Investigating the Effects of T-Wise Interaction Sampling for Vulnerability Discovery in Highly-Configurable Software Systems

Tim Bächle
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
tim.baechle@kit.edu

Erik Hofmayer
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
erik.hofmayer@student.kit.edu

Christoph König
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
christoph.koenig@kit.edu

Tobias Pett
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
tobias.pett@kit.edu

Ina Schaefer
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
ina.schaefer@kit.edu

Abstract

Empirical evidence has shown that *variability bugs*, i.e., bugs that only manifest if certain features of a configurable software system are selected, are not only a theoretical concept. Many variability bugs involve an intricate interplay of multiple features, turning them into so-called *feature-interaction bugs*. The strategy of *t-wise interaction sampling* can be used to identify variability bugs in highly-configurable systems. In this regard, the number of findings, as well as the overall sample size, typically increase with stronger interaction sampling (i.e., higher t values). In this paper, we aim to confirm these observations for vulnerabilities. We use the static source code analysis platform Vari-Joern to analyze real-world highly-configurable software systems for the presence of vulnerability patterns using t-wise interaction sampling of varying strength and compare the number of findings and associated sample sizes. We analyze the feature configurations associated with the vulnerability warnings raised by our approach to evaluate the presence of feature-interaction vulnerabilities. Our results show that stronger interaction sampling produces a greater number of findings at a higher computational cost, also for vulnerabilities. The increase in findings can be attributed to the identification of feature-interaction vulnerabilities involving an interplay of a greater number of features.

CCS Concepts

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

Keywords

Vulnerability Discovery, Software Product Lines, Combinatorial Interaction Testing, T-Wise Interaction Sampling, Static Analysis

ACM Reference Format:
Tim Bächle, Erik Hofmayer, Christoph König, Tobias Pett, and Ina Schaefer. 2025. Investigating the Effects of T-Wise Interaction Sampling for Vulnerability Discovery in Highly-Configurable Software Systems. In *29th ACM International Systems and Software Product Line Conference - Volume A (SPLC-A '25), September 01–05, 2025, A Coruña, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744915.3748462

1 Introduction

A *vulnerability* is a weakness that can be exploited by one or more threats [37]. The exploitation of weaknesses in a system's source code by malicious users can lead to various types of harm, ranging from compromised sensitive data to damage to physical or virtual infrastructure [81]. A famous example is the Heartbleed vulnerability [33] that was identified in OpenSSL[1] in 2014. This vulnerability allowed sensitive user data, such as login credentials or private keys, to be read by exploiting the fact that the number of bytes to copy between two buffers could exceed the source buffer's length, thereby exposing uninitialized heap memory to the network [33, 80]. Considering the high costs vulnerabilities can cause [22, 52], analyzing software for their presence is often an indispensable activity in the development and maintenance process.

In the context of highly-configurable software systems, as they have become commonplace in many domains [6, 18], performing a thorough analysis is challenging [66, 75]. One challenge is the inherent variability and subsequent presence of *variability bugs*, i.e., bugs that only manifest in a subset of derivable software variants [1, 2, 57]. Bugs of this kind are common and have been identified in multiple real-world systems [1, 2, 57]. While the reasons for the presence of variability bugs are diverse, many have been found to be caused by specific combinations of two or more configurable features and the resulting interplay of feature-related source code [1, 2]. This class of bugs is referred to as *feature-interaction bugs* [1, 2]. Even though feature-interaction bugs have been the subject of multiple studies [1, 2, 27], it remains unclear how common feature-interaction bugs are among real-world systems [10, 27]. To reliably identify variability (and thus feature-interaction) bugs,

[1]https://www.openssl.org/.

every variant of a configurable system would need to be checked. As the number of variants that can be derived is up to exponential in the number of configurable features, this is typically infeasible [6, 54, 66, 73, 77]. In response, previous research has investigated various sampling approaches aimed at identifying a representative subset of variants to be analyzed [31, 54]. In this regard, t-wise interaction sampling is a popular sampling strategy that has demonstrated great effectiveness in identifying variability bugs [31, 54]. The goal of this strategy is to sample a set of variants such that all valid configurations between t-tuples of features are present in at least one variant [31, 53]. Using t-wise interaction sampling, two characteristics relating to an increase in interaction sampling strength (i.e., higher t value) have previously been observed [6, 31, 54]: (1) the total number of identified variability bugs is increased and (2) the resulting sample rapidly grows in size.

This paper aims to confirm these observations for vulnerabilities found in real-world highly-configurable software systems. We introduce the analysis platform Vari-Joern that allows the source code of a configurable system to be statically analyzed for the presence of potentially vulnerable source code patterns by utilizing popular sampling strategies (e.g., t-wise interaction sampling). Leveraging this platform, we analyze a set of real-world systems for the presence of common vulnerability patterns, collecting measurements on the number of vulnerability warnings and the sample size. These measurements allow us to assess the effects of using t-wise interaction sampling of varying strength for vulnerability discovery in configurable systems. In addition, we analyze the feature configurations associated with the vulnerability warnings identified throughout our experiments to assess whether changes between sampling strengths can be attributed to the identification of feature-interaction vulnerabilities involving a greater number of features.

In summary, we make the following contributions:

- We introduce the terminology of variability-induced and feature-interaction vulnerabilities, which is inspired by the related concepts of variability and feature-interaction bugs in configurable software systems (Section 3).
- We provide the analysis platform Vari-Joern for analyzing configurable software systems for the presence of potential vulnerabilities using popular sampling techniques and the static source code analysis tool Joern (Section 4).
- We examine the relationship between sampling strength t, sample size, and the number of variability-induced and feature-interaction vulnerabilities identified in the variants sampled from real-world highly-configurable software systems using t-wise interaction sampling (Section 5).

2 Background

In this section, we give an overview of fundamental concepts for highly-configurable software systems (Section 2.1) and detail the strategy of t-wise interaction sampling (Section 2.2).

2.1 Highly-Configurable Software Systems

Highly-configurable software systems have become commonplace in many domains [3, 18]. Even though the upfront engineering effort for a highly-configurable system is typically higher than for a non-configurable system, this effort usually pays off in the long term as it fosters systematic reuse [3, 16, 17]. This not only results in significant cost savings when software is tailored to many customers, but also reduces the time to market and allows quick reactions to changing market conditions [3, 15, 53, 70].

From a technical point of view, highly-configurable software systems are typically realized as *software product lines*. A software product line is a family of related software products (commonly referred to as *variants* [2, 39, 77]) that rely on a common code base and are distinguished by the *features* (i.e., increments in end-user-visible behavior [5, 14, 59, 70]) they provide [6, 16, 69, 73]. Typically, not every possible *configuration*, i.e., selection of features [27, 66], describes a valid or desired variant (e.g., due to mutually exclusive features) [3, 51, 73, 74]. As a result, a *feature model* usually models the valid configurations by specifying the available features and the constraints that exist between them [2, 3, 9, 39, 66, 73].

Many highly-configurable systems in practice are implemented following an *annotative* approach [2, 4, 11, 18]. This approach merges the source code realizing individual features into a single code base and annotates sections belonging to certain features or combinations thereof [3, 41]. Variants are derived by discarding all code not required for a specific feature configuration [3, 14]. In this context, propositional formulas[2] describing the feature configurations under which individual code fragments are incorporated into variants are referred to as *presence conditions* [1, 3, 43]. For systems implemented in C, annotations are typically realized using conditional compilation via the C preprocessor [3, 25, 50, 56, 69]. An example of this approach is illustrated in Listing 1. In the example, we follow common practice [1, 2, 44, 72] and prefix configuration-related preprocessor macros (i.e., features) with `CONFIG_`. Through the use of the `#ifdef` annotation in line 5, the `PROCESS_INPUT` feature controls whether lines 6 to 9 are included during preprocessing. Similarly, the `SEND_DATA` feature controls whether line 8 is included through the `#ifdef` annotation in line 7.

