

Family-Based Vulnerability Discovery for Highly-Configurable Software Systems

Tim Bächle^[0009-0006-8951-4244], Erik Hofmayer^[0009-0002-6486-9584], and
Tobias Pett^[0000-0001-7652-6525]

KASTEL - Institute of Information Security and Dependability,
Karlsruhe Institute of Technology, Karlsruhe, Germany
`{tim.baechle,tobias.pett}@kit.edu, erik.hofmayer@student.kit.edu`

Abstract. Discoveries such as the infamous *Heartbleed* vulnerability highlight the importance of analyzing configurable software for the presence of vulnerabilities. Static source code analysis offers two prominent solutions for configurable software: *product-based* and *family-based* strategies. Previous research compared the capabilities of these strategies for the discovery of bugs and found that family-based approaches tend to be more effective and efficient. While approaches for product-based vulnerability discovery exist, there currently is no corresponding family-based approach. In this paper, we aim to fill this gap. We extend the analysis platform VARI-JOERN, offering product-based vulnerability discovery for configurable software systems, with a family-based approach. Using our extended version of VARI-JOERN, we compare product- and family-based vulnerability discovery in terms of their effectiveness and efficiency when applied to three real-world, highly-configurable software systems. Our results indicate that limitations of state-of-the-art variability encoding, along with challenges that arise from analyzing variability-encoded source code, prevent the family-based approach from achieving effectiveness comparable to the product-based alternative. Nonetheless, its efficiency remains comparable to the product-based analysis. Overall, our work offers insights into the potential of family-based vulnerability discovery and the open issues that remain.

Keywords: Vulnerability Discovery · Configurable Software Systems · Software Product Lines · Static Analysis · Family-Based · Product-Based

1 Introduction

Software *vulnerabilities*, i.e., weaknesses in source code that can be exploited by one or more threats [29], can allow malicious users to inflict various types of harm [21, 41, 64]. The resulting risk is further increased by the widespread adoption of *configurable software systems* [3, 19], i.e., software characterized by variable *features* that can be selected to create *variants* for different use cases [17, 25, 61]. Configurable software systems have become popular in many domains and are increasingly applied in safety- and mission-critical application areas [3,

6, 19, 50]. Crucially, vulnerabilities in these systems can be *variability-induced*, meaning that they are not always present but tied to specific *configurations* (i.e., feature selections) [8, 22, 43]. A prominent example is the *Heartbleed* vulnerability in OPENSSL [47], which allowed attackers to access sensitive user data [26, 63] only if a particular feature was enabled [22, 61]. Considering the enormous set of possible configurations of modern configurable software systems [60], identifying variability-induced vulnerabilities poses a serious challenge [8, 43].

Static source code analysis represents a popular method for vulnerability discovery [15, 20, 21, 41, 56]. However, static analysis techniques typically cannot cope with the variability found inside configurable software systems [10, 23, 40, 48, 49, 53, 60]. As a result, two analysis strategies are typically distinguished [6, 40, 57]. A *product-based* analysis derives variants from a configurable system that are analyzed individually using conventional (i.e., variability-oblivious) analysis techniques [57]. As the number of variants that can be derived from a configurable system can grow exponentially with the number of its features [6, 19, 40, 43], this strategy is often augmented with sampling techniques that limit the analysis to a representative subset of variants [51, 57, 60]. Instead of considering individual variants, a *family-based* analysis analyzes the entirety of a configurable system in a single pass [6, 28]. This is achieved by analyzing the system’s implementation artifacts still containing variability [19, 40] using either dedicated (i.e., variability-aware) analysis techniques or *variability encoding*, which transforms variable source code into a form accepted by conventional analysis tooling [28, 61].

Previous research [5, 6, 40, 49, 60] compared product- and family-based bug discovery and found that a family-based strategy tends to be more effective (i.e., capable of identifying more bugs) and more efficient (i.e., takes less time) than a product-based strategy using sampling. Whether this only applies to bug discovery or whether developers should also be advised to follow a family-based strategy for the identification of vulnerabilities in configurable systems is unknown. While product-based vulnerability discovery approaches have been proposed [8, 43], to the best of our knowledge, there is no family-based counterpart.

This paper aims to fill this gap by proposing a family-based vulnerability discovery approach. To this end, we extend the VARI-JOERN¹ platform of Bächle et al. [8] with a family-based analysis strategy built on the idea of *lifting by simulation* [28]. Our approach applies the variability encoding solution SUGARC [48] to variable C source code, transforming compile-time variability (i.e., preprocessor conditionals) into runtime variability (i.e., if-statements) [5, 61]. The transformed source code simulates the behavior of the possible software variants and is analyzed for potential vulnerabilities using the static analysis tool JOERN [30, 62].

We evaluate the developed family-based vulnerability discovery approach by comparing it to the product-based alternative available in VARI-JOERN [8]. For the product-based approach, we employ the popular sampling techniques of *t-wise Feature Interaction Sampling (t-wise FIS)* and *Uniform Random Sampling (URS)*. We analyze the three real-world, highly-configurable subject systems AXTLS [7], TOYBOX [59], and BUSYBOX [13] for potential vulnerabilities. Com-

¹ <https://github.com/KIT-TVA/Vari-Joern>.

paring the potential vulnerabilities across the subject systems, we assess the effectiveness of our family-based approach. To evaluate its efficiency, we compare the time required for a complete analysis of the individual subject systems.

Our results show that although our family-based approach exhibits effectiveness comparable to the product-based alternative on AXTLS, limitations of state-of-the-art variability encoding, along with challenges that arise from analyzing variability-encoded source code, prevent it from achieving similar effectiveness on more complex subject systems. Nevertheless, it remains largely comparable to the product-based approach with regard to efficiency.

In summary, we make the following contributions:

- We extend the analysis platform VARI-JOERN with a family-based analysis approach for vulnerability discovery in highly-configurable software.
- We evaluate the effectiveness and efficiency of our family-based approach on real-world software systems by comparing it to a product-based alternative.
- We make our extended version of VARI-JOERN² as well as our evaluation scripts and data³ publicly available.

Significance. While our findings suggest that the benefits of a family-based strategy observed for bug discovery do not necessarily hold for vulnerability discovery, they show that this is not a conceptual issue inherent to the family-based analysis strategy. The observed shortcomings are instead caused by limitations of current variability encoding solutions and the challenges that arise when analyzing variability-encoded source code. As a result, our work establishes a foundation for future research and offers guidance for both researchers and practitioners aiming to increase the security of highly-configurable software systems.

2 Background

In this section, we outline fundamental concepts of highly-configurable software systems (Section 2.1). In addition, we detail product- and family-based strategies commonly used for the analysis of configurable software (Section 2.2).

2.1 Highly-Configurable Software Systems

Highly-configurable software systems have become common across many domains [3, 19] and are typically realized as *Software Product Lines (SPLs)*. An SPL represents a collection of similar software products that are distinguished by the end-user-visible increments in functionality (i.e., *features* [5, 14, 45, 54]) they provide [6, 53, 57]. Individual software products are referred to as *variants* and specified by a particular *configuration* (i.e., a selection) of features [2, 19, 32, 61]. Considering that not all configurations necessarily describe meaningful variants [58], a *feature model* often describes the features and the relationships that

² <https://doi.org/10.5281/zenodo.19486000>.

³ <https://doi.org/10.5281/zenodo.19485457>.

```

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

```

Listing 1: A configurable C function adapted from the examples provided by Yamaguchi et al. [62] and Bächle et al. [8]

exist between them [2, 3, 9, 32, 51]. A textual way to express feature models is through KCONFIG [1, 40, 45], which has become a popular tool for variability management in system software [2, 40, 45, 46].

In practice, highly-configurable software systems are usually implemented following an *annotative implementation strategy* [2, 4, 12, 19]. This means that the source code of all variants is merged into a common code base [3]. Code fragments related to certain features or combinations thereof are annotated accordingly [3, 33]. Concrete variants can then be derived from the system by removing all code fragments not related to the chosen feature configuration [3, 14, 33]. From a technical point of view, an annotative implementation strategy is typically realized using preprocessor directives [22, 35, 40, 43, 53, 55]. An example is shown in Listing 1. In the `foo` function, an `ifdef` (i.e., a conditional-inclusion directive) surrounds the source code in Line 5 and ties it to a selection of the `SEND_DATA` feature. Consequently, when preprocessing the source code before compilation, Line 5 remains in the source code only if the `SEND_DATA` feature is selected.

2.2 Analysis Strategies for Configurable Software

Analysis techniques act as important means to ensure a system’s correctness, reliability, and security [6]. However, standard analysis techniques usually cannot cope with a configurable software system’s variability [10, 23, 40, 48, 49, 53, 60]. As a result, dedicated analysis strategies have emerged [57]. Two strategies that are often distinguished are *product-based* and *family-based* analyses [6, 40, 60].

Product-Based Analysis. The goal of a *product-based* analysis is to derive concrete variants of a configurable system that can then be analyzed individually [57]. As deriving concrete variants resolves the variability of the configurable system, this enables the application of conventional (i.e., *variability-oblivious*) analysis techniques [57]. Depending on whether all possible variants are considered, *exhaustive* and *optimized* product-based strategies can be distinguished [57]. Especially for real-world configurable systems, analyzing all possible variants (i.e., following an exhaustive strategy) quickly becomes infeasible, as this number can grow exponentially with the number of features [10, 16, 40, 43, 57, 60]. For this reason, optimized product-based strategies that use sampling techniques

to select a representative subset of variants for the analysis have become the de facto standard [40, 60]. In this regard, two popular sampling techniques are *t-wise Feature Interaction Sampling (t-wise FIS)* and *Uniform Random Sampling (URS)*. *T-wise FIS* aims at creating a sample such that all valid configurations between *t*-tuples of features appear in at least one sampled variant [16]. Contrary, *URS* samples variants across the configuration space of the configurable system with an equal (i.e., uniform) probability [27, 50].

Family-Based Analysis. Instead of focusing on individual variants, a *family-based* analysis aims to analyze a configurable system as a whole [6, 28, 40]. This means that code artifacts are analyzed before variant derivation and thus while still containing variability [40]. The benefit is that code shared across multiple variants is analyzed only once [40, 57, 60]. Furthermore, since the code base is analyzed in full, the inherent incompleteness of optimized product-based strategies can be avoided [60]. Two methods how family-based analyses can be realized are typically distinguished [28, 61]: (1) developing novel family-based analysis solutions or altering off-the-shelf tooling for the operation on variable source code [40, 57], and (2) applying *lifting by simulation* [28]. Lifting by simulation refers to the application of *variability encoding*, a technique for transforming configurable software into a so-called *variant simulator*, i.e., a single variant able to simulate all derivable variants using only runtime variability [5, 28, 57, 61]. Analysis tooling is then applied to the variant simulator rather than to the original source code [28]. While the first method is intuitive, it requires expert knowledge to avoid errors and can be very labor-intensive [24, 28, 55, 61]. The second method requires a variability encoding solution but offers the advantage that off-the-shelf tooling can be used for the analysis of the variant simulator [28, 48, 61].

3 Vulnerability Discovery for Configurable Software

While vulnerability discovery approaches based on a product-based strategy have already been proposed [8, 43], there currently is no approach leveraging a family-based strategy. Consequently, we devised our own family-based approach. To avoid the overhead and error-proneness of engineering a novel solution [60], we opted to realize our approach using lifting by simulation [28]. Furthermore, we decided to integrate our approach into a solution from the literature. Specifically, we chose the VARI-JOERN analysis platform of Bächle et al. [8] due to its extensible design and support for an optimized product-based strategy that we could leverage for the evaluation of our family-based approach. The structure of the extended version of VARI-JOERN is shown in Figure 1 and detailed below. Section 3.1 provides a summary of the existing product-based analysis, whereas Section 3.2 details our family-based extension using lifting by simulation.

3.1 Product-Based Vulnerability Discovery

The initial version of VARI-JOERN proposed by Bächle et al. [8] implements an optimized product-based strategy (cf. Section 2.2). Its analysis pipeline is

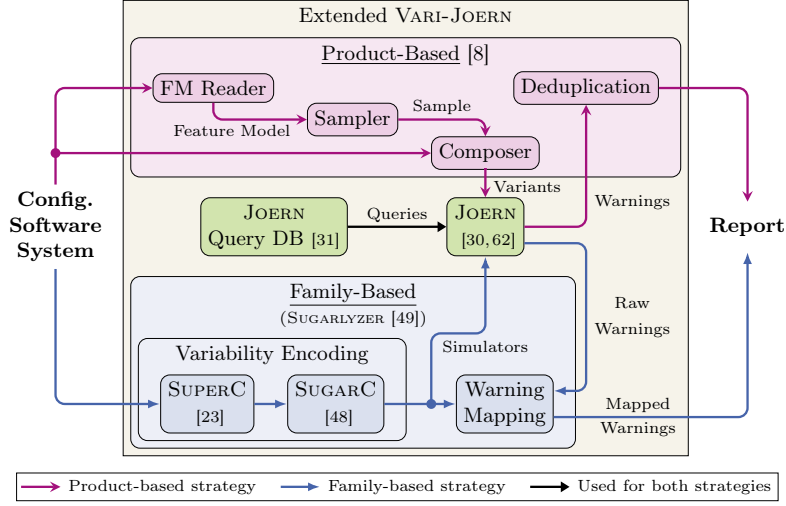


Figure 1: Structure of the extended version of VARI-JOERN

split into five steps, as shown in the upper half of Figure 1. Initially, a *Feature Model Reader* extracts the feature model described by the KCONFIG files of the configurable system that should be analyzed. Using the extracted information, a *Sampler* decides on a set of system configurations to analyze. For this task, VARI-JOERN currently supports the popular sampling techniques of *t*-wise FIS (realized through the YASA sampling algorithm of Krieter et al. [38]) and URS (realized through SMARCH of Oh et al. [45]). A *Composer* takes the sampled configurations and generates corresponding software variants that only contain the source code relevant to the sampled configurations. These variants are handed to the static analysis tool JOERN [30, 62], which reports potential vulnerabilities per variant. To avoid duplicates caused by findings affecting multiple variants, the reported warnings undergo *Deduplication* before being presented to the user.

3.2 Family-Based Vulnerability Discovery

Our family-based approach is based on lifting by simulation [28]. Below, we describe the underlying concept and its realization inside VARI-JOERN.

Concept. Lifting by simulation applies variability encoding to variable source code such that an off-the-shelf tool can be used for the analysis of the resulting product simulator (cf. Section 2.2). For our purpose, this involves three steps:

Step 1 creates a variant simulator by applying variability encoding to the variable source code. In the case of C, the target language of VARI-JOERN [8], compile-time variability (i.e., conditional-inclusion directives) is transformed into runtime variability (i.e., if-statements). Listing 2 exemplifies this procedure in the context of the `foo` function (cf. Listing 1). As the `ifdef` in Line 4 of Listing 2a wraps an entire statement, it can be substituted with an if-statement over

<pre> 1 void foo() { 2 int x = source(); 3 if(x < MAX) { 4 #ifdef CONFIG_SEND_DATA 5 sink(x); 6 #endif 7 // ... 8 } 9 }</pre>	<pre> 1 bool sendData; // Init. nondeterministically. 2 void foo() { 3 int x = source(); 4 if(x < MAX) { 5 if(sendData) { 6 sink(x); 7 } 8 // ... 9 } 10 }</pre>
--	---

(a) Before variability encoding
(b) After variability encoding

Listing 2: Variability encoding of the `foo` function (cf. Listing 1)

a boolean feature variable. The corresponding feature variable in Line 1 of Listing 2b is initialized nondeterministically. This prevents static analysis tools from inferring that bodies of if-statements introduced through variability encoding are dead code, potentially excluding them from analysis [5, 58]. While variability encoding of toy examples is straightforward, transforming real-world C represents a serious challenge that has been the subject of extensive research [28, 48, 61].

Step 2 takes the variant simulator and analyzes it using off-the-shelf vulnerability discovery tools. Since the variant simulator is free of preprocessor-based variability and contains only conventional C source code, existing analysis tools can operate on it without modification [28, 48, 61]. However, as variability encoding can severely alter source code structure [28, 48], the vulnerability warnings reported by the analysis tool can relate to completely different code regions.

Step 3 addresses this issue by mapping the warnings reported by the analysis tool to their corresponding location in the original source code.

Realization. The realization of our family-based vulnerability discovery approach is illustrated in the lower half of Figure 1. We implement lifting by simulation using the SUGARLYZER framework of Patterson [49]. In essence, SUGARLYZER aims to automate the process of applying variability-oblivious analysis tools to variant simulators of a configurable system’s individual source files [49]. It therefore realizes the three steps outlined in the aforementioned concept on a per-file basis. SUGARLYZER uses SUGARC [48], a state-of-the-art variability encoding solution built on the variability-aware C parser SUPERC [23]. Furthermore, it provides a mapping component that uses specially formatted comments inserted into the variant simulators by SUGARC to map findings reported by an analysis tool back to their original source code location. This mapping component also performs deduplication, considering that the code duplication typically introduced through variability encoding [28, 61] can lead to duplicate findings.

As the SUGARLYZER framework was intended for bug discovery inside configurable software systems [49], it only supported common bug finding tools out of the box. Considering the potential the static analysis tool JOERN [30, 62] has for vulnerability discovery [8], we added support for JOERN. In addition, we integrated the framework into Bächle et al.’s analysis platform VARI-JOERN [8] (cf.

Figure 1) and altered the command-line interface such that switching between our family-based approach and the existing product-based strategy is possible.

4 Evaluation

For the evaluation of our presented family-based vulnerability discovery approach (cf. Section 3.2), we perform a comparison to VARI-JOERN’s optimized product-based strategy (cf. Section 3.1). We opt for this evaluation method, as optimized product-based approaches represent the de facto standard for analyzing configurable software [40]. Furthermore, it reflects common practice in studies on family-based analysis approaches [5, 6, 40, 49, 60]. Since VARI-JOERN’s optimized product-based strategy is configurable and offers support for both t -wise FIS and URS, we first perform a comparison between the available sampling techniques. Consequently, we first aim at answering the research question RQ_1 :

RQ_1 : How do t -wise FIS and URS compare when used for vulnerability discovery inside highly-configurable software systems?

Leveraging the findings for RQ_1 , we settle on the most promising sampling technique for the comparison to our family-based approach. In this regard, we aim at answering the second research question RQ_2 :

RQ_2 : How does the presented family-based vulnerability discovery approach compare to a product-based alternative in terms of the potential vulnerabilities detected?

Even the most effective vulnerability discovery approach becomes infeasible if it incurs impractical resource demands. Accordingly, we also consider the resource demand of our approach, as expressed in a third research question RQ_3 :

RQ_3 : How does the presented family-based vulnerability discovery approach compare to a product-based alternative in terms of execution time for a full analysis?

4.1 Subject Systems

We selected the three real world, highly-configurable subject systems AXTLS [7], TOYBOX [59], and BUSYBOX [13] for the evaluation. We chose these systems as their implementation is open source, they encompass a significant share of C code, and they rely on KCONFIG [34] for their configuration management.⁴ Additionally, all systems served as subject systems in recent studies on the analysis of configurable software [8, 44, 45, 48, 50] and thus act as popular representatives. The systems are further characterized in Table 1. Besides their name, version, and type, we also capture the number of lines of code and the number of features as rough estimates of the systems’ complexity. For all systems, we used the

⁴ All three characteristics are currently required for an analysis with VARI-JOERN [8].

Table 1: The set of subject systems used in our evaluation

Name	Version	Type	C-LoC	#Features
AXTLS [7]	2.1.5	Client/server TLS library	17,556	63
TOYBOX [59]	0.8.12	Collection of Linux command line utilities	61,324	350
BUSYBOX [13]	1.36.1	Collection of UNIX utilities	182,966	1,027

latest stable release as of September 2025. The C-LoC column shows the number of lines of code found in C source files as measured using the tool CLOC,⁵ whereas the #Features column shows the number of features found in the feature models extracted from the systems by VARI-JOERN’s feature model reader (cf. Section 3.1). The complexity of the systems in terms of lines of code and features more than triples from AXTLS to TOYBOX. A similar increase in complexity can be observed between TOYBOX and BUSYBOX.

4.2 Experimental Setup

We adopt a quantitative study focusing on the reported vulnerability warnings and the required execution time for a full analysis of the subject systems for answering our research questions. The setup for the individual research questions and the used execution environment is detailed below.

RQ1. To assess how the sampling techniques of t -wise FIS and URS compare for vulnerability discovery, we analyze all subject systems (cf. Section 4.1) using VARI-JOERN’s optimized product-based strategy (cf. Section 3.1) and the respective sampling technique. For t -wise FIS, we use the sampling strengths $t \in \{1, 2, 3\}$. We decided against using higher sampling strengths, as these quickly result in unreasonably high computational costs for real-world systems [42, 51], such as those selected for the evaluation. Since URS lacks a clear stopping criterion and can thus be run indefinitely, we employ two target sample sizes for the corresponding analysis: (1) the average sample size obtained by 3-wise FIS (N), and (2) this average multiplied by a factor of 3 ($N \times 3$). For each analysis run, we collect the total number of vulnerability warnings reported by VARI-JOERN that are not related to the analyzed system’s infrastructure (KCONFIG files, unit tests, etc.). In addition, we measure the execution time using the tool GNU TIME⁶ without taking advantage of parallel composer and analyzer components (cf. Section 3.1). To account for outliers in the number of findings and execution time, we repeat all analysis runs ten times. We then compare the sampling techniques in terms of their distribution of raised vulnerability warnings across executions and their average execution time for a full analysis.

⁵ <https://github.com/AlDanial/cloc> (version 2.06).

⁶ <https://www.gnu.org/software/time> (version 1.9).

RQ2. For evaluating the vulnerability discovery capabilities of our family-based approach (cf. Section 3.2), we follow a similar setup to RQ_1 . We use VARI-JOERN to analyze all subject systems using both the optimized product-based approach and our family-based approach. For the optimized product-based analysis, which we use as ground truth, we focus on the sampling technique identified as most promising through RQ_1 to reduce the scope of the comparison. In this regard, we reuse the analysis results collected for answering RQ_1 . We then use both analysis strategies’ results to categorize the raised (non-infrastructure-related) vulnerability warnings according to whether they were raised by both analysis strategies (Type 1), raised only using the optimized product-based strategy (Type 2), or raised only by the family-based strategy (Type 3). As the sampling techniques employed by VARI-JOERN’s optimized product-based analysis are not deterministic in their execution, we repeat this process ten times and accumulate the results. To reason about the instances raised only by one of the strategies, we manually examine a sample of corresponding vulnerability warnings.

RQ3. To evaluate the resource demand of our family-based vulnerability discovery approach (cf. Section 3.2), we combine aspects of the setups of RQ_1 and RQ_2 . We analyze the subject systems using VARI-JOERN’s optimized product-based strategy (leveraging the sampling technique found most promising through RQ_1) and our family-based approach. For the optimized product-based analysis, we reuse the individual time measurements from RQ_1 . For the family-based analysis, we measure the execution time of the ten analysis runs conducted for RQ_2 . In this regard, we again disable any explicit parallelization features of VARI-JOERN. Similar to RQ_1 , we then determine and compare the average execution times between the optimized product-based and family-based analysis.

Execution Environment. For all of our experiments, we execute VARI-JOERN within its provided DOCKER container. We run this container on a machine equipped with a 16-core AMD RYZEN THREADRIPPER PRO 5955WX processor and 128 GB RAM, running UBUNTU 24.04.3 LTS.

4.3 T-Wise Interaction Sampling vs. Uniform Random Sampling

For addressing RQ_1 , we compare t -wise FIS and URS with regard to the raised vulnerability warnings and the required execution time (cf. Section 4.2).

Results. Figure 2 depicts box plots of the vulnerability warnings reported by VARI-JOERN for the three subject systems using t -wise FIS and URS. For AXTLS, all sampling techniques except 1-wise FIS always yielded 102 warnings. Similarly, the analysis of TOYBOX consistently produced 685 warnings. For BUSYBOX, the number of vulnerability warnings grew with increasing sampling strength (i.e., t -values) of t -wise FIS, albeit at a rate that quickly declined from $t = 2$ to $t = 3$. Additionally, URS resulted in fewer warnings than 3-wise FIS,

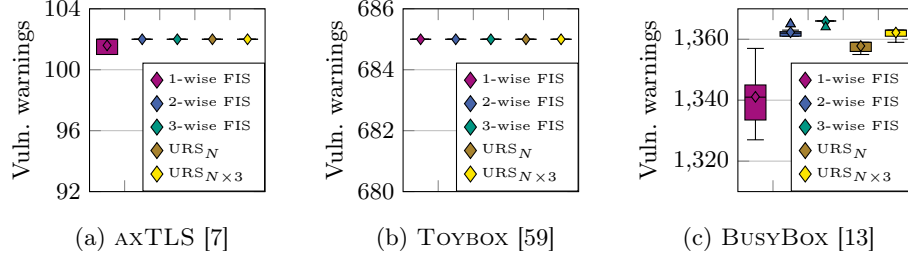


Figure 2: Distribution of vulnerability warnings raised using t -wise Feature Interaction Sampling (t -wise FIS) and Uniform Random Sampling (URS)

Table 2: Average execution time (*hh:mm:ss*) for a full analysis using t -wise Feature Interaction Sampling (t -wise FIS) and Uniform Random Sampling (URS)

System	1-wise FIS	2-wise FIS	3-wise FIS	URS _N	URS _{N×3}
AXTLS [7]	00:01:40	00:05:44	00:22:45	00:23:14	01:12:04
TOYBOX [59]	00:04:17	00:14:10	00:57:53	00:53:11	02:39:34
BUSYBOX [13]	00:16:53	01:37:24	10:11:14	10:30:18	31:38:00

independent of the considered target sample size, and only approached the numbers of vulnerability warnings produced by 2-wise FIS when the larger target sample size ($N \times 3$) was used. Nonetheless, even using the smaller target sample size (N), URS generally produced noticeably more warnings than 1-wise FIS.

Considering the analysis effort, Table 2 reports the average execution time across the subject systems and sampling techniques. Increasing the sampling strength of t -wise FIS always resulted in a significant increase in execution time. URS using the smaller target sample size (N) yielded an execution time slightly slower than 3-wise FIS for both AXTLS and BUSYBOX, while being marginally faster than 3-wise FIS on TOYBOX. Using the larger target sample size ($N \times 3$) for URS roughly triples the average execution time across all subject systems compared to the smaller target sample size (N). Independent of the sampling technique, the execution time was lowest for AXTLS and highest for BUSYBOX.

Discussion. Our results show that the sampling techniques of t -wise FIS and URS yield the same number of vulnerability warnings on AXTLS and TOYBOX in all but a single case (cf. Figures 2a and 2b). While both subject systems are highly-configurable and of considerable size, compared to other configurable software systems (e.g., BUSYBOX), they exhibit only a fraction of the complexity (cf. Table 1). It is therefore to be expected that only minor differences will emerge between sampling techniques. Furthermore, in the case of TOYBOX, much of the system’s variability is found on file granularity and tied to single features. Such variability patterns can easily be covered by even simple sampling techniques (e.g., 1-wise FIS). This explains why the results for TOYBOX show less variation

between sampling techniques than those for AXTLS (cf. Figures 2a and 2b), despite the system being more than three times as large and configurable (cf. Table 1). Contrary to AXTLS and TOYBOX, the results for BUSYBOX show noticeable differences between sampling techniques. For the different sampling strengths of t -wise FIS, we notice the same near-logarithmic relationship between the parameter t and the number of findings as indicated in earlier studies [8, 25, 42]. Despite using a target sample size equal (or multiple in the case of $URS_{N \times 3}$) to the average sample size yielded by 3-wise FIS, URS only regularly outperforms t -wise FIS in the number of raised vulnerability warnings for a sampling strength of $t = 1$ (i.e., 1-wise FIS). This is likely caused by the fact that URS does not aim to cover feature interactions. Consequently, so-called *feature-interaction vulnerabilities* (i.e., variability-induced vulnerabilities whose presence depends on the interplay of multiple features’ source code [8]) may frequently be missed.

Considering the total execution time across sampling techniques (cf. Table 2), the analysis effort grows nearly exponentially with the t -wise FIS strength t . This is consistent with previous observations showing that the sample size (and thus the number of software variants that need to be analyzed) increases exponentially with the sampling strength [8, 25, 36, 37, 42]. Furthermore, independent of the subject system, the analysis using URS and the target sample size N exhibits an execution time comparable to the time using 3-wise FIS. As N is equal to the average sample size yielded by 3-wise FIS, this suggests that the resource demand for the sampling of configurations does not differ substantially between sampling techniques. Lastly, as expected, tripling the target sample size used for URS (i.e., using $N \times 3$) approximately triples the average execution time across all three subject systems. In summary, we can conclude RQ_1 as follows:

Conclusion RQ_1 : URS generally outperforms 1-wise FIS in the number of raised vulnerability warnings. However, to achieve a comparable number of vulnerability warnings to even 2-wise FIS, it requires a substantially larger target sample size. Consequently, the corresponding analysis is considerably more resource-demanding than all considered parameterizations of t -wise FIS, even vastly exceeding the execution time for 3-wise FIS. Our results therefore indicate that t -wise FIS is more suitable for vulnerability discovery in highly-configurable software systems than URS.

4.4 Family-Based vs. Product-Based Vulnerability Discovery

Our experimental results indicate that t -wise FIS is more suitable for vulnerability discovery than URS (cf. Section 4.3). For answering RQ_2 and RQ_3 , we thus focus on a comparison between our family-based vulnerability discovery approach and VARI-JOERN’s optimized product-based strategy using t -wise FIS (cf. Section 3). We concentrate on 2-wise and 3-wise FIS for the optimized product-based strategy since the results for RQ_1 showed that 1-wise FIS performed the worst among all considered sampling techniques (cf. Figure 2).

Results. Table 3 shows the aggregated shares of vulnerability warnings with regard to three types: (1) those raised by both analysis strategies, (2) those

Table 3: Aggregated shares (and numbers) of vulnerability warnings reported by both analysis strategies (Type 1), only by the optimized product-based strategy (Type 2), and only by our family-based strategy (Type 3)

System	t	Share Type 1 (Both)	Share Type 2 (P-Based)	Share Type 3 (F-Based)	Total
AXTLS [7]	2, 3	95.28% (1010)	0.94% (10)	3.77% (40)	1060
TOYBOX [59]	2, 3	78.83% (5770)	14.75% (1080)	6.42% (470)	7320
BUSYBOX [13]	2	34.59% (4991)	59.82% (8632)	5.60% (808)	14431
	3	34.76% (5019)	59.83% (8639)	5.41% (781)	14439

raised only using the optimized product-based strategy, and (3) those raised only using our family-based approach. For AXTLS, independent of the FIS strength t used for the comparison, the vast majority of warnings are of Type 1 (Both), with only small shares belonging to Type 2 (P-Based) and Type 3 (F-Based). While the results for TOYBOX are similar, the shares of Type 2 (P-Based) and Type 3 (F-Based) are noticeably larger than for AXTLS, decreasing the share of Type 1 (Both). This trend extends partially to BUSYBOX. There, the share of Type 2 (P-Based) is substantially larger than for TOYBOX and accounts for the majority of vulnerability warnings. At the same time, the share of Type 1 (Both) decreases significantly, whereas the share of Type 3 (F-Based) decreases only slightly compared to TOYBOX. For BUSYBOX, increasing the FIS strength t used for the comparison slightly increases the shares for Type 1 (Both) and Type 2 (P-Based). It also slightly decreases the share for Type 3 (F-Based).

Using our family-based approach, we measured average execution times for a full analysis of 00:07:20 (*hh:mm:ss*), 01:57:32, and 03:46:57 for AXTLS, TOYBOX, and BUSYBOX, respectively. Considering the average execution times of the optimized product-based strategy (cf. Table 2), our family-based approach is thus positioned between the optimized product-based analysis using 2-wise and 3-wise FIS for AXTLS and BUSYBOX. For TOYBOX, it is more resource-intensive than even the analysis using 3-wise FIS, requiring more than twice the execution time.

Discussion. For assessing the vulnerability discovery capabilities of our family-based approach, we examine the classification in Table 3. With the optimized product-based analysis serving as ground truth, Type 1 (Both) of the classification can be interpreted as true positives, Type 2 (P-Based) as false negatives, and Type 3 (F-Based) as false positives. The large share of Type 1 (Both) for AXTLS suggests that both analysis strategies produce similar sets of vulnerability warnings. This finding does not extend to TOYBOX and BUSYBOX, where

we observe a drastic increase in the share of Type 2 (P-Based) vulnerability warnings (i.e., false negatives). This behavior is not affected by the FIS strength t of the ground truth, since it only affects BUSYBOX and results in changes below 1%. To reason about the causes of the massive increase in false negatives for TOYBOX and BUSYBOX, we manually inspected a sample of corresponding Type 2 (P-Based) vulnerability warnings and found that they could be explained by a number of recurring patterns that fall into two categories: (1) limitations of variability-encoding solutions and (2) challenges of analyzing variability-encoded source code. We describe these categories in more detail below.

Limitations of Variability-Encoding Solutions. Despite multiple previous efforts [28, 48, 61], performing variability-encoding of real-world C code remains a challenging task. Not only is it inherently difficult to support all language constructs of a general-purpose programming language, such as C, but it also represents a non-trivial task to ensure that the transformation ensures behavior preservation [28, 48, 61]. In addition, with variability encoding typically realized through systematic code duplication and renaming, transforming variable source code can be very computationally demanding [28, 48, 61]. Considering these challenges, it is unsurprising that even a state-of-the-art tool such as SUGARC [48], which we use in our family-based approach (cf. Section 3.2), faces limitations. For TOYBOX and BUSYBOX, we observed that certain code regions (e.g., specific conditional blocks or functions) may not always be correctly transformed. Furthermore, variability encoding of some source files (2 for TOYBOX and 7 for BUSYBOX) regularly exceeds ten minutes, which the family-based approach uses as a timeout to avoid excessively long analysis runs. Consequently, vulnerability warnings that should have been raised in source files or code regions that could not be correctly variability encoded are missed. Another behavior that we observed is that the line number comments emitted into a variant simulator by SUGARC to enable a mapping to the original source code locations [48] may be missing or contain imprecise information. This creates two issues. First, if mappings are missing, not every vulnerability warning raised on a variant simulator can be traced back to its corresponding location within the original source code, leading to missed vulnerability warnings. Second, if mappings are imprecise, it becomes effectively impossible to accurately match vulnerability warnings between our family-based approach and the optimized product-based ground truth as required for Type 1 (Both) of our categorization (cf. Table 3). We first recognized the limitations of SUGARC during the implementation of our approach. In response, we investigated their causes and already resolved some of the issues. However, as the underlying reasons are conceptually complex, fully resolving the limitations of current state-of-the-art variability encoding solutions represents a challenge that calls for a collaborative effort by the community.

Challenges of Analyzing Variability-Encoded Source Code. The code duplication and renaming typically used to realize variability encoding can severely alter the structure and length of the transformed source code [28, 48, 61]. When analyzing a variant simulator with an off-the-shelf analysis tool, these changes can have consequences. As variability encoding can obfuscate identifier names

and structures targeted by an analysis tool, it can prevent problematic patterns from being identified [28, 48]. To avoid this, we extended SUGARC’s functionality to exclude certain identifiers from renaming [49] to arbitