systems challenges quality assurance [3, 22, 23, 30]. Developers need to consider multiple configurations when they execute tests or perform static analyses to find faults and vulnerabilities. As the configuration space often explodes exponentially with the number of configuration options, analyzing every individual system configuration becomes infeasible in real-world projects; for example, the *Linux Kernel* has more than 12 thousand compile-time configuration options. Configuration-related faults that occur only in a subset of all configurations are especially tricky to find [30]. As such, it is not surprising that many configuration-related faults have been found in highly-configurable systems, such as the *Linux Kernel*, *Gcc*, *BusyBox*, and *Apache* [1, 14, 20, 31, 32, 53].

Although researchers have proposed approaches to analyze complete configuration spaces in a sound fashion for some classes of defects [15, 20, 21, 53, 54], the vast majority of mature quality-assurance techniques consider only a single configuration at a time. For example, static-analysis tools operate typically on C code after the C preprocessor has resolved configuration options implemented through conditional compilation (e.g., using `#ifdef` directives). To reuse state-of-the-art tools, such as *gcc*, for detecting configuration-related faults, *sampling* is a viable alternative [18, 29, 37, 39, 50]. That is, instead of analyzing all configurations, one selects a *subset* of configurations to analyze individually. The effectiveness of sampling for detecting configuration-related faults depends significantly on how samples are selected, though.

Several sampling algorithms have been proposed in the literature, such as *t-wise* [18,25,29,39], *statement-coverage* [51], and *one-disabled* [1]. To select a suitable sampling algorithm, one needs to understand the tradeoffs, especially with regard to effort (i.e., how large are the sample sets) and fault-detection capabilities (i.e., how many faults can be found in the sampled configurations). Unfortunately, a comparison of sampling algorithms for finding configuration-related faults is not available. More importantly, many proposed sampling algorithms make severe assumptions that may not be realistic for practical applications and that are not always clearly communicated. For instance, they perform analyses per file instead of globally, and they ignore constraints among configuration options, header files, and build-system information [23, 25, 36, 46]. Applying sampling algorithms under different assumptions may introduce significant additional effort or reduce coverage, as we will discuss. A lack of understanding of the tradeoffs and assumptions of sampling algorithms can lead to both undetected faults, which decrease software quality, and time-consuming code analysis, which increases costs.

We conducted a comparative study to analyze 10 sampling algorithms in detail to fill that gap. We compared the selected sample sizes and the fault-detection capabilities of the sampling algorithms in a study of 135 known configuration-related faults in 24 open-source C systems, each configurable with conditional compilation. Specifically, we analyzed a set of sampling algorithms proposed in the research literature: 5 variations of *t-wise* [18,29,37,39]; *statement-coverage* [51]; *random*; *one-disabled* [1]; *one-enabled*; and *most-enabled-*

disabled. In summary, we analyzed 10 sampling algorithms and 35 combinations of algorithms in two studies. In the first study, we compared sample sizes and fault-detection capabilities of the different sampling algorithms and their combinations on a large set of open-source systems under favorable assumptions (e.g., ignoring constraints and header files). In the second study, we explored the influence of considering constraints, header files, build-system information, and global analysis, which are often neglected in the literature and practice [23, 25, 36, 46].

Our results show that all algorithms select configurations with more than 66% of the configuration-related faults in our corpus. Almost 84% of all faults are detected by enabling or disabling one or two configuration options, but there are also faults that require developers to enable or disable up to *seven* options. As expected, we found that the algorithms with the largest sample sizes detected the most faults. However, simple algorithms with small sample sets, such as *most-enabled-disabled*, are the most efficient in many scenarios. More interestingly, we identified several novel combinations of algorithms that provide a useful balance between sample size and fault-detection capabilities.

As a further result, we found that considering constraints among configuration options, global analysis, header files, and build-system information influence the performance of most sampling algorithms substantially, up to the point that several algorithms are no longer feasible in practice. Considering constraints increases the time of analysis significantly, which prohibits us to use some algorithms, such as *three-wise* and *four-wise*, at all. Including build-system information increases the size of sample sets slightly, whereas global analysis and analyses that include configuration options from header files turn the analysis to be practically infeasible for most algorithms.

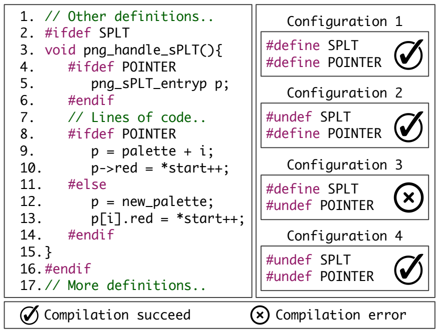
In summary, our main contributions are:

- A comparative study of 10 sampling algorithms and 35 combinations of algorithms regarding their fault-detection capability and size of sample sets;
- A study on the influence of considering header files, constraints, build-system information, and global analysis on the performance of sampling algorithms;
- A discussion of results showing that some sampling algorithms become infeasible under realistic settings, for example, when incorporating header files and applying global analysis;
- A report of significant changes of the efficiency ranking of sampling algorithms when considering different pieces of information, such as build-system and constraint information;
- Results supporting sampling algorithms with an efficient balance between sample size and fault-detection capabilities under different assumptions, such as the *most-enabled-disabled* algorithm.

All data used in this study are available on our *Website*.[1]

2. CONFIGURATION-RELATED FAULTS

Conditional compilation is used in many real-world systems to make the source code configurable [26]. For instance, Figure 1 depicts a code snippet of *Libpng*[2] related to splitting images into segments. The splitting feature is optional and is included only when configuration option SPLT is enabled. This code snippet also contains a configuration option that checks for pointer-index support, controlled by configuration option POINTER. Using the C preprocessor, we can generate four different configurations from this code snippet: (1) both configuration options enabled; (2) only POINTER enabled; (3) only SPLT enabled; and (4) both configuration options disabled.



```
1. // Other definitions..
2. #ifdef SPLT
3. void png_handle_sPLT(){
4.    #ifdef POINTER
5.       png_sPLT_entryp p;
6.    #endif
7.    // Lines of code..
8.    #ifdef POINTER
9.       p = palette + i;
10.      p->red = *start++;
11.   #else
12.      p = new_palette;
13.      p[i].red = *start++;
14.   #endif
15.}
16.#endif
17.// More definitions..
```

Configuration 1

```
#define SPLT
#define POINTER
```

✓

Configuration 2

```
#undef SPLT
#define POINTER
```

✓

Configuration 3

```
#define SPLT
#undef POINTER
```

⊗

Configuration 4

```
#undef SPLT
#undef POINTER
```

✓

Figure 1: A fault in *Libpng* that occurs when SPLT is enabled and POINTER is disabled.

Most analysis tools for C code, such as *gcc*, operate on preprocessed code, one configuration at a time. By compiling the code snippet of Figure 1 with SPLT enabled and POINTER disabled, we get a compilation error at Line 12. This line uses variable p, which is not declared before (Line 5) when we disable POINTER. Because common analysis tools check only one configuration at a time, they do not show warning or error messages when one compiles the code depicted in Figure 1 considering other configurations. This is an example of a configuration-related fault that can only be exposed in some combinations of configuration options [14, 23, 56]. Unfortunately, the space of possible combinations is exponential, in the worst case, and it is usually too large to explore exhaustively. For instance, the *Linux Kernel* offers more than 12K configuration options, which give rise to more configurations than there are atoms in the universe.

To analyze real-world configurable systems, developers often use sampling algorithms that select only a few configurations for analysis. For instance, one can check the code snippet presented in Figure 1 using the *most-enabled-disabled* sampling algorithm. It considers two configurations: (1) all configuration options enabled and (2) all options disabled. However, it is not possible to detect the fault presented in Figure 1 using the *most-enabled-disabled* algorithm, as the fault requires enabling one configuration option while disabling another. By using other sampling algorithms, one can detect this specific fault in *Libpng*, but other faults possibly not. For instance, one can use *one-disabled* [1], which disables one configuration option at a time, or *statement-coverage* [51], which enables each block of optional code at least once.

Previous work [1, 12, 14] has studied configuration-related faults similar to the one we discuss here and proposed many sampling algorithms [1,37,39,51]. However, researchers make assumptions that may not be realistic in practice. For instance, they perform per-file instead of global analysis, and they ignore constraints between configuration options, header files and build-system information. In this paper, we report on a comparative study of sampling algorithms initially accepting those assumptions (Section 4), but explicitly evaluate the influence of including different types of information in a second study (Section 5).

[1] http://www.dsc.ufcg.edu.br/~spg/sampling/

[2] http://www.libpng.org/

3. STUDY DESIGN AND SAMPLING ALGORITHMS

Our overall goal is to compare state-of-the-art sampling algorithms regarding their capability to detect configuration-related faults and the size of their sample sets. Furthermore, we study four assumptions of previous work, which often does not consider (1) constraints, (2) global analysis, (3) build-system information, and (4) header files. We perform our studies in the context of the C programming language and configuration options implemented with the C preprocessor (i.e., `#ifdef`), as illustrated in the previous section.

We aim at answering the following research questions:

- **RQ1.** What is the number of configuration-related faults detected by each sampling algorithm?
- **RQ2.** What is the size of the sample set selected by each sampling algorithm?
- **RQ3.** Which combinations of sampling algorithms maximize the number of faults detected and minimize the number of configurations selected?
- **RQ4.** What is the influence of the four assumptions on the feasibility to perform the analysis for each sampling algorithm?
- **RQ5.** What is the influence of the four assumptions on the number of faults detected by each sampling algorithm?
- **RQ6.** What is the influence of the four assumptions on the size of the sample set selected by each sampling algorithm?

3.1 Overall Study Design

At first glance, a study comparing sampling algorithms (RQ1–3) seems straightforward. We use a number of different sampling algorithms (independent variable) to measure how many of the faults we can find with them in different software systems and how big the sample set is (dependent variables). However, there are several challenges to overcome in the design of such an experiment.

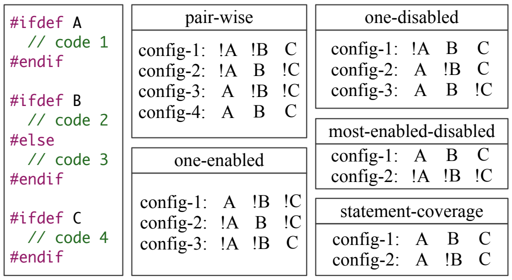
Sampling the configuration space needs to be combined with a technique to detect faults in the respective selected configurations, such as inspection (unrealistically laborious), executing existing test suites (if available), automated test-case generation (looking for crashing defects), or static analysis (prone to false positives). If not conduced carefully, we might be evaluating the fault-detection technique instead of the sampling algorithm. We address this potential bias by taking the fault-detection technique out of the loop and by using a corpus of previously found configuration-related faults. For each known fault, we check whether the sampling algorithms select configurations in which the fault could have been found, assuming a suitable fault-detection technique. That is, by using a corpus of confirmed configuration-related faults, we eliminate the fault-detection technique as a confounding factor from our study setup. However, we actually do not know if the sampling algorithms actually discovered more or different faults. We discuss this threat and an alternative study design in Section 6.

A second design challenge is how to evaluate the influence of the four assumptions (regarding global analysis, header files, constraints, and build-system information) behind many sampling algorithms. As we will show, lifting these assumptions can make it infeasible to apply some of the algorithms to real-world software systems. Therefore, we decided to proceed in two steps: First, we study tradeoffs among sampling algorithms (RQ1-3) under favorable conditions (i.e., fulfilling all assumptions). Subsequently, we investigate the influence of the assumptions (RQ4-6) on a smaller set of subject systems in a second study. The four assumptions are:

- **Constraints:** Constraints between configuration options may exclude certain configurations (e.g., option X may only be selected if option Y is selected) from the set of valid configurations. A sample set may contain configurations that violate constraints. Unfortunately, configuration constraints are rarely documented explicitly—the *Linux Kernel* is an exception and has been studied therefore extensively [38, 52]. In the presence of constraints, sample sets are often larger to achieve the same coverage, and highly optimized covering array tables[3] cannot be used. As we do not know configuration constraints for most of our subject systems, we exclude contraints entirely from the sampling process in our first study.
- **Global analysis:** We can sample configurations per file or globally for the entire system. Even in systems with many configuration options, individual files are usually affected only by few options. Sampling over the global configuration space may detect inter-file faults (e.g., linker issues), but this often creates huge sample sets, which hardly affect individual files. Thus, in the first study, we assume a per-file analysis.
- **Header files:** In C code, a significant amount of variability arises from header files. However, detecting all configuration options from header files in a sound way is a difficult and expensive task, which requires some form of variability-aware analysis [2, 10, 20, 54]. It is necessary to resolve includes and macro expansions, but to keep the conditional directives (i.e., partial preprocessing). We therefore analyze only configuration options inside source files in our first study.
- **Build system:** The build system may induce a significant amount of variability, such that certain files are not compiled in all configurations [10, 38]. Since build systems are inherently difficult to analyze [35], we do not use build-system information in the first study.

3.2 Sampling Algorithms

In both studies, we will analyze the same set of 10 sampling algorithms, proposed in prior work [1, 18, 28, 29, 37, 39, 51] as well as their combinations. We explain each sampling algorithm using the example code snippet of Figure 2.



```
#ifdef A
  // code 1
#endif

#ifdef B
  // code 2
#else
  // code 3
#endif

#ifdef C
  // code 4
#endif
```

pair-wise			
config-1:	!A	!B	C
config-2:	!A	B	!C
config-3:	A	!B	!C
config-4:	A	B	C

one-enabled			
config-1:	A	!B	!C
config-2:	!A	B	!C
config-3:	!A	!B	C

one-disabled			
config-1:	!A	B	C
config-2:	A	!B	C
config-3:	A	B	!C

most-enabled-disabled			
config-1:	A	B	C
config-2:	!A	!B	!C

statement-coverage			
config-1:	A	B	C
config-2:	A	!B	C

Figure 2: Comparing the sampling algorithms by example.

[3] A covering array is a mathematical object used for software testing, which ensures specific coverage criteria. For example, a *pair-wise* covering array ensures that all pairs of configuration options are considered by the array [18, 56].

The **t-wise** algorithm covers all combinations of t configuration options: *pair-wise* checks all pairs of configuration options ($t = 2$) [18, 29, 37, 39], and it selects four configurations of the example of Figure 2. Considering options A and B, we can see that there is a configuration where both options are disabled (*config-1*), two other configurations with only one of them enabled (*config-2* and *config-3*), and another configuration where both configuration options are enabled (*config-4*). The same situation occurs for configuration options A and C and options B and C. However, t can take integer values to check different combinations of options, such as *three-wise* ($t = 3$), *four-wise* ($t = 4$), and *five-wise* ($t = 5$). As we increase t, the sizes of the sample sets also increase. Figure 3 presents the sample-set distributions of *three-wise*, *four-wise*, *five-wise*, and *six-wise*. As we can see, *three-wise* and *four-wise* create small sample sets; *five-wise* and *six-wise* create much larger sample sets. We selected samples based on precomputed and optimal covering-array tables[4] that select a minimal set of configurations that covers all t combinations of configuration options. These tables do not consider constraints between configuration options. There are tools that implement *t-wise* considering constraints, such as *SPLCATool* [18], *CASA* [13], and *ACTS* [6]. However, these tools do not necessarily select a minimal sample set or even guarantee *t-wise* coverage, as discussed in Section 5.

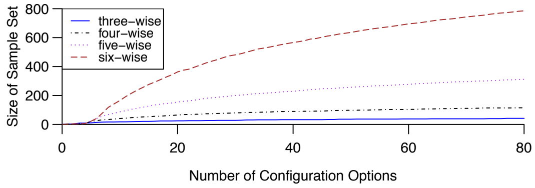


Figure 3: Sample sets of *t-wise* sampling considering a file with a number of configuration options ranging from zero to eighty.

The **statement-coverage** algorithm selects a set of configurations in which each block of optional code is enabled at least once [50]. We used *statement-coverage* as implemented in the *Undertaker* [51] tool suite.[5] Notice that we are not using *Undertaker* to detect dead code [51], but to select configurations with the *statement-coverage* algorithm. As presented in Figure 2, by enabling configuration options A, B, and C, the algorithm ensures that the optional code blocks `code 1`, `code 2` and `code 4` are enabled at least once. However, it needs another configuration (e.g., A and C enabled, and B disabled) to enable `code 3`. Including each block of optional code at least once does not guarantee that all possible combinations of individual blocks of optional code are considered, though.

The **most-enabled-disabled** algorithm checks two samples independently of the number of configuration options. When there are no constraints among configuration options, it enables all options (*config-1*), and then it disables all configuration options (*config-2*). **One-disabled** is an algorithm suggested by Abal et al. [1] based on 42 faults found in the *Linux Kernel*. It disables one configuration option at a time. We can also see in Figure 2 that it disables configuration option A in *config-1*, option B in *config-2*, and option C in *config-3*. In contrast, **one-enabled** enables one configuration option at a time.

Finally, we implemented a *random* sampling algorithm. Random sampling receives as input the maximum number of configurations (n) to check per file. Then, it creates n distinct configurations with all configuration options of the file and randomly assigns `true` or `false` for every option of each configuration. For files which a *brute-force* algorithm requires fewer configurations than the maximum number of configurations (n) per file, random sampling selects all configurations. We ran random sampling considering different numbers of configurations per file, ranging from 1 to 40. For each number, we ran the analysis ten times and computed the average number of detected faults and the 95% confidence interval.

4. DETECTING FAULTS

In this first study, we compared the fault-detection capabilities and the sample sizes of the 10 sampling algorithms using a corpus of 135 known faults of 24 open-source systems to answer questions RQ1–3. As explained in Section 3, we performed the first study under favorable assumptions, that is, without constraints, global analysis, build-system information, and header files.

We proceeded in three steps, as illustrated in Figure 4. In *Step 1*, we select each source file of the given subject system. *Step 2* applies each sampling algorithm to select the samples for every file. *Step 3* determines the number of configuration-related faults detected (RQ1) and measures the size of the sample set (RQ2) for each algorithm. The size of the sample set is the sum of the numbers of sampled configurations for every source file. To identify the sampling algorithms that detect a fault, we consider its presence condition, which is a subset of system configurations in which the fault can be found [41], assuming a suitable fault-detection technique. We checked whether we could find at least one configuration of this subset in the sampled configurations for each algorithm. Finally, we repeat the process for combinations of sampling algorithms (RQ3).

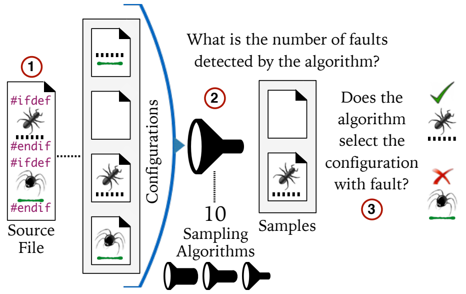

1
#ifdef
#endif
#ifdef
#endif
Source File
Configurations
What is the number of faults detected by the algorithm?
2
10 Sampling Algorithms
Samples
Does the algorithm select the configuration with fault?
3

Figure 4: Strategy used to compare the sampling algorithms.

4.1 Corpus of Faults

Using a corpus of configuration-related faults in a study raises the question of how to acquire a proper corpus and whether it is a representative corpus of configuration-related faults in real systems. Faults identified with existing sampling algorithms will obviously bias results toward these specific algorithms. Instead, we assembled a corpus of faults in which all faults have been identified in one of two ways:

- Variability-aware analysis tools are able to identify certain kinds of faults (mostly syntax and type errors) by

[4] The precomputed and optimal covering arrays used in our study are available at `http://math.nist.gov/coveringarrays/`.

[5] Despite the existence of an algorithm to compute an optimal solution for the coverage problem [28], which is *NP-hard*, we used an algorithm that computes the sample set much faster, but may produce a sample set that is possibly larger than optimal.

covering the entire configuration space without sampling. Difficulties in setting up these tools and narrow classes of detectable faults limit their applicability at this point, and their prototype status leads to false positives. We collected only configuration-related faults that have been reported by such tools, reported to the original developers, and confirmed or fixed by the developers [20, 31].

- We use configuration-related faults that have been manually identified and fixed by developers. Faults reported by users and fixed in the repository by the system's developers may be slightly biased toward more popular configurations, but are not systematically biased toward specific sampling algorithms. They represent configuration-related faults that are routinely detected and fixed in real software systems. We started with Abal's corpus of faults of the *Linux Kernel* [1], and complemented it with faults found in other studies [14,42], and our own investigation of software repositories (see Table 1).

Overall, the corpus of faults used in our study includes 135 configuration-related faults from 24 subject systems of various sizes and from different domains, over 125 different files with distinct numbers of configuration options (see Figure 5). Our corpus contains faults of different kinds, including syntax errors (34%), memory leaks (22%), null-pointer dereferences (17%), uninitialized variables (13%), undeclared variables and functions (5%), resource leaks (3%), array and buffer overflows (3%), arithmetic faults (2%), and type errors (1%). Table 2 presents a characterization of the subject systems we use in the first study, listing the project name, application domain, lines of code, number of files, number of configuration options, and number of known faults considered in our study.

Table 3 shows the presence conditions of the faults and the number of configuration options that we need to enable or disable to detect the configuration-related faults we consider in the first study: for 78 faults (58%), we need to enable some options; for 27 faults (20%), we need to disable some configuration options; and for another 30 faults (22%), we need to enable some options and disable others. The majority of faults (83%) are related to one or two configuration options, while less than 5% related to more than four configuration options.

Notice that we discarded seven faults of the *Linux Kernel* from our corpus that span multiple files, because we performed a per-file analysis in our first study. We considered faults that require inter-procedural analysis, as long as all procedures are defined in the same file.

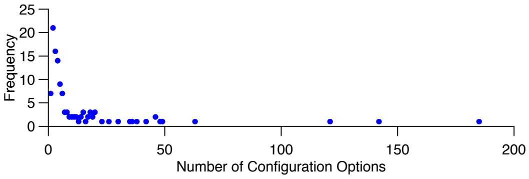


Figure 5: Number of distinct configuration options in files with configuration-related faults.

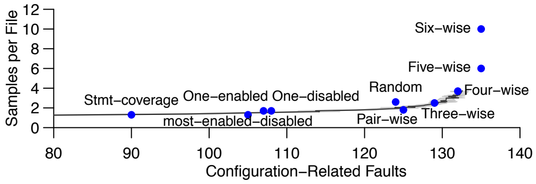
4.2 Results and Discussion

For each sampling algorithm, we answered the research questions RQ1–2. Figure 6 presents the number of faults detected and the corresponding size of the sample set for each algorithm. Note that detecting more faults does not mean more efficiency, because there is a tradeoff between the number of faults detected and the size of the sample set. We consider the efficiency of the sampling algorithms in terms of *Efficiency*: E = *SizeOfSampleSet* / *NumberOfFaults*. This ratio represents the number of configurations that one needs to check per fault to be detected. Furthermore, we analyzed 35 combinations of sampling algorithms to answer research question RQ3, as illustrated in Figure 7. We discuss the results in terms of the three research questions next.

RQ1. *What is the number of configuration-related faults detected by each sampling algorithm?*

Overall, we found that all algorithms detected more than 66% of all faults of our corpus. *Statement-coverage* detected the lowest number of faults, while *six-wise* detected the highest number. The majority of faults in our corpus can be detected by enabling or disabling fewer than six configuration options. This way, *six-wise* is able to detect all these faults. *Statement-coverage* missed 45 faults because they require developers to enable some configuration options and disable others (i.e., require specific combinations of multiple blocks of codes), whereas *statement-coverage* is only concerned with including each block of code at least once in a system configuration.

All *t-wise* sampling algorithms detected more than 92% of the 135 configuration-related faults. *Six-wise* and *five-wise* detected all faults. *Most-enabled-disabled*, *one-enabled*, and *one-disabled* detected all between 78% to 80% of the faults. Furthermore, we present the average values of random sampling with a 95% confidence interval (gray area) in Figure 6. We ran random sampling with the maximum number of configurations per file (n) ranging from 1 to 40, ten times for each value of n.[6] We report the mean of all runs; it detected 124 (92%) configuration-related faults.



Sampling Algorithm	Faults	Samples/File
Statement-coverage	90	1.3
Most-enabled-disabled	105	1.3
One-enabled	107	1.7
One-disabled	108	1.7
Random	124	2.6
Pair-wise	125	1.8
Three-wise	129	2.5
Four-wise	132	3.7
Five-wise	135	6.0
Six-wise	135	10.0

Figure 6: Number of configuration-related faults and samples per file for each sampling algorithm.

RQ2. *What is the size of the sample set selected by each sampling algorithm?*

The sizes of the sample sets range from 1.3 to 10 configurations per file. The algorithm *most-enabled-disabled* selected the smallest sample set; *six-wise* required the largest sample set (with more than 500K sampled configurations across all projects). The number of configurations to check influences the time of analysis. So, it is not feasible to use algorithms with large sample sets in all situations, as we will discuss

[6] Random selects 2.6 samples per file, on average.

Table 1: Configuration-related faults considered in our first study.

Source	Faults	Kind	Strategy	Subject system (number of faults)
[1]	30	Memory, type, and arithmetic	Repository mining	*Linux* (30)
[20]	10	Syntax	*TypeChef*	*BusyBox* (10)
[14]	5	Include, and arithmetic	Repository mining	*Gcc* (3), *Firefox* (2)
[42]	3	Type	Repository mining	*Gnome-keyring* (1), *Gnome-vfs* (1), and *Totem* (1)
[31]	22	Syntax	*TypeChef*	*Apache* (3), *Bash* (2), *Dia* (2), *Gnuplot* (5), *Libpng* (3), and *Libssh* (7)
-	65	Memory, type, and arithmetic	Our repository mining	*Apache* (9), *Bison* (2), *Cherokee* (3), *Cvs* (1), *Dia* (1), *Fvwm* (10), *Gnuplot* (5), *Irssi* (4), *Libpng* (1), *Lua* (1), *Libssh* (10), *Linux* (7), *Libxml* (2), *Lighttpd* (1), *Vim* (5), *Xfig* (1), and *Xterm* (2)
Total	135	70 faults collected from previous studies and 65 detected in our additional repository analysis.		

in Section 7. Based on our efficiency measure, we rank the algorithms starting from the most efficient: *most-enabled-disabled*, *pair-wise*, *stmt-coverage*, *one-disabled*, *one-enabled*, *three-wise*, *random*, *four-wise*, *five-wise*, and *six-wise*.

RQ3. *Which combinations of sampling algorithms maximize the number of faults detected and minimize the number of configurations selected?*

In addition to the individual algorithms, we analyzed combinations (that is, the union of the sample sets produced by the respective sampling algorithms) of two and three sampling algorithms, excluding *random*, *five-wise*, and *six-wise* algorithms. We excluded *random* because it detects different numbers of faults in different runs, and we excluded *five-wise* and *six-wise* because they already detected all 135 faults. Furthermore, we excluded combinations with more than three algorithms, because they resulted in inefficient combinations according to our efficiency function.

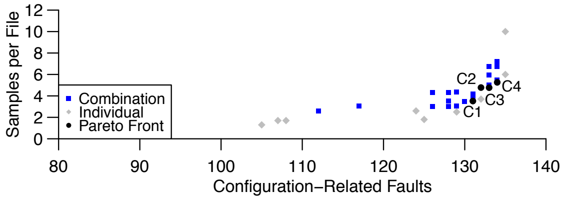
Figure 7 relates the number of faults and the size of sample sets for all combinations of sampling algorithms. Based on the results, we determined the *Pareto Front* to illustrate tradeoffs between number of detected faults and size of the sample sets. Figure 7 also presents the combinations of sampling algorithms on the *Pareto Front*, starting from the most efficient: *C1*, *C3*, *C2*, and *C4*.

Summary

All sampling algorithms are able to detect at least 66% of the configuration-related faults; most-enabled-disabled, pair-wise, and statement-coverage are the most efficient algorithms; some combinations provide a useful balance between sample size and fault-detection capabilities.

5. EFFECTS OF ASSUMPTIONS

In the first study, we made many simplifying assumptions also made in related studies on sampling [23, 25, 46]. We ignored constraints, header files, and build-system information, and we did a per-file analysis only. In more realistic conditions, these assumptions often do not hold: For example, constraints often exist, and ignoring them may lead to false positives, but constraints are rarely documented systematically and therefore easily ignored. Similarly information from build systems may increase precision but build systems are inherently difficult to analyze [2, 10]. While the simplifying assumptions allow researchers and practitioners to apply sampling algorithms quickly to a large set of systems, as we did in our first study, their influence on practicability and effectiveness is not well understood. Therefore, in a second study, we explore the effect of each assumption on the efficiency of the sampling algorithms.



Sampling Algorithm
C1 Pair-wise and one-disabled
C2 One-enabled, one-disabled, and statement-coverage
C3 One-enabled, one-disabled, and most-enabled-disabled
C4 One-enabled, one-disabled, and pair-wise

	Faults	Samples/File		Faults	Samples/File
C1	131	3.5	C2	132	4.8
C3	133	4.8	C4	134	5.3

Figure 7: Number of configuration-related faults and samples per file for the combination of algorithms on the *Pareto Front*.

Basically, we replicate the first study for a subset of the corpus, investigating how the assumptions affect each sampling algorithm (**RQ4–6**). To increase internal validity [47], we considered each assumption separately as an independent variable that we manipulate to understand the influence of each assumption on sampling. We limit the second study to faults of the *Linux Kernel* and *BusyBox* (47 faults from the first study), because these subject systems are the only ones for which we have build-system and constraint information from the *LVAT* and *TypeChef* projects [5, 34, 45]. For the *Linux Kernel*, we consider additionally seven known faults that cross files, which we excluded from our original corpus, as we discussed in Section 4.

Table 4 summarizes the number of configuration-related faults detected, sizes of sample sets, and the ranking of sampling algorithms per lifted assumption.

5.1 Constraints

Constraints exclude certain combinations of configuration options (e.g., option X must be selected if option Y is selected) from the set of valid configurations. Faults identified in invalid configurations are considered false positives (which did not occur in the first study, because we consider only a corpus of true positives); hence sampling invalid configurations adds no value. The analyzed version of the *Linux Kernel* has 293,826 constraint clauses among its configuration options; *BusyBox* has 615.

In the original sample sets of the first study, many sampled configurations are actually invalid in these highly constrained configuration spaces. For instance, *random* selects 24% of valid configurations and the percentage goes up to 43% when picking *most-enabled-disabled*. Sampling within

Table 2: Project characterization and the total number of known faults used in the first study.

Project	**Application domain**	**LOC**	**Files**	**Configuration options**	**Faults**
Apache	*Web* server	144,768	362	700	12
Bash	language interpreter	44,824	138	1,427	2
Bison	parser generator	24,325	129	269	2
Busybox	UNIX utilities	189,722	805	1,418	10
Cherokee	*Web* server	63,109	346	452	3
Cvs	version control system	76,125	236	628	1
Dia	diagramming software	28,074	132	307	3
Firefox	*Web* browser	6,017,673	22,423	17,415	2
Fvwm	windows manager	102,301	270	301	10
Gcc	C/C++ compiler	1,946,622	22,034	3,825	3
Gnome-keyring	daemon application	76,525	376	213	1
Gnome-vfs	file system library	78,380	286	427	1
Gnuplot	plotting tool	79,557	152	500	10
Irssi	IRC client	51,356	308	157	4
Libpng	PNG library	44,828	61	327	4
Libssh	SSH library	28,015	125	115	17
Libxml	XML library	234,934	162	2,126	2
Lighttpd	*Web* server	38,847	132	215	1
Linux	operating system	12,594,584	37,520	26,427	37
Lua	language interpreter	14,503	59	145	1
Totem	movie player	31,596	135	84	1
Vim	text editor	288,654	178	942	5
Xfig	vector graphics editor	70,493	192	143	1
Xterm	terminal emulator	50,830	58	501	2
Total					**135**

such constrained spaces is more challenging for all sampling algorithms, as solvers or search-based strategies are needed. We incorporate constraints as follows:

- *Most-enabled-disabled:* We cannot simply enable all options if some of them are mutually exclusive. Instead, we use a solver to find two valid configurations with the maximum number of configuration options enabled and disabled. If there are multiple optimal solutions, we pick the first offered by the solver.
- *One-enabled/disabled:* Similarly, for each option, we use a solver to identify the valid configuration that disables/enables the most other options.
- *Random sampling:* We randomly assigned *true* or *false* for every configuration option inside a file and discard invalid assignments until we find the desired number of configurations. Truly random sampling in large constrained spaces with many options is still a research problem though, with recent progress in theory [7] and recent pragmatic search heuristics [17].
- *Statement-coverage:* To select a minimal set of covering configurations, we need to consider constraints. Conceptually we can use the original implementation of *statement-coverage*, as part of *Undertaker* [51], but the tool is not flexible to handle other projects than *Linux*. Thus, we use an alternative implementation that we created in previous work [28].
- *T-wise sampling:* The covering array tables used in the first study are precomputed, often optimal solutions that, however, assume independence of all options. Recent research investigated strategies to generate *t-wise* covering arrays for constrained configuration spaces, such as *SPLCATool* [18], *CASA* [13], and *ACTS* [6]. All tools use heuristics and may produce larger-than-optimal sampling sets and sampling sets that do not actually achieve full t-wise coverage. To generate the *pair-wise* covering array, we used *SPLCATool*. We failed to generate *three-wise* or even higher covering arrays for the *Linux Kernel*: Even with 120 Gb RAM we ran out of memory; a developer from *CASA* estimated that that the generation could take months and would require a 1.6 Tb array to track the covered options. Overall, we could not find an alternative to implement the *three-wise*, *four-wise*, *five-wise* and *six-wise* algorithms considering constraints; existing approaches are intractable for the size and complexity of the *Linux Kernel*.

The changes in sampling algorithms to incorporate constraints changed the efficiency of the algorithms as summarized in Table 4. Most affected were t-wise strategies: *Pair-wise* required a larger sample set and detected fewer faults (including faults that *pair-wise* should have guarantee to find) from the *Linux Kernel*, because the used heuristics are unsound and do not cover all valid pairs of options. *Three-wise* sampling and beyond was not tractable at all.

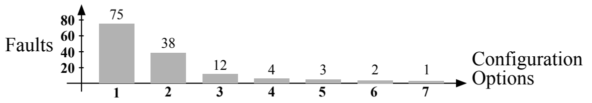
The time to compute sample sets increases significantly when adding constraints. Our use of a SAT solver required significant additional time and memory to generate the sample sets. On average, we created sample sets for each file in 0.04 seconds without constraints (the first study), while the analysis with constraints took 0.75 seconds per file, on average. This time represents an increase from 15 minutes to over 4 hours for the *Linux Kernel*. Regarding the ranking of algorithms, *most-enabled-disabled* and *statement-coverage* remain at top positions (see Table 4); the *t-wise* algorithms dropped significantly or were not feasible at all.

Summary

When considering constraints, we substantially reduce false positives; but there are high costs for generating sample sets, which are often not optimal.

Table 3: Presence conditions of the configuration-related faults.

Some configuration options enabled	78 (58%)
a	59
$a \wedge b$	13
$a \wedge b \wedge c$	5
$a \wedge b \wedge c \wedge d \wedge e$	1
Some configuration options disabled	27 (20%)
$!a$	16
$!a \wedge !b$	8
$!a \wedge !b \wedge !c$	1
$!a \wedge !b \wedge !c \wedge !d$	1
$!a \wedge !b \wedge !c \wedge !d \wedge !e \wedge !f \wedge !g$	1
Some options enabled and some disabled	30 (22%)
$(!a \wedge b) \vee (a \wedge !b)$	17
$(a \wedge b \wedge !c) \vee (!a \wedge !b \wedge c)$	6
$(a \wedge b \wedge !c \wedge !d) \vee (a \wedge b \wedge c \wedge !d)$	3
$(a \wedge b \wedge c \wedge d \wedge !e) \vee (!a \wedge !b \wedge !c \wedge !d \wedge e)$	2
$a \wedge !b \wedge !c \wedge !d \wedge !e \wedge !f$	1
$a \wedge b \wedge !c \wedge !d \wedge !e \wedge !f$	1



5.2 Global Analysis

To perform global analysis, we created a single sample set across all files, instead of a distinct set per file. Such global set allows us to perform cross-file analysis to find faults that cannot be identified on a per-file basis, such as linking problems. However, for global analysis, a sampling algorithm needs to consider all options in the system, not just the subset of options used in each file.

We were not able to generate global sample sets with any *t-wise* algorithm at the scale of our subject systems. The largest precomputed tables we found covered up to 2K options (pair-wise) or 191 options (six-wise). We are not aware of any tool that has the capability to generate covering arrays for such a large number of configuration options, even without constraints. *Statement-coverage* also turns intractable, as it requires to solve the coverage problem considering all source files of the project (i.e., equivalent to concatenating all source code into a single file and finding a set of configurations that enabled all optional code blocks at least once). *One-enabled* and *one-disabled* require substantially larger sample sets, as more options are considered (from 1.7 to almost 8K). *Random* requires larger sample sets, on average, because previously we could use smaller sample sets when the file had only few options. *Most-enabled-disabled* is the only algorithm for which the size of sample sets was not influenced, because it is not sensitive to the number of options and it always selects exactly two configurations.

To explore the ability of global analysis to identify cross-file faults, we opportunistically analyzed 7 known faults of the *Linux Kernel* [1] that span multiple files, which we had to exclude from our first study. We detected all seven faults by applying *one-enabled* and *one-disabled* with global analysis. *Most-enabled-disabled* detected five (71%) out of the seven faults, and *random* detected four (57%) faults. The other algorithms are not feasible with global analysis.

> Summary
>
> *Using a global analysis, we can potentially detect faults that span multiple files; it causes an explosion in the number of configuration options that leads to large sample sets, too large for t-wise and statement-coverage.*

5.3 Header Files

In C source code, variability may be introduced by header files, because macros used in `#ifdefs` can have non-local effect. If sampling is applied only to variability in the main C source file, faults stemming from variability in header files may not be detected. For example, a function may not be declared in all configurations of the header, a type name may be defined as either `int` or `long` depending on configuration decisions in the header, or a macro may be defined in the header only in some configurations. Precisely analyzing header variability is challenging, though, due to the interaction of `#include` directives with conditional compilation and macros. Precise analyses exist [15, 20], but are challenging and time-consuming to use, because one needs to set up the environment with all header files used by the project.

Incorporating header files increases the number of configuration options per file significantly. For instance, whereas the files of the *Linux Kernel* contain, on average, 3 distinct configuration options when ignoring variability from header files, headers add another 238 distinct configuration options, on average. This increases the size of the sample set for all algorithms, except for *most-enabled-disabled.* For *statement-coverage*, *five-wise*, and *six-wise*, our subject systems reach configuration spaces for which these algorithms become intractable.

Since our corpus does not include faults caused by misconfigurations from header files, most algorithms detect the same faults. The *one-enabled* algorithm detected more faults, because including configuration options from headers allowed it to disable more options, while enabling one at a time.

> Summary
>
> *When incorporating header files, there is a potential to detect additional faults from header files; but the setup is difficult and the sample sets are much larger (if feasible at all), which lead to ranking changes.*

5.4 Build-System Information

The build system controls which files are compiled and included. Files may be included only when specific configuration options are selected or may be compiled with additional parameters. This is equivalent to wrapping an additional `#ifdef` around each source file or define additional macros in the beginning of a file. Like ignoring constraints, ignoring build-system information can lead to false positives, where faults are reported in configurations that are prevented in practice by the build system.

Build systems often have a strong influence on the configurability of a system; for instance, in the *Linux Kernel*, 97% of source files are compiled only in certain configurations, 80% in *BusyBox.* Still, extracting configuration knowledge from build systems is very difficult. While *Linux* and *Busybox* have been analyzed with specialized parsers that recognize common patterns [5, 34], and more modern build systems use a more declarative style, which is easier to analyze [33], analyzing Make files in general is an open research problem with only few initial solutions [49, 57].

Table 4: Number of faults, size of sample sets and ranking considering the 47 faults of the second study.

Algorithms	**Constraints**			**Global analysis**			**Header files**			**Build system**		
	Faults	Configs	Rank	Faults	Configs	Rank	Faults	Configs	Rank	Faults	Configs	Rank
Pair-wise	33 ↓	30 ⇑	5	–	–	–	39 =	936 ⇑	4	33 ↓	2.8 ↑	4
Three-wise	–	–	–	–	–	–	43 =	1,218 ⇑	5	42 ↓	3.9 ↑	5
Four-wise	–	–	–	–	–	–	45 =	1,639 ⇑	7	45 =	5.7 ↑	8
Five-wise	–	–	–	–	–	–	–	–	–	47 =	8.3 ↑	9
Six-wise	–	–	–	–	–	–	–	–	–	47 =	12 ↑	10
Most-enabled-disabled	23 ↓	1.4 =	1	27 =	1.4 =	1	27 =	1.4 =	1	26 ↓	1.4 ↑	2
One-enabled	30 ↑	1.1 ↓	3	31 ↑	7,943 ⇑	3	31 ↑	890 ⇑	6	20 ↓	2.3 ↑	7
One-disabled	38 ↓	1.1 ↓	4	39 =	7,943 ⇑	2	39 =	890 ⇑	3	39 =	2.3 ↑	3
Random	39 ↓	4.1 =	6	29 ⇓	8,123 ⇑	4	40 ↓	17.2 ⇑	2	41 =	4.2 ↑	6
Stmt-coverage	32 ↑	4.1 ↑	2	–	–	–	–	–	–	25 =	1.3 ↑	1

Some algorithms do not scale, indicated using dashes (–). We use ↑ and ↓ to represent small changes in the number of faults and size of sample set, as compared to our first study and we use ⇑ and ⇓ to represent larger changes.

Considering build-system information, the presence conditions of faults become more complex, because we include the condition when the file is compiled: Whereas without build-system information 40 % of all faults of our corpus can be found by enabling or disabling a single option, only 17 % can be found the same way with build-system information. By requiring more options to pinpoint faults, incorporating build-system information decreases the efficiency of algorithms. *Pair-wise*, *three-wise*, *most-enabled-disabled*, and *one-enabled* detected fewer faults than in the first study.

The sizes of the sample sets are slightly increased in all sampling algorithms (except *most-enabled-disabled*), as we consider additional configuration options used in the build system. Time required to compute sample sets is increased only by a few milliseconds.

> Summary
>
> *Including build-system information requires to consider more configuration options in most files, but does not significantly affect any sampling algorithm or their efficiency.*

6. THREATS TO VALIDITY

Regarding external validity, we studied only systems that implement variability with conditional compilation and cannot generalize to systems that use other mechanisms to implement variability.

Regarding internal validity, the corpus of faults is critical for our research strategy. Creating a representative corpus is difficult, primarily because we have no means of knowing all faults in the system. We address this threat with two strategies:

- We avoided biasing our corpus to any specific sampling algorithm. As the corpus has been partially mined from software repositories, it might be biased towards more popular system configurations. Still, our corpus is the most comprehensive corpus of configuration-related faults we are aware of.
- We conducted a complimentary experiment using an automated bug-finding technique instead of a corpus of known faults. This experiment yielded comparable results and complements and confirms the first study on our corpus. In a nutshell, we measured with which sampling algorithm the bug-finding technique, static analysis with *Cppcheck*, would expose the most warnings per sampled configuration. The experiment introduces a different threat to internal validity in terms of false positives, but triangulating the results across both setups with orthogonal threats increases confidence in our findings. We omit details for space reasons and refer the interested reader to Appendix A.

7. GUIDANCE FOR PRACTITIONERS

Our study provides empirical evidence about the efficiency and typical sample sizes for analyzing configurable C code with various sampling algorithms both under ideal and real-world conditions. There is not a single sampling algorithm that is optimal for all systems and in all conditions, but practitioners can use our results to identify plausible candidates for their purposes.

For instance, during initial phases of a project, when developers are changing the source code frequently, they may prefer sampling algorithms with small sample sets to run the analysis fast. At some point, such as before a release, developers might want to use algorithms with larger sample sets, to minimize the number of configuration-related faults. Under favorable conditions, that is, without considering constraints, global analysis, header files, and build-system information, all algorithms scale in practice; we recommend *t-wise* sampling with a high *t* for rigorous testing and simpler sampling algorithms, such as *most-enabled-disabled*, *pair-wise*, *statement-coverage*, and combinations of these, for quicker early runs. Combining many and expensive sampling strategies does bring only marginal benefits but requires very large sample sets; we do not recommend them.

When considering header files, constraints, and global analysis, we recommend to go for simple algorithms, such as *most-enabled-disabled*, because many other algorithms become intractable in practice, as presented in Table 4.

Again, while no algorithm fits all contexts, we hope that our data will help practitioners to identify suitable candidate sampling algorithms for their specific scenario.

8. RELATED WORK

Several researchers have studied the way developers use the C preprocessor. They performed empirical studies with open-source systems written in C that are statically configurable with the C preprocessor [4, 11, 27]. Liebig et al. [27] found that almost 16% of the preprocessor usage is undisciplined according to an empirical study of 40 C software systems. In a previous study [30], we 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. According to our results, developers check only a few configurations of the source code. All these studies discussed the

C preprocessor and its problems, such as faults, inconsistencies, code quality, and incomplete testing. These studies motivated us to analyze sampling algorithms to support developers in finding configuration-related faults.

Other studies have analyzed software repositories by considering faults already fixed by developers to understand the characteristics of configuration-related faults [1,31]. In particular, researchers analyzed configuration-related faults in dynamic configurable systems [14,23]. They concluded that the majority of configuration-related faults involve a few configuration options, a result similar to ours. Abal et al. [1] analyzed the *Linux Kernel* software repository to study configuration-related faults. Tartler et al. [50] also performed studies to find configuration-related faults in the *Linux Kernel*. In our study, we considered some configuration-related faults reported by these previous studies. In addition, there are several studies proposing tools to find faults, such as *Undertaker* [51], *Tracker* [55], and *Splint* [24].

Researchers have proposed various strategies to deal with configuration-related faults. They considered combinatorial interaction testing to check different combinations of configuration options and prioritize test cases [8,9,23,40,43,56]. Nie et al. [36] performed a survey with combinatorial testing approaches, but without considering the complexities of C, such as header files and build-system information. Most studies on sampling make assumptions that might not be realistic in practice, such as ignoring constraints among configuration options. Including constraints, build-system information, and header files is a non-trivial task. Several researchers used the *t-wise* sampling algorithm to cover all t configuration option combinations [18,29,37,39], many studies without considering constraints [23,25,36,46]. Other researchers proposed the *statement-coverage* [51] sampling algorithm and applied a per-file analysis. Abal et al. [1] suggested the *one-disabled* algorithm. Sánchez et al. [44] studied realistic settings and studied the use of non-functional data for test case prioritization. Other researchers applied *t-wise* algorithms with constraints [6,13,18], and Grindal et al. [16] studied different constraint handling methods. However, a comparative study to understand the fault-detection capability and effort (size of sample set) of sampling algorithms, and the influence of limiting assumptions on sampling was not covered in previous studies.

Kästner et al. [20] developed a variability-aware parser that analyzes all possible configurations of a C program simultaneously. It also performs type checking [19] and dataflow analysis [28]. Gazzillo and Grimm [15] developed a similar parser. In our work, we considered faults detected by *TypeChef* and reported in previous studies [21]. Difficulties in setting up these tools and narrow classes of detectable faults limit their applicability and lead to false positives. In addition, variability-aware tools work at the preprocessor level, which hinders the reuse of existing bug checkers of traditional C tools, including *Gcc* and *Clang*.

Some studies have compared sample-based and variability-aware strategies. Apel et al. [3] developed a model checking tool for product lines and used it to compare sample-based and variability-aware strategies with regard to verification performance and the ability to find defects. Liebig et al. [28] performed studies to detect the strengths and weaknesses of variability-aware and sampling-based analyses. They considered two analysis implementations (type checking and liveness analysis) and applied them to a number of subject systems, such as *Busybox* and the *Linux Kernel*. In our study, we performed complimentary analyses regarding sampling algorithms and filled a gap by comparing sampling algorithms considering the influence of assumptions made in previous studies.

9. CONCLUDING REMARKS

We compared 10 sampling algorithms. Our study makes a step toward understanding the tradeoffs between effort (i.e., how large are the sample sets) and fault-detection capabilities (i.e., how many faults can be found in the selected configurations).

In a first study, we used a corpus of 135 configuration-related faults from 24 popular C projects reported in previous studies. We initially ran the comparison accepting some assumptions and we ignored configuration constraints, header files and build-system information, and we applied a per-file analysis. The results reveal that all sampling algorithms selected configurations that include at least 66% of the 135 faults reported in previous work. As expected, the algorithms with the largest sample sizes detected the most faults. More interestingly, we identified several combinations of algorithms that provide a useful balance between sample size and fault-detection capabilities.

Subsequently, we performed a complementary study to measure the influence of considering constraints, global analysis, header files, and build-system information on sampling. We found that, when considering constraints, we can reduce false positives, but it increases the costs for generating sample sets, which are often not optimal. Using a global analysis, we can potentially detect non-modular faults that span multiple files, but it causes an explosion in the number of considered configuration options that leads to large sample sets. When incorporating header files, there is a potential to detect additional faults, but the setup is difficult and the algorithms produce much larger sample sets. When including build-system information, we face a difficult analysis, a few more configuration options to consider, but no significant changes. Overall, we found that global analysis and analyses that include configuration options from header files turn the analysis to be practically infeasible for most algorithms.

Our study fills a gap, as a comparison of sampling algorithms for finding configuration-related faults was not available. Our findings are meant to support developers in understanding the tradeoffs regarding effort and fault-detection capabilities of sampling algorithms, aiding developers in selecting an algorithm and deciding what kind of information they include in the analysis. A lack of understanding the tradeoffs and assumptions of sampling algorithms can lead to both, undetected faults, which decreases software quality, and time-consuming code analysis, which increases costs.

Acknowledgments

This work has been supported by CNPq 460883/2014-3, 573964/2008-4 (INES), 477943/2013-6, and 306610/2013-2, CAPES 175956, project DEVASSES (European Union Seventh Framework Programme, agreement no PIRSES-GA-2013-612569), NSF award 1318808, the U.S. Department of Defense through the Systems Engineering Research Center (SERC, contract H98230-08-D-0171), the Science of Security Initiative, and the German Research Foundation (AP 206/4 and AP 206/6).

10. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 421–432. IEEE/ACM, 2014.

[2] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter. The evolution of the Linux build system. In *Proc. of the Int. Symposium on Software Evolution*, 2007.

[3] S. Apel, A. v. Rhein, P. Wendler, A. Grösslinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. of the Int. Conf. on Software Engineering*, pages 482–491. IEEE, 2013.

[4] I. Baxter. Design maintenance systems. *Communication of the ACM*, 35(4):73–89, 1992.

[5] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 73–82. ACM, 2010.

[6] M. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of acts: A case study. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation*, pages 591–600. IEEE, 2012.

[7] S. Chakraborty, D. Fremont, K. Meel, S. Seshia, and M. Vardi. On parallel scalable uniform SAT witness generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 304–319. Springer, 2015.

[8] D. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[9] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proc. of the Int. Conf. on Software Engineering*, pages 38–48. IEEE, 2003.

[10] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A robust approach for variability extraction from the Linux build system. In *Proc. of the Software Product-Line Conf.*, pages 21–30. ACM, 2012.

[11] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[12] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proc. of the Int. Conf. on Software Maintenance*, pages 379–388. IEEE, 2005.

[13] B. Garvin, M. Cohen, and M. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Proc. of the Int. Symposium on Search Based Software Engineering*, pages 13–22. IEEE, 2009.

[14] B. J. Garvin and M. B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceeding of the Int. Symposium on Software Reliability Engineering*, pages 90–99. IEEE, 2011.

[15] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proc. of the Programming Language Design and Implementation*, pages 323–334. ACM, 2012.

[16] M. Grindal, J. Offutt, and J. Mellin. Handling constraints in the input space when using combination strategies for software testing. Technical Report TR-06-001, University of Skövde, 2006.

[17] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proc. of the Int. Conf. on Software Engineering*, pages 517–528. ACM, 2015.

[18] M. F. Johansen, O. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proc. of the Int. Software Product Line Conf.*, pages 46–55. ACM, 2012.

[19] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39, 2012.

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

[21] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proc. of Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 773–792. ACM, 2012.

[22] P. Knauber and J. Bosch. Software variability management. In *Proc. of the Int. Conf. on Software Engineering*, pages 779–780. IEEE, 2003.

[23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[24] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. of the Conf. on USENIX Security Symposium.* USENIX Association, 2001.

[25] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[26] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. of the Int. Conf. on Software Engineering*, pages 105–114. ACM, 2010.

[27] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proc. of the Aspect-Oriented Software Development*, pages 191–202. ACM, 2011.

[28] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. of the Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[29] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *Proc. of the Int. Software Product Line Conf.*, pages 227–235, 2013.

[30] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C

preprocessor: An interview study. In *Proceedings of the European Conf. on Object-Oriented Programming*, pages 999–1022. ACM, 2015.

[31] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proc. of the Int. Conf. on Generative Programming: Concepts & Experiences*, pages 75–84. ACM, 2013.

[32] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proc. of the Int. Conf. on Generative Programming: Concepts & Experiences*, pages 35–44. ACM, 2015.

[33] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at Google. In *Proc. of the Int. Workshop on Managing Technical Debt*, 2012.

[34] S. Nadi, T. Berger, Kästner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering*, 41(8):820–841, 2015.

[35] S. Nadi and R. C. Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8), 2013.

[36] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011.

[37] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Lines: Going Beyond*, pages 196–210. Springer, 2010.

[38] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *Proc. of the Int. Software Product Line Conf.*, pages 91–100. ACM, 2013.

[39] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for product lines. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation*, pages 459–468. IEEE, 2010.

[40] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proc. of the Int. Symposium on Software Testing and Analysis*, pages 75–86. ACM, 2008.

[41] A. Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. Presence-condition simplification in highly configurable systems. In *Proc. of the Int. Conf. on Software Engineering*, pages 178–188. ACM, 2015.

[42] M. Ribeiro, P. Borba, and C. Kästner. Feature maintenance with emergent interfaces. In *Proc. of the Int. Conf. on Software Engineering*, pages 989–1000. ACM, 2014.

[43] S. Sampath, R. Bryce, G. Viswanath, V. Kandimalla, and A. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proc. of the Int. Conf. on Software Testing, Verification, and Validation*, pages 141–150. IEEE, 2008.

[44] A. B. Sánchez, S. Segura, J. A. Parejo, and A. Ruiz-Cortés. Variability testing in the wild: The drupal case study. *Software and Systems Modeling*, 14(52), 2015.

[45] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. In *Proc. of the Variability Modeling of Software-intensive Systems.* ACM, 2010.

[46] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Proc. of the Int. Conf. on Computational Science*, pages 1088–1091. Springer, 2005.

[47] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Int. Conf. on Software Engineering*, pages 9–19, 2015.

[48] H. Spencer and G. Collyer. Ifdef considered harmful, or portability experience with C news. In *Proc. of the USENIX Annual Technical Conf.* USENIX Association, 1992.

[49] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen. Build code analysis with symbolic evaluation. In *Proc. of the Int. Conf. on Software Engineering*, pages 650–660, 2012.

[50] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *USENIX Annual Technical Conf.* USENIX Association, 2014.

[51] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. In *Proc. of the Workshop on Programming Languages and Operating Systems.* ACM, 2011.

[52] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. of the Int. Conf. on Computer Systems.* ACM, 2011.

[53] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *Int. Journal on Software Tools for Technology Transfer*, 14(5), 2012.

[54] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1), 2014.

[55] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In *Proc. of the Int. Conf. on Software Engineering.* ACM, 2010.

[56] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1), 2006.

[57] L. Zhu, D. Xu, X. S. Xu, A. B. Tran, I. Weber, and L. Bass. Challenges in practicing high frequency releases in cloud environments. In *Proc. of the Int. Workshop on Release Engineering*, 2014.

APPENDIX

A. CPPCHECK WARNINGS

The goal of this experiment is to compare the sampling algorithms (**RQ1–3**) from a different perspective. Instead of measuring fault-detection capabilities in terms of a corpus of known configuration-related faults, we use a static-analysis tool as our automated fault-detection mechanism.

They key difference to *Study 1* is how we operationalize the dependent variable regarding fault-detection capabilities. Unfortunately, there is no tool that would produce a reliable ground truth.[7] We run the static analysis tool (*Cppcheck*) on each sampled configuration of each file and count all reported warnings. We discard warnings that occur in all configurations, because they are not configuration related. Although a warning does not necessarily correspond to a fault, it provides a rough estimate of the number of issues a developer would have to investigate (also *Cppcheck* claims to minimize false positives). We assume that the distribution of warnings throughout the code is roughly similar to the distribution of real faults in C code and can hence serve as a proxy to measure how configuration-related faults are distributed over the configuration space.

We performed the study on a fresh set of subject systems, that does not overlap with the corpus of *Study 1*: *expat*, *flex*, *gimp*, *gnumeric*, *gzip*, *kindb*, *mplayer*, *mpsolve*, *mptris*, *openldap*, *parrot*, *prc-tools*, *privoxy*, *sylpheed*, *tk*, *xine-lib*. We selected these systems guided by previous work [26, 27], which identified projects statically configurable with the C preprocessor.

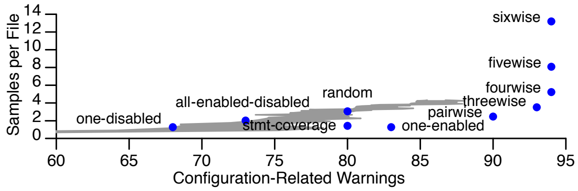
A.1 Results and Discussion

Overall, *Cppcheck* reported 96 warnings that appear only in specific configurations of the code over 77 distinct files. All 10 sampling algorithms reported more than 70% of the 96 configuration-related warnings, and no sampling algorithm reported all 96 warnings. We summarize the results of this experiment in Figure 8. Again, *five-wise* and *six-wise* reported the highest number of warnings. *One-disabled* and *statement-coverage* reported the lowest number of warnings. There is a warning for *Xine-lib*, where developers need to disable eight distinct configuration options to make *Cppcheck* report it. *Six-wise* misses this specific warning. However, other sampling algorithms, such as *most-enabled-disabled* and *one-enabled*, reported the warning for *Xine-lib*. Furthermore, we computed the number of warnings reported for the combinations of sampling algorithms and found combinations that reported all 96 warnings (e.g., *C2* and *C3*), as depicted in Figure 9.

The sizes of sample sets range from 1.3 to 13.2 configurations per file. Again, *six-wise* selected the highest number of configurations (more than 100K across all projects), while *one-enabled* and *one-disabled* selected the lowest number of configurations. The majority of the combinations of algorithms created a very large sample set. Figure 9 presents four combinations of sampling algorithms on the *Pareto Front*: *C2*, *C3*, *C5*, and *C6*.

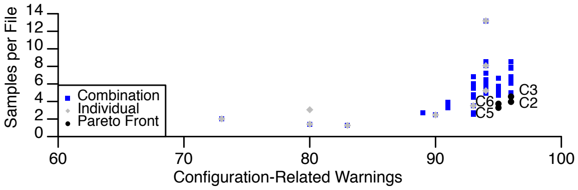
We computed the ranking of algorithms considering the efficiency function of Section 4.2. The algorithms, starting from the most efficient, are: *one-enabled*, *stmt-coverage*, *one-disabled*, *pair-wise*, *most-enabled-disabled*, *three-wise*, *random*, *four-wise*, *five-wise*, and *six-wise*. Overall, the ranking is stable when compared to *Study 1* and there were only minor changes: *most-enabled-disabled* and *pair-wise* are less efficient here, while *one-enabled*, *one-disabled*, and *statement-coverage* are more efficient. These changes can be explained by analyzing the number of files with only one configuration option, which is higher in our experiment than in *Study 1*. *Most-enabled-disabled* requires two configurations for each file with one configuration option; *one-enabled* and *one-disabled* require only one configuration per file. It makes *one-enabled* and *one-disabled* more efficient and impacts the ranking. Regarding the five least efficient algorithms, the ranking is exactly the same as in *Study 1*.

Study 1 and this experiment complement and confirm each other, as we obtain essentially the same results regarding the fault-detection capabilities of the sampling algorithms by using different perspectives: known faults reported in previous studies (*Study 1*) and *Cppcheck* as our fault-detected mechanism. We found two combinations of sampling algorithms (*C2*, and *C3*) that are on the *Pareto Front* of *Study 1* as well, which support them as efficient combinations. By triangulating the results, we gain confidence in the findings.



Sampling Algorithm	Faults	Samples
One-disabled	68	1.3
Most-enabled-disabled	73	2.0
Statement-coverage	80	1.4
Random	80	3.1
One-enabled	83	1.3
Pair-wise	90	2.5
Three-wise	93	3.5
Four-wise	94	5.2
Five-wise	94	8.1
Six-wise	94	13.2

Figure 8: Number of warnings reported and samples per file for each sampling algorithm considered in *Study 2*.



Sampling Algorithm
C2 One-enabled, one-disabled and statement-coverage
C3 One-enabled, one-disabled and most-enabled-disabled
C5 One-enabled, and most-enabled-disabled
C6 Pair-wise and one-enabled

	Faults	Samples		Faults	Samples
C2	96	4.0	C5	95	3.3
C3	96	4.6	C6	95	3.7

Figure 9: Number of faults and samples per file for the combinations of sampling algorithms on the *Pareto Front*.

[7] Variability-aware analysis tools, such as *TypeChef* [20, 21] and *SuperC* [15], could soundly cover all configurations regarding syntax or type errors, but would require a time-consuming initial setup that would make our study infeasible.