One problem that arises from configurable code artifacts is that standard static analysis techniques cannot be applied directly [25, 51, 60, 69]. A popular solution to this problem is to follow a *product-based* analysis strategy that derives variants from the configurable system and analyzes them individually [73]. Since the number of variants that can be derived from a configurable system is, in general, exponential in the number of features, considering all variants is typically infeasible [17, 39, 51, 56, 62]. The de facto standard is

[2]We assume that features are of Boolean type. In the presence of non-Boolean features, propositional formulas would not be sufficient.

```
1  void foo() {
2      int x = source(); // Attacker-controlled.
3      if(x < MAX){ // Does not enforce x >= 0.
4          int y = 0;
5  #ifdef CONFIG_PROCESS_INPUT
6          y = 2 * x;
7    #ifdef CONFIG_SEND_DATA
8          sink(y); // Security-sensitive operation.
9    #endif
10 #endif
11     }
12 }
```

Listing 1: A configurable C function adapted from the example provided by Yamaguchi et al. [79]

thus to employ sampling strategies that aim at the selection of a representative set of variants [51]. One particular sampling strategy that has proven to provide a good trade-off between sample size and adequate coverage of the configurable system is t-wise interaction sampling [6, 31].

2.2 T-Wise Interaction Sampling

T-wise interaction sampling has its origins in *Combinatorial Interaction Testing (CIT)*, a testing technique that leverages the observation that most bugs are caused by interactions between a small number of input parameters [27, 47, 62, 67]. Motivated by observations that many issues in configurable systems are caused by interactions between a small number of features [1, 2, 31, 55–57], t-wise interaction sampling extends the concept of CIT to the configuration level [27]. Instead of considering all variants derivable from a configurable system, t-wise interaction sampling aims to sample a set of variants such that all valid configurations between t-tuples of features appear in at least one variant [17]. In this context, the parameter t is used to control the sampling strength (i.e., the feature interactions covered by the sample) and is also referred to as the *interaction strength* of the sample [16, 17, 31]. An important characteristic is that sampling of strength t subsumes sampling of strength $t-1$ [17]. This is the case because the set of t-tuples built from a set of features F always acts as a superset to the set of $(t-1)$-tuples built from the same set of features F [17]. Determining a t-wise interaction sample is equivalent to the NP-complete minimum set cover problem [39]. As a result, numerous studies [16, 39, 40, 46, 53, 62] have proposed heuristic approaches that enable scalable approximations. Among them, the YASA algorithm proposed by Krieter et al. [46] is currently regarded as the best performing approach [45, 46, 65].

Table 1 shows an example for configurations selected using 1-wise and 2-wise interaction sampling for the `foo` function (cf. Listing 1). For 1-wise sampling, all valid configurations of 1-tuples of features (i.e., single features) must be present in at least one configuration. This requires two configurations, as each can only cover one possible configuration (selected or unselected) of each of the two features at a time. 2-wise sampling, also known as pairwise sampling [6, 75], requires all valid configurations of 2-tuples of features (i.e., pairs) to be present. In the example, the only possible 2-tuple is (`PROCESS_INPUT`, `SEND_DATA`). This tuple has four valid configurations (both features selected, both unselected, and two configurations where only one of the two features is selected). Accordingly, four configurations are required.

Strength	Config.	`PROCESS_INPUT`	`SEND_DATA`
$t = 1$	1	✓	✓
	2	✗	✗
$t = 2$	1	✓	✓
	2	✗	✗
	3	✗	✓
	4	✓	✗

Table 1: Example of configurations selected using 1-wise and 2-wise interaction sampling for the `foo` function of Listing 1

3 Vulnerabilities in Highly-Configurable Software Systems

As not every fragment of a configurable system's implementation is necessarily incorporated into every derivable software variant (cf. Section 2.1), vulnerabilities can be tied to specific configurations and, thus, might not always manifest. Accordingly, in this section, we introduce terminology for classifying vulnerabilities in highly-configurable software. We base this terminology on the concepts of variability and feature-interaction bugs, which capture similar observations for bugs in configurable software [1, 2, 27, 57].

Variability-Induced Vulnerabilities. When the source code involved in a vulnerability is at least partially tied to one or more variable features, the presence of the vulnerability depends on the feature configuration of the system. We define corresponding vulnerabilities as variability-induced vulnerabilities:

Definition 1. A *Variability-Induced Vulnerability (VIV)* is a vulnerability whose causing source code fragments are not confined to the common code shared between all variants of a configurable system. Instead, at least one code fragment is associated with variable features of the system. The vulnerability is, therefore, only present in some derivable variants.

Feature-Interaction Vulnerabilities (FIVs). The presence of a VIV depends on the configuration of at least one variable feature. In this regard, it is possible to distinguish between VIVs that manifest through a specific configuration of a single feature and those that require a specific configuration of multiple (i.e., two or more) features. We refer to the latter subset as feature-interaction vulnerabilities, defined as follows:

Definition 2. A *Feature-Interaction Vulnerability (FIV)* is a Variability-Induced Vulnerability (VIV) whose presence depends on the interaction between the selection or unselection of two or more features.

Example. We use the `foo` function (cf. Listing 1) to demonstrate a vulnerability that can be categorized as both a VIV and FIV. The highlighted code lines 2, 6, and 8 constitute a classic taint-style vulnerability [78, 80]: Potentially attacker-controlled data read from the `source` operation (line 2) is allowed to reach the security-sensitive `sink` operation (line 8) without previously undergoing proper sanitization. The vulnerability manifests in the variant for which both the `PROCESS_INPUT` and `SEND_DATA` features are selected. It does not manifest in all other variants, as either line 8 or both lines 6 and 8 will be discarded during preprocessing, depending on the feature selection. As a result, the vulnerability constitutes a VIV. Since its presence is not tied to a single feature, but depends on a feature interaction, it also constitutes a FIV.

4 Sample-Based Vulnerability Discovery

Due to the lack of off-the-shelf tooling capable of analyzing highly-configurable systems for the presence of potential vulnerabilities using sampling, we had to develop a sample-based vulnerability discovery approach (Section 4.1). We implemented this approach in the form of the novel analysis platform Vari-Joern (Section 4.2).

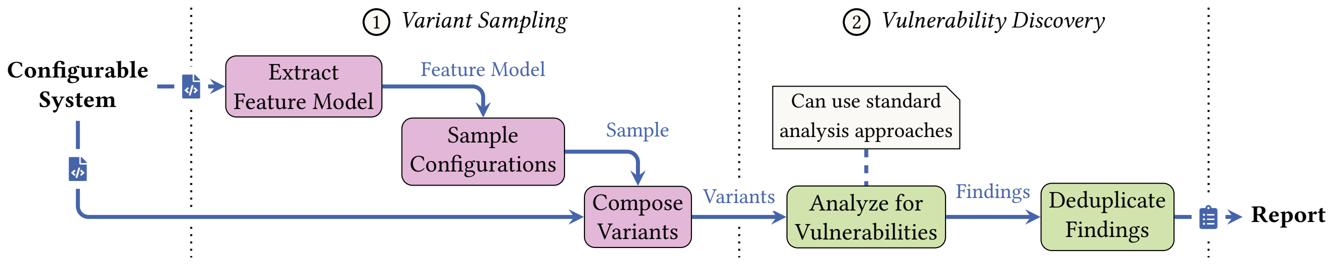

Figure 1: Fundamental concept of sample-based vulnerability discovery

4.1 Concept

The fundamental idea behind sample-based vulnerability discovery is to sample a representative set of variants from a configurable system that can then be analyzed for the presence of vulnerabilities using standard vulnerability analysis tooling. The structure of this process is illustrated in Figure 1 with two phases.

Variant Sampling. Information on the configuration space of the analyzed system must first be gathered to gain knowledge of the available features and to avoid sampling invalid configurations. This information is frequently encoded in the form of a feature model (cf. Section 2.1). Using the feature model, different sampling algorithms (e.g., t-wise interaction sampling) can be applied to construct a set of representative configurations. As these configurations represent only abstract descriptions of possible variants, actual variants have to be derived from the configurable system. In the case of systems implemented in C, this usually means that preprocessor macros linking features to specific code regions are either defined or undefined according to a sampled configuration. This step is crucial because it resolves the variability of the configurable system.

Vulnerability Discovery. Having resolved the variability during variant sampling, standard (i.e., variability-oblivious [61, 69]) analysis approaches can be applied to the sampled variants. Many approaches targeted at vulnerability discovery make use of static source code analysis [7, 20, 49, 55, 71, 79]. This means that they analyze the source code of a variant without executing it [22]. The goal is typically to build abstract representations of source code that approximate certain properties of a program and can be analyzed for typical vulnerability patterns [49]. Given that individual variants of a configurable system can share significant portions of source code (both through the common code and shared features), the findings reported by a vulnerability analysis approach across the sampled variants may include duplicates. To ease the interpretability of the results, such duplicates must be removed before reporting the findings to the user.

4.2 Realization

We realize the concept of sample-based vulnerability discovery (cf. Section 4.1) through VARI-JOERN,[3] a novel customizable analysis platform for highly-configurable systems implemented in C. The architecture of VARI-JOERN is illustrated in Figure 2 and consists of the five main components explained below.

Feature Model Reader. The feature model reader component extracts the feature model from a configurable system. For systems employing KCONFIG [42] for variability management, VARI-JOERN offers a reference implementation based on TORTE by Kuiter et al. [48] and the KMAX tool suite by Gazzillo et al. [28]. Support for other variability management approaches, such as UVL [9] or FEATUREIDE's [21] XML-based format, can be incorporated by adding a corresponding implementation to this component.

Sampler. The sampler component is responsible for generating a sample of configurations from a given feature model using a specified sampling strategy. While implementations for different sampling strategies can be freely added, two common sampling strategies in the form of t-wise interaction and uniform random sampling are currently supported. We decided to support uniform random sampling besides t-wise interaction sampling as it allows making observations that are representative of the complete configuration space [57, 64]. It therefore enables the analysis platform to serve as a foundation for future studies. The implementation for t-wise interaction sampling uses the YASA algorithm [46] found

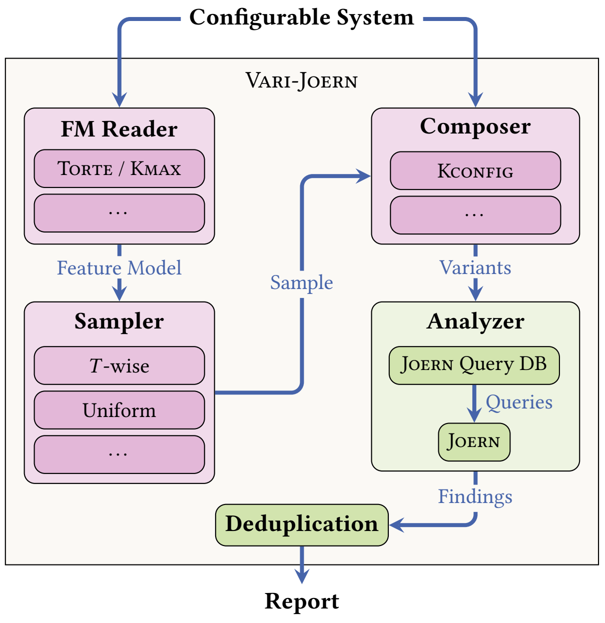

Figure 2: Architectural overview of VARI-JOERN

[3]https://doi.org/10.5281/zenodo.15647963.

in the FeatureIDE [21] library version 3.9.3. For uniform random sampling, the tool Smarch by Oh et al. [59] is used.

Composer. For a set of configurations, the composer component derives corresponding variants from a configurable system. The reference implementation available for Kconfig-based systems uses the Kconfig infrastructure and Makefiles to resolve variability. In addition, it determines the presence conditions associated with individual lines of code. To this end, Kmax [28] is used to determine presence conditions of entire source files, while SuperC [29] is used for presence conditions of individual lines within a file. For the final presence condition of a line of code, both are joined by conjunction.

Analyzer. For vulnerability analysis of individual variants, Vari-Joern utilizes the static source code analysis tool Joern.[4] Originally introduced by Yamaguchi et al. [79] as an open-source reference implementation for the source code representation of code property graphs, Joern has become a well-established tool that enjoys widespread use [12, 19, 30, 32, 81]. Even though Joern is officially marketed as a "bug hunter's workbench" [38], the ability to fully customize its analysis makes it well suited for the identification of vulnerabilities. Users can freely codify potentially vulnerable source code patterns of a program into *queries* that can be used to control the analysis process. Vari-Joern uses the default set of queries for C provided in the Joern query database,[5] as these queries are maintained by the community and target common issues that are prone to cause vulnerabilities.

Deduplication. The deduplication component analyzes a set of vulnerability warnings for the presence of duplicate findings. Findings are compared with regard to their causing Joern query and position within the configurable system's implementation. Those deemed identical are grouped together before being written to a plain text or JSON-based report file that is presented as output of Vari-Joern.

5 Evaluation

Petke et al. [63] demonstrated that stronger combinatorial interaction testing (cf. Section 2.2) increases the number of faults identified in non-configurable systems, while also significantly increasing the testing effort. Apel et al. [6], Medeiros et al. [54], and Halin et al. [31] have shown that these insights extend to the identification of variability bugs in configurable systems using t-wise interaction sampling. However, to the best of our knowledge, there is no study that investigates this trade-off for configurable systems and vulnerabilities. Consequently, we aim at answering the following research question RQ_1:

> RQ_1: What is the trade-off between the sample size and the number of findings when using t-wise interaction sampling of varying strength t for vulnerability discovery in highly-configurable software?

As described in Section 2.2, t-wise interaction sampling of greater sampling strength t ensures that more complex feature interactions are taken into account when building a sample. In addition, previous research has shown that feature-interaction bugs of varying feature interaction strengths exist in real-world systems [1, 2, 57]. Assuming that this insight extends to vulnerabilities and that FIVs therefore exist in real-world systems, stronger interaction sampling should be able to identify FIVs involving an interplay of more features (i.e., those having a higher *interaction strength*). To confirm whether this hypothesis proves true in practice, we formulate a second research question RQ_2 as shown below:

> RQ_2: Can the increase in vulnerability warnings reported using stronger t-wise interaction sampling be attributed to more FIVs of stronger interaction strength being identified?

5.1 Subject Systems

We selected three real-world software systems for the evaluation that are open source, highly configurable, and use Kconfig [42] to model the system's variability. Furthermore, the systems are supported by Vari-Joern and have been used as targets in evaluations of recent research on highly-configurable software [57, 59, 61, 64]. Table 2 characterizes the systems in more detail. The C-LoC column reports the total number of C code lines, measured using `cloc`,[6] considering only source files, as Vari-Joern targets the analysis of configurable systems implemented in C (cf. Section 4.2). The #Features column displays the number of features in the feature model extracted from the systems by Vari-Joern's feature model reader component (cf. Figure 2). The systems stem from different domains (an SSL library, a microkernel, and a UNIX utility collection for axTLS, Fiasco, and BusyBox, respectively) and range from less than 20,000 lines of C code and 63 features (axTLS) to more than 180,00 lines of C code and over 1,000 features (BusyBox). For axTLS and BusyBox, we used the latest stable release. For Fiasco, we used a recent commit to its code repository due to the unavailability of any more recent official releases.

5.2 Experimental Setup

To answer RQ_1, we conduct a quantitative study focusing on the number of reported vulnerability warnings and corresponding sample sizes for interaction sampling of varying sampling strength t. For RQ_2, we adopt a more qualitative study, analyzing the presence conditions associated with vulnerability warnings identified using stronger interaction sampling. All results and the accompanying evaluation scripts are made publicly available.[7]

RQ1. To assess the trade-off between sample size and number of vulnerability warnings for different sampling strengths (i.e., t values), we analyze all subject systems (cf. Table 2) using Vari-Joern. For each subject system, we perform a full analysis using t-wise interaction sampling of strengths $t \in \{1, 2, 3\}$. Since the version

[4]https://joern.io.
[5]https://queries.joern.io/.
[6]https://github.com/AlDanial/cloc.
[7]https://doi.org/10.5281/zenodo.15849290.

Name	Version	C-LoC	#Features
axTLS [8]	2.1.5	17,556	63
Fiasco [26]	Commit 4076045	46,013	99
BusyBox [13]	1.36.1	182,966	1,027

Table 2: The subject systems used for our evaluation

of the YASA sampling algorithm [46] employed by Vari-Joern is not deterministic in its execution, we repeat the previous step 10 times. The report files produced by Vari-Joern allow us to collect information on both the sample size and the resulting vulnerability warnings for each run. As a post-processing step, we filter out all warnings relating to files of a subject system’s build infrastructure (e.g., Kconfig or files generated during the build process) before counting the vulnerability warnings for a specific run.

To facilitate the comparison between the results for different sampling strengths, we determine Spearman’s rank correlation coefficient ρ [35, 68] between t and the sample size, as well as between t and the number of vulnerability warnings. This allows us to assess whether an increase in the sampling strength t tends to correspond with an increase in the sample size and the number of vulnerability warnings. We use the Spearman rank correlation coefficient, as empirical evidence from real-world variability bugs indicates that the relationships between the sampling strength t and factors like sample size or number of findings are not necessarily linear [31, 54]. Measuring the linearity of a correlation (e.g., through the Pearson correlation coefficient) therefore proves to be insufficient [35, 68].

RQ2. For investigating whether the increase in vulnerability warnings caused by stronger interaction sampling can be attributed to FIVs of greater interaction strength, we use the results collected for answering RQ_1. Specifically, we examine the post-processed vulnerability warnings originally produced from an analysis of BusyBox using t-wise interaction sampling of strengths $t \in \{1, 2, 3\}$. We chose to focus on the results for BusyBox as it constitutes the largest subject system with regard to both lines of code and features (cf. Table 2). Given the increased complexity compared to axTLS and Fiasco, it promises to yield the most meaningful results.

For each of the 10 analysis runs performed for BusyBox and sampling strength $t \in \{1, 2, 3\}$, we first filter out all vulnerability warnings for which Vari-Joern was not able to determine a presence condition based on the information supplied by its composer component (cf. Section 4.2). We leverage the presence conditions of the remaining warnings for determining their *true interaction strength*. In this regard, we follow a brute-force analysis approach. We search for a partial configuration of minimal size that satisfies two conditions: (1) it is valid with respect to the feature model, and (2) it implies the presence condition of the respective warning given the feature model’s constraints. To reduce the number of partial configurations that have to be evaluated, we perform an optimization that limits the search to features potentially relevant for the presence condition. At first, we record all features that directly contribute to satisfying the presence condition. We transitively extend this set of features using clauses of the conjunctive normal form of the feature model’s constraints to factor in implication chains formed by multiple constraints. The size of the partial configuration resulting from this approach then represents the true interaction strength of a particular vulnerability warning.

Having determined their true interaction strengths, we group vulnerability warnings across all considered sampling strengths t according to the causing Joern query and affected source code location to identify unique findings. This enables us to identify the minimum sampling strength required for the discovery of each finding. Comparing the minimum sampling strength with the true interaction strength derived via our brute-force analysis approach, we can judge whether theory proves right in practice and stronger t-wise interaction sampling is capable of identifying FIVs of greater interaction strength.

Execution Environment. For the analysis of the selected subject systems, we execute Vari-Joern within a dedicated Docker container that uses an image based on Ubuntu 24.04.1. The container was run on a server equipped with a 16-core AMD Ryzen Threadripper PRO 5955WX processor and 128 GB of RAM, running Ubuntu 24.04.2 LTS.

5.3 Results

RQ1. Figure 3 illustrates the distribution of sample sizes across all considered subject systems and sampling strengths as individual box plots. The average sample sizes are illustrated by colored diamonds. While only a few configurations suffice for achieving 1-wise feature coverage, the sample size rapidly increases for $t = 2$ and exceeded 100 for $t = 3$ across all subject systems. For all sampling strengths, the resulting sample size was always greatest for BusyBox, which is the system with the highest number of features.

Investigating the effects greater sampling strengths have on vulnerability discovery, Figure 4 shows the distribution of the total number of vulnerability warnings for all three subject systems and $t \in \{1, 2, 3\}$. The number of vulnerability warnings generally increases with stronger interaction sampling (i.e., greater t). While increasing the sampling strength from $t = 1$ to $t = 2$ results in a noticeable increase in the number of findings, the same magnitude of growth cannot be observed for the increase from $t = 2$ to $t = 3$. In fact, for axTLS, there was no increase in the number of vulnerability warnings between $t = 2$ and $t = 3$ across all 10 analysis runs.

To ease the interpretation of the results regarding the sample size and number of vulnerability warnings for different sampling strengths, Table 3 shows the corresponding Spearman rank correlation coefficients. ρ_{sample} represents the correlation between the sampling strength t and the sample size, whereas $\rho_{\text{vulnerabilities}}$ represents the correlation between the sampling strength t and the number of raised vulnerability warnings. Based on the interpretation of correlation coefficients presented by Schober et al. [68], the results for ρ_{sample} show a very strong correlation for all three

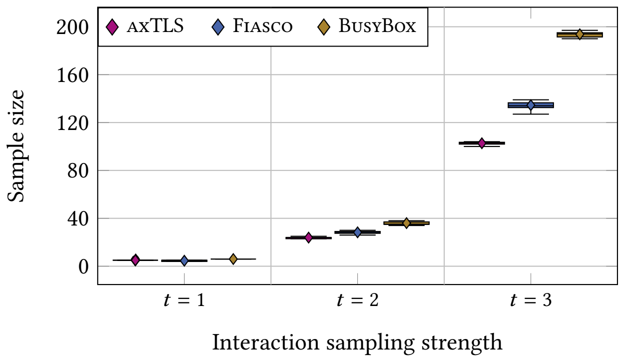

Figure 3: Relationship between sampling strength t and sample size for the three subject systems

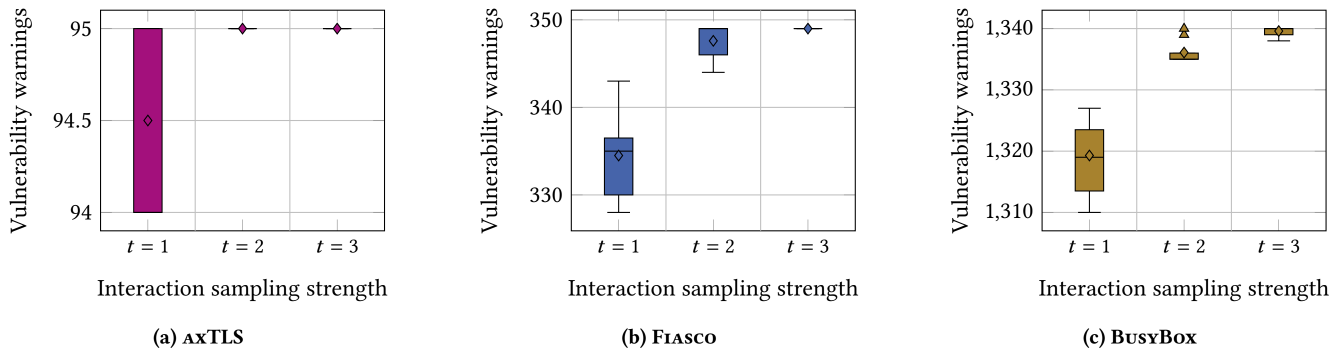

Figure 4: Relationship between interaction sampling strength t and total number of vulnerability warnings

subject systems. Similarly, the results for $\rho_{\text{vulnerabilities}}$ show a very strong correlation for BusyBox, a strong correlation for Fiasco, and a moderate correlation for AxTLS.

RQ2. Concentrating on the largest subject system, BusyBox, Figure 5 shows the totaled shares of unique vulnerability warnings raised during the 10 analysis runs with regard to three types: (1) those first raised at a sampling strength equal to their interaction strength, (2) those first raised at a sampling strength lower than their interaction strength, and (3) those raised at a sampling strength greater than their interaction strength. It provides an overview of the sampling strength required to raise particular vulnerability warnings. The majority of warnings (88.96%) are first raised at a sampling strength equal to their true interaction strength. Additionally, small shares are first raised using sampling of lower (2.76%) and higher (8.28%) strength than their true interaction strength.

With FIVs requiring the interaction of at least two features (cf. Section 3), Figure 6 focuses on the vulnerability warnings raised on BusyBox using 2-wise and 3-wise interaction sampling. Specifically, it excludes all vulnerability warnings already raised using a weaker sampling strength and categorizes the remaining warnings based on their interaction strength as well as the sampling strength t required for their discovery. The result is a series of box plots, reflecting the distributions of vulnerability warnings first raised using 2-wise but not 1-wise, and 3-wise but not 2-wise sampling, respectively. The majority of warnings raised using 2-wise but not 1-wise interaction sampling relates to code segments of interaction strength 2, with a smaller number relating to code of interaction strengths 1 and 3. In contrast, the vulnerability warnings raised by 3-wise but not 2-wise interaction sampling are primarily related to code of interaction strength 1, with only a small number being

Subject System	ρ_{sample}	$\rho_{\text{vulnerabilities}}$
AxTLS [8]	0.96	0.55
Fiasco [26]	0.95	0.87
BusyBox [13]	0.96	0.92

Table 3: Spearman's rank correlation coefficients between t and the sample size (ρ_{sample}) and between t and the number of vulnerability warnings ($\rho_{\text{vulnerabilities}}$)

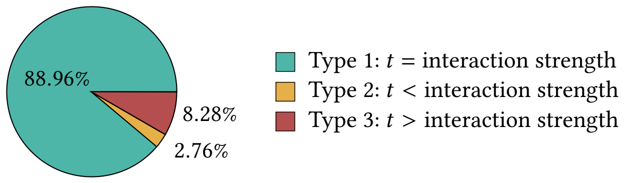

Figure 5: Totaled shares of vulnerability warnings for BusyBox identified by interaction sampling of strength t compared to the true interaction strength

related to code of interaction strength 3. For both 2-wise or 3-wise sampling, none of the vulnerability warnings not already raised by weaker sampling are related to code having an interaction strength greater than 3.

5.4 Discussion

RQ1. Regarding the trade-off between sample size and number of raised vulnerability warnings for t-wise interaction sampling of varying strength, we consider the effects sampling strength has on both metrics. For sample size, a near-exponential relationship in the interaction sampling strength t appears to emerge for all three subject systems (cf. Figure 3). Ignoring constraints between features, a set of t Boolean features alone has 2^t configurations. As

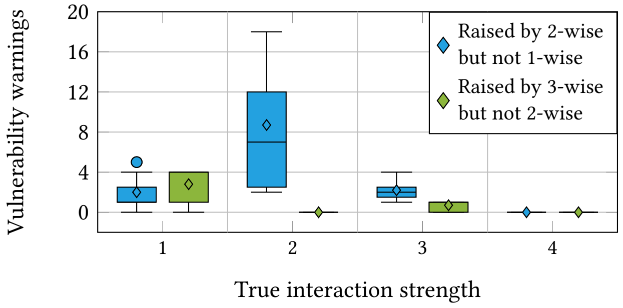

Figure 6: Distribution of vulnerability warnings raised in BusyBox using stronger interaction sampling for interaction strengths ranging from 1 to 4

t-wise interaction sampling aims at covering all valid configurations of t features (cf. Section 2.2), a near-exponential relationship is therefore expected [45] and is in line with empirical results of previous studies [31, 54]. This relationship also explains why the Spearman rank correlation coefficients show a very strong correlation between sampling strength and sample size across all three systems (cf. Table 3). Apart from $t = 1$ for axTLS and Fiasco, we also notice that the sample size for a given t is generally larger for systems that have a more complex configuration space with regard to the number of available features (cf. Figure 3 and Table 2). Again, this is consistent with both empirical results [54] and general expectation. More features lead to more possible tuples of t features whose valid configurations must be covered.

Looking at the number of raised vulnerability warnings, a different relationship can be observed. While the correlation between sampling strength t and number of raised vulnerability warnings is moderate for axTLS and high for both Fiasco and BusyBox (cf. Table 3), an increase in sampling strength does not necessarily translate to a significant increase in vulnerability warnings. In fact, the relationship appears to be almost logarithmic in the sampling strength t across all subject systems (cf. Figure 4). As a result, the benefits of using stronger sampling for vulnerability discovery quickly diminish. This observation closely resembles previous findings made in the context of variability bugs [31, 54]. It also intuitively makes sense, as (1) the number of potential vulnerabilities within a system is limited, meaning there must be a plateau in vulnerability warnings, and (2) empirical results indicate that most issues caused by feature interactions involve a small number of features [1, 2, 31, 55–57]. As a result, interaction sampling of low strength is typically able to identify most potential vulnerabilities. Increasing the sampling strength then enables the identification of the rare occasions where many features are involved until the plateau formed by all existing vulnerabilities is ultimately reached.

Taking into account the results and effects described above, we find that the sampling strength t has different effects on sample size and vulnerability discovery capabilities. Notably, while an increase in sampling strength increases the vulnerability discovery capabilities, it also rapidly increases the sample size and thus the required analysis effort. We can therefore answer RQ_1 as follows:

Conclusion RQ_1: Increasing the strength of t-wise interaction sampling increases the sample size and the number of reported vulnerability warnings. Although the magnitude of this effect increases rapidly for the sample size, it simultaneously quickly diminishes for the number of vulnerability warnings. We can therefore extend previous observations of a trade-off situation, made in the context of variability bugs, to vulnerabilities in configurable software. Based on our experiments, 2-wise interaction sampling seems to strike the optimal balance in this trade-off between sample size and vulnerability discovery capabilities.

RQ2. For investigating whether the increase in vulnerability warnings caused by stronger interaction sampling can be attributed to FIVs of higher interaction strength, we take a closer look at the classification shown in Figure 5. Considering the sampling strength t required for discovering a potential vulnerability for the first time, the classification categorizes vulnerability warnings based on their true interaction strength into three types: (1) t = interaction strength, (2) $t <$ interaction strength, and (3) $t >$ interaction strength.

Instances of type 1 reflect the expected behavior of t-wise interaction sampling. As all valid interactions between t features are deliberately covered (cf. Section 2.2), it is expected that vulnerabilities with a corresponding interaction strength are discovered. It is therefore not surprising that the majority of vulnerability warnings raised during our experiments belong to type 1 (cf. Figure 5).

Instances of type 2, on the other hand, are a result of so-called *collateral effects* [31, 63]. While t-wise interaction sampling covers all possible interactions between t features, it also yields incomplete coverage of interactions involving more than t features. It can therefore happen that a vulnerability with an interaction strength greater than t is discovered by coincidence. The chances of this happening in the case of a complex configuration space are however small, confirmed by the less than 3% of vulnerability warnings of this type identified during our experiments (cf. Figure 5).

Instances of type 3 are of special interest, as in theory, these instances should not exist. For example, a vulnerability warning related to a source code segment of interaction strength 1 should be discovered by 1-wise sampling, as it covers all possible configurations of single features. Therefore, discovering such a finding only at sampling strengths greater than 1 contradicts expectations. Fortunately, more than 95% of type 3 instances identified during our experiments (cf. Figure 5) can be explained by vulnerability warnings associated with an interaction strength of 0.[8] When first discovering these instances using 1-wise sampling (the weakest sampling strength considered), the sampling strength of 1 naturally exceeds the underlying interaction strength of 0. The remaining instances of type 3 can be attributed to a technical limitation of Vari-Joern. The composer component (cf. Section 4.2) only determines presence conditions on the granularity of single lines of code. However, through manual analysis, we found that the presence condition of a whole line of code can differ from the presence condition of individual tokens located within. In this regard, we identified two patterns causing this behavior: (1) the use of a macro, whose substitution varies based on the configuration of specific features, and (2) the presence of a token, whose meaning changes between a function call and a macro substitution depending on the feature configuration. In both cases, Vari-Joern issues an incomplete presence condition that causes our brute-force analysis approach to report a lower interaction strength. Through manual identification of the presence conditions of corresponding instances, we found that the correct interaction strength is always greater or equal to the used sampling strength t.

These insights allow us to reason about the interaction strengths of the vulnerability warnings raised only using stronger interaction sampling of strengths $t = 2$ and $t = 3$ (cf. Figure 6). Vulnerability warnings for which our brute-force analysis approach determined an interaction strength of 1 all belong to type 3 and are caused by the aforementioned limitation of Vari-Joern with regard to presence conditions. In fact, we found that all vulnerability warnings of this type are at least of interaction strength 2. The ones raised by 3-wise but not 2-wise sampling are even all of interaction strength 6

[8]An interaction strength of 0 indicates that a potential vulnerability is not associated with any variable feature and therefore shared between all variants of the system.

and thus a clear result of collateral effects. The warnings with an interaction strength of 2 that were raised by 2-wise but not 1-wise sampling are instances of type 1. The same applies to the warnings with an interaction strength of 3 that were raised by 3-wise but not 2-wise sampling. Last, the warnings with an interaction strength of 3 that were raised by 2-wise but not 1-wise sampling are again examples of type 2 and caused by collateral effects.

For 2- and 3-wise interaction sampling, we therefore find that all vulnerability warnings raised only using stronger sampling are instances of types 1 and 2. Based on this key finding and the discussion that preceded it, we can answer RQ_2 as follows:

Conclusion RQ_2: Within the scope of our experiments, we find that vulnerability warnings raised only using stronger interaction sampling are always associated with code having an interaction strength that meets or exceeds the sampling strength used. More specifically, we find that the warnings first raised using 2-wise and 3-wise sampling have at least a true interaction strength of 2 and 3, respectively. As a result, they represent warnings for potential FIVs. For 2-wise and 3-wise interaction sampling, we can therefore confirm the expectation that the increase in the number of vulnerability warnings resulting from stronger sampling can be attributed to FIVs of a higher interaction strength.

5.5 Threats to Validity

Internal Validity. Using the Joern query database for the analysis with Vari-Joern (cf. Section 4.2) poses a threat to the internal validity of our results. While queries in the database generally model source code patterns that are often exploitable (e.g., a copy operation allowing uninitialized data to remain in a buffer), some only check for the presence of simple structures, such as a call to a potentially unsafe function. These simpler queries might not allow us to capture the true effects that t-wise interaction sampling has on the discovery of complex vulnerabilities. However, since the majority of queries in the database model more advanced source code patterns, the resulting bias should be limited. Furthermore, there is no publicly available collection of queries solely dedicated to complex vulnerability patterns. While using such a collection would help minimize the threat, its creation would require detailed expert knowledge of both the software systems and complex vulnerability patterns and is therefore beyond the scope of this paper.

A threat affecting our results for RQ_2 is that we had to exclude certain vulnerability warnings from further consideration. We had to exclude warnings where Vari-Joern was not able to determine a presence condition. Additionally, some presence conditions represented contradictions, as the corresponding code regions are incorporated only for execution environments different from ours. As a result of excluding both types, the results relating to RQ_2 might be biased. However, given the relatively small number of vulnerability warnings that had to be excluded (across all 10 executions and for all $t \in \{1, 2, 3\}$ 13.41% were missing a presence condition and a further 0.23% had a contradictory presence condition), we argue that the extent of this bias should be small.

External Validity. A threat to the external validity of our results lies in the fact that we based our experiments on a limited set of three configurable systems. While we tried to choose real-world systems of moderate to large size that were also used in recent research on highly-configurable systems (cf. Section 5.1), our insights might not extend to highly-configurable systems as a whole.

Moreover, we only considered interaction sampling of strength $t \in \{1, 2, 3\}$, even though issues caused by feature interactions involving more features have been shown to exist [1, 2, 31, 55, 56]. Our results may therefore not generalize to higher sampling strengths. However, as empirical research [1, 2, 31, 55, 56] has indicated, issues caused by feature interactions involving more than three features are rare. In addition, applying higher sampling strengths in the context of real-world systems quickly results in prohibitive computational costs [54, 66]. Both aspects raise serious questions about the viability and relevance of stronger interaction sampling for vulnerability discovery in practice.

6 Related Work

This paper investigates the effects of using t-wise interaction sampling with varying sampling strength t for vulnerability discovery in highly-configurable software systems. As a result, it connects to three main areas of research: (1) approaches for the identification of issues in configurable systems using sampling, (2) characterizations of issues found in configurable systems, and (3) comparisons between sampling strategies.

Identification of Issues Using Sampling. Mordahl et al. [57] propose a customizable sampling-based approach for the discovery of variability bugs in configurable systems. They evaluated their approach on three C-based configurable systems using four off-the-shelf static analysis tools for bug discovery. Our approach uses Joern to target vulnerabilities instead of bugs. Furthermore, we use t-wise interaction sampling, whereas they rely on uniform random sampling of configurations, realized using Smarch by Oh et al. [59].

Similarly, Medeiros et al. [55] propose a sampling-based approach for detecting code weaknesses in configurable systems and evaluated it using 24 configurable systems implemented in C. While their approach is similar to ours, they used a custom linear sampling strategy instead of t-wise interaction sampling. Furthermore, they analyzed variants with the static analysis tool Cppcheck,[9] whereas our approach relies on Joern. To determine whether reported weaknesses are configuration-related, they performed a manual analysis rather than automatically deriving presence conditions.

Characterization of Issues in Configurable Systems. Several studies [1, 2, 25, 27, 56, 58] investigated characteristics of issues found in configurable software systems. Garvin and Cohen [27] conducted an explorative study on the distribution of feature-interaction bugs in Firefox and GCC. Analyzing reports from the systems' bug tracking systems, they extracted 28 variability bugs, of which only 3 could be confirmed as feature-interaction bugs.

Motivated by the absence of a publicly available database of variability bugs, Abal et al. [1] performed a qualitative study of 42 real-world variability bugs identified in the Linux kernel by inspecting bug-fixing commits. They consolidated their findings into a database detailing properties such as type, cause, and interaction strength of individual variability bugs. In a follow-up study [2], this

[9]https://cppcheck.sourceforge.io/.

database was extended to 98 variability bugs by also considering bug-fixing commits from Apache, BusyBox, and Marlin.

Leveraging the variability-aware parser TypeChef [43] for identifying issues in configurable systems, Medeiros et al. [56] analyzed 15 real-world systems for faults and warnings. They characterized findings with regard to the reasons for their introduction, the number of features involved, and the time until the issue was resolved.

Ferreira et al. [25] followed a similar approach and used TypeChef [43] for examining the relationship between vulnerabilities and configuration complexity in the Linux kernel. Analyzing previously vulnerable functions, they were able to conclude that vulnerable functions are usually associated with higher variability (i.e., more `#ifdefs`) and constrained by fewer features.

Extending the consideration of vulnerabilities, Muniz et al. [58] focused on characteristics of weaknesses (i.e., mistakes in code that can lead to vulnerabilities when exploited) and how developers try to discover them. To this end, they conducted a qualitative analysis of 27 weaknesses identified through the bug tracking systems of Apache, HTTPD, Linux, and OpenSSL and performed a survey involving 110 of the systems' developers.

In contrast to these studies, we use an automated sample-based approach for identifying findings across configurable systems. Furthermore, we analyze vulnerability warnings reported by our approach only with regard to their presence condition and the associated true interaction strength, without considering any other characteristics of the findings.

Comparisons Between Sampling Strategies. Multiple studies have focused on classifying [75] or comparing [6, 24, 31, 51, 54, 76] different sampling strategies and their implementations. Varshosaz et al. [75] propose a classification of sampling strategies based on input data, algorithm, coverage, and evaluation. They applied this classification to 48 studies from the literature that either introduced new strategies or evaluated existing ones. Apel et al. [6], Liebig et al. [51], and von Rhein et al. [76] all compared different sampling strategies against each other and to family-based alternatives that aim to consider not variants but the configurable system as a whole [6, 18, 36, 51]. To this end, they compared the different approaches in the context of static analyses [51, 76] and model checking [6]. Using a set of known bugs as ground truth, Medeiros et al. [54] compared 10 sampling strategies with regard to resulting sample size and bug detection capabilities. Focusing specifically on t-wise interaction sampling, Ferreira et al. [24] compared five different implementations for sampling strengths ranging from $t = 1$ to $t = 4$, using brute force (i.e., exhaustive) and random sampling strategies as baselines. In contrast, Halin et al. [31] aimed to establish an exhaustive ground truth for comparison between sampling strategies. They built and tested all possible variants of the configurable web application code generator JHipster. Using the set of variants causing either failures or faults (i.e., defects) as ground truth, they compared 20 sampling strategies with regard to their defect-finding capabilities and sample sizes.

For our work, we investigate the effects of using the strategy of t-wise interaction sampling for vulnerability discovery. Instead of classifying or comparing different sampling strategies or implementations thereof, we vary only the interaction strength parameter t to assess its impact on vulnerability discovery.

In Summary. This paper combines aspects of all three aforementioned areas of research. It extends the work on the identification of issues in configurable systems by introducing a dedicated analysis platform that uniquely integrates t-wise interaction sampling with the powerful analysis tool Joern. We further contribute to the characterization of issues in configurable systems through an automated analysis of the true interaction strength of potential vulnerabilities. Moreover, we expand on previous work comparing different sampling strategies by assessing how varying sampling strengths of t-wise interaction sampling impact vulnerability discovery. Taken together, our work extends existing observations on the effects of t-wise feature interaction sampling for the detection of software faults to the discovery of vulnerabilities.

7 Conclusion

Being able to identify vulnerabilities in software as early and efficiently as possible is crucial to prevent the disastrous consequences exploited vulnerabilities can have. In this paper, we investigated the effects of t-wise interaction sampling for vulnerability discovery in highly-configurable software systems. To characterize vulnerabilities in configurable systems, we introduced the terminology of variability-induced and feature-interaction vulnerabilities. Additionally, we proposed an approach for sample-based vulnerability discovery, which we implemented in the analysis platform Vari-Joern. Using this platform, we were able to extend previous observations about the trade-off between sample size and number of findings, originally made in the context of variability bugs, to vulnerabilities. Our results showed that the increase in findings incurred by stronger sampling could be attributed to feature-interaction vulnerabilities involving an interplay of a greater number of features.

As part of future work, we plan to expand the set of queries used by Vari-Joern by identifying additional vulnerability patterns and modeling them as Joern queries. Considering the interest in Uniform Random Sampling (URS) approaches for analyzing configurable software systems [23, 34, 57], we also want to extend our considerations of the effects of different sampling approaches to URS. Finally, with family-based analysis approaches having shown great potential for the discovery of bugs [6, 36, 51, 60, 61, 69, 76], we plan to investigate whether previous observations made in this context also extend to vulnerability discovery.

Acknowledgments

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. The authors would like to thank the Ministry of Science, Research and Arts of the Federal State of Baden-Württemberg for the financial support of the project FLT1 - Dev4Test within the InnovationCampus Future Mobility.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 421–432. doi:10.1145/2642937.2642990

[2] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2017. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology* 26, 3, Article 10 (July 2017), 34 pages. doi:10.1145/3149119

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation* (1 ed.). Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/978-3-642-37521-7

[4] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *The Journal of Object Technology* 8, 5 (2009), 49–84. doi:10.5381/jot.2009.8.5.c5

[5] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions Using Feature-Aware Verification. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE, USA, 372–375. doi:10.1109/ASE.2011.6100075

[6] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Groslinger, and Dirk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *2013 35th International Conference on Software Engineering (ICSE '13)*. IEEE, USA, 482–491. doi:10.1109/ICSE.2013.6606594

[7] Marcelo Arroyo, Francisco Chiotta, and Francisco Bavera. 2016. An User Configurable Clang Static Analyzer Taint Checker. In *2016 35th International Conference of the Chilean Computer Science Society (SCCC '16)*. IEEE, USA, 1–12. doi:10.1109/SCCC.2016.7835996

[8] axTLS 2024. axTLS Embedded SSL. https://axtls.sourceforge.net/

[9] David Benavides, Chico Sundermann, Kevin Feichtinger, José A. Galindo, Rick Rabiser, and Thomas Thüm. 2025. UVL: Feature Modelling with the Universal Variability Language. *Journal of Systems and Software* 225 (July 2025), 112326. doi:10.1016/j.jss.2024.112326

[10] Sabrina Böhm, Sebastian Krieter, Tobias Heß, Thomas Thüm, and Malte Lochau. 2024. Incremental Identification of T-Wise Feature Interactions. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '24)*. ACM, New York, NY, USA, 27–36. doi:10.1145/3634713.3634715

[11] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. 2013. Intraprocedural Dataflow Analysis for Software Product Lines. In *Transactions on Aspect-Oriented Software Development X*. Vol. 7800. Springer Berlin Heidelberg, Berlin, Heidelberg, 73–108. doi:10.1007/978-3-642-36964-3_3

[12] Van-Cong Bui and Xuan-Cho Do. 2023. Detecting Software Vulnerabilities Based on Source Code Analysis Using GCN Transformer. In *2023 RIVF International Conference on Computing and Communication Technologies (RIVF '23)*. IEEE, USA, 112–117. doi:10.1109/RIVF60135.2023.10471834

[13] BusyBox 2024. BusyBox. https://busybox.net/

[14] Thiago Castro, Leopoldo Teixeira, Vander Alves, Sven Apel, Maxime Cordy, and Rohit Gheyi. 2021. A Formal Framework of Software Product Line Analyses. *ACM Transactions on Software Engineering and Methodology* 30, 3 (July 2021), 1–37. doi:10.1145/3442389

[15] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 335–344. doi:10.1145/1806799.1806850

[16] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering* 34, 5 (Sept. 2008), 633–650. doi:10.1109/TSE.2008.50

[17] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2006. Coverage and Adequacy in Software Product Line Testing. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA '06)*. ACM, New York, NY, USA, 53–63. doi:10.1145/1147249.1147257

[18] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2019. Finding Suitable Variability Abstractions for Lifted Analysis. *Formal Aspects of Computing* 31, 2 (April 2019), 231–259. doi:10.1007/s00165-019-00479-y

[19] Xiang Du, Liangze Yin, Peng Wu, Liyuan Jia, and Wei Dong. 2020. Vulnerability Analysis through Interface-based Checker Design. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C '20)*. IEEE, USA, 46–52. doi:10.1109/QRS-C51114.2020.00019

[20] David Evans and David Larochelle. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software* 19, 1 (Aug. 2002), 42–51. doi:10.1109/52.976940

[21] FeatureIDE 2025. FeatureIDE. https://featureide.github.io/

[22] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. 2016. Chapter One - Security Testing: A Survey. In *Advances in Computers*. Vol. 101. Elsevier, Cambridge, MA, USA, 1–51. doi:10.1016/bs.adcom.2015.11.003

[23] David Fernandez-Amoros, Ruben Heradio, Jose Miguel Horcas Aguilera, José A. Galindo, David Benavides, and Lidia Fuentes. 2024. Pragmatic Random Sampling of the Linux Kernel: Enhancing the Randomness and Correctness of the Conf Tool. In *28th ACM International Systems and Software Product Line Conference (SPLC '24)*. ACM, New York, NY, USA, 24–35. doi:10.1145/3646548.3672586

[24] Fischer Ferreira, Gustavo Vale, João P. Diniz, and Eduardo Figueiredo. 2021. Evaluating T-wise Testing Strategies in a Community-Wide Dataset of Configurable Software Systems. *Journal of Systems and Software* 179 (Sept. 2021), 110990. doi:10.1016/j.jss.2021.110990

[25] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 65–73. doi:10.1145/2934466.2934467

[26] Fiasco 2025. The L4Re Microkernel Repository. https://github.com/kernkonzept/fiasco

[27] Brady J. Garvin and Myra B. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE '11)*. IEEE, USA, 90–99. doi:10.1109/ISSRE.2011.25

[28] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, New York, NY, USA, 279–290. doi:10.1145/3106237.3106283

[29] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 323–334. doi:10.1145/2254064.2254103

[30] Lea Gerling and Klaus Schmid. 2019. Variability-Aware Semantic Slicing Using Code Property Graphs. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A (SPLC '19)*. ACM, New York, NY, USA, 65–71. doi:10.1145/3336294.3336312

[31] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. *Empirical Software Engineering* 24, 2 (April 2019), 674–717. doi:10.1007/s10664-018-9635-4

[32] Zhang Haojie, Li Yujun, Liu Yiwei, and Zhou Nanxin. 2021. Vulmg: A Static Detection Solution For Source Code Vulnerabilities Based On Code Property Graph and Graph Attention Network. In *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP '21)*. IEEE, USA, 250–255. doi:10.1109/ICCWAMTIP53232.2021.9674145

[33] Heartbleed 2024. The Heartbleed Bug. https://heartbleed.com/

[34] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, David Benavides, and Don Batory. 2022. Uniform and Scalable Sampling of Highly Configurable Systems. *Empirical Software Engineering* 27, 2 (March 2022), 44. doi:10.1007/s10664-021-10102-5

[35] Christian Heumann, Michael Schomaker, and Shalabh. 2022. *Introduction to Statistics and Data Analysis: With Exercises, Solutions and Applications in R*. Springer International Publishing, Cham. doi:10.1007/978-3-031-11833-3

[36] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2017. Effective Analysis of C Programs by Rewriting Variability. *The Art, Science, and Engineering of Programming* 1, 1 (Jan. 2017), 1. doi:10.22152/programming-journal.org/2017/1/1

[37] ISO/IEC. 2018. ISO/IEC 27000:2018(E) - Information Technology - Security Techniques - Information Security Management Systems - Overview and Vocabulary.

[38] Joern 2024. Joern - The Bug Hunter's Workbench. https://joern.io/

[39] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS '11, Vol. 6981)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 638–652. doi:10.1007/978-3-642-24485-8_47

[40] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 46–55. doi:10.1145/2362536.2362547

[41] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 311. doi:10.1145/1368088.1368131

[42] Kconfig 2025. Kconfig. https://docs.kernel.org/kbuild/kconfig-language.html

[43] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #ifdef Variability in C. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM, New York, NY, USA, 25–32. doi:10.1145/1868688.1868693

[44] Adam Krafczyk. 2019. *Variability-Aware Analysis of C Source Code.* Master's thesis. University of Hildesheim, Hildesheim. https://sse.uni-hildesheim.de/media/fb4/informatik/AG_SSE/Adam_Krafczyk.pdf

[45] Sebastian Krieter. 2020. Large-Scale T-wise Interaction Sampling Using YASA. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A (SPLC '20)*. ACM, New York, NY, USA, 1–4. doi:10.1145/3382025.3414989

[46] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20)*. ACM, New York, NY, USA, 1–10. doi:10.1145/3377024.3377042

[47] Daniela R. Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software*

Engineering 30, 6 (June 2004), 418–421. doi:10.1109/TSE.2004.24

[48] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Hess, Sebastian Krieter, and Gunter Saake. 2025. How Configurable Is the Linux Kernel? Analyzing Two Decades of Feature-Model History. *ACM Transactions on Software Engineering and Methodology* (April 2025). doi:10.1145/3729423

[49] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 1–18. doi:10.1109/TDSC.2024.3354789

[50] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 105–114. doi:10.1145/1806799.1806819

[51] Jörg Liebig, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, New York, NY, USA, 81–91. doi:10.1145/2491411.2491437

[52] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM '05, Vol. 14)*. USENIX Association, USA, 18. https://dl.acm.org/doi/10.5555/1251398.1251416

[53] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. 2013. Practical Pairwise Testing for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC '13)*. ACM, New York, NY, USA, 227–235. doi:10.1145/2491627.2491646

[54] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 643–654. doi:10.1145/2884781.2884793

[55] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Larissa Braz, Christian Kästner, Sven Apel, and Kleber Santos. 2020. An Empirical Study on Configuration-Related Code Weaknesses. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering (SBES '20)*. ACM, New York, NY, USA, 193–202. doi:10.1145/3422392.3422409

[56] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. 2015. An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '15)*. ACM, New York, NY, USA, 35–44. doi:10.1145/2814204.2814206

[57] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. ACM, New York, NY, USA, 50–61. doi:10.1145/3338906.3338967

[58] Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade, Baldoino Fonseca, and Márcio Ribeiro. 2018. A Qualitative Analysis of Variability Weaknesses in Configurable Systems with #ifdefs. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '18)*. ACM, New York, NY, USA, 51–58. doi:10.1145/3168365.3168382

[59] Jeho Oh, Don Batory, Marijn Heule, Margaret Myers, and Paul Gazzillo. 2019. *Uniform Sampling from Kconfig Feature Models*. Technical Report 19. The University of Texas at Austin, Department of Computer Science, Austin, Texas.

[60] Zachery Patterson. 2023. *Toward Applying Variability-Oblivious Static Analyses to Software Product Lines*. Ph. D. Dissertation. The University of Texas at Dallas, Dallas. https://utd-ir.tdl.org/items/1623bed4-684c-44e7-94c2-f20cfeb7c976

[61] Zachary Patterson, Zenong Zhang, Brent Pappas, Shiyi Wei, and Paul Gazzillo. 2022. SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 2056–2067. doi:10.1145/3510003.3512763

[62] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE, USA, 459–468. doi:10.1109/ICST.2010.43

[63] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, New York, NY, USA, 26–36. doi:10.1145/2491411.2491436

[64] Tobias Pett, Tobias Heß, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2023. Continuous T-Wise Coverage. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23)*. ACM, New York, NY, USA, 87–98. doi:10.1145/3579027.3608980

[65] Tobias Pett, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2024. MulTi-Wise Sampling: Trading Uniform T-Wise Feature Interaction Coverage for Smaller Samples. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference (SPLC '24)*. ACM, New York, NY, USA, 47–53. doi:10.1145/3646548.3672589

[66] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A (SPLC '19)*. ACM, New York, NY, USA, 78–83. doi:10.1145/3336294.3336322

[67] Mohd Zanes Sahid, Abu Bakar Md Sultan, Abdul Azim Abdul Ghani, and Salmi Baharom. 2016. Combinatorial Interaction Testing of Software Product Lines: A Mapping Study. *Journal of Computer Science* 12, 8 (Aug. 2016), 379–398. doi:10.3844/jcssp.2016.379.398

[68] Patrick Schober, Christa Boer, and Lothar A. Schwarte. 2018. Correlation Coefficients: Appropriate Use and Interpretation. *Anesthesia & Analgesia* 126, 5 (May 2018), 1763–1768. doi:10.1213/ANE.0000000000002864

[69] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. 2022. Static Data-Flow Analysis for Software Product Lines in C: Revoking the Preprocessor's Special Role. *Automated Software Engineering* 29, 35, Article 35 (May 2022), 37 pages. doi:10.1007/s10515-022-00333-1

[70] Sandro Schulze, Oliver Richers, and Ina Schaefer. 2013. Refactoring Delta-Oriented Software Product Lines. In *Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development (AOSD '13)*. ACM, New York, NY, USA, 73–84. doi:10.1145/2451436.2451446

[71] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM '01, Vol. 10)*. USENIX Association, USA, Article 16, 17 pages. https://www.usenix.org/conference/10th-usenix-security-symposium/detecting-format-string-vulnerabilities-type-qualifiers

[72] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review* 45, 3 (Jan. 2012), 10–14. doi:10.1145/2094091.2094095

[73] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (July 2014), 1–45. doi:10.1145/2580950

[74] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE '12)*. ACM, New York, NY, USA, 11–20. doi:10.1145/2371401.2371404

[75] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18)*. ACM, New York, NY, USA, 1–13. doi:10.1145/3233027.3233035

[76] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 27, 4 (Oct. 2018), 1–33. doi:10.1145/3280986

[77] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability Encoding: From Compile-Time to Load-Time Variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (Jan. 2016), 125–145. doi:10.1016/j.jlamp.2015.06.007

[78] Fabian Yamaguchi. 2015. *Pattern-Based Vulnerability Discovery*. Ph. D. Dissertation. Georg-August-University, Göttingen. doi:10.53846/goediss-5356

[79] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy (S&P '14)*. IEEE, USA, 590–604. doi:10.1109/SP.2014.44

[80] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy (S&P '15)*. IEEE, USA, 797–812. doi:10.1109/SP.2015.54

[81] Li Zhou, Minhuan Huang, Yujun Li, Yuanping Nie, Jin Li, and Yiwei Liu. 2021. GraphEye: A Novel Solution for Detecting Vulnerable Functions Based on Graph Attention Network. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC '21)*. IEEE, USA, 381–388. doi:10.1109/DSC53577.2021.00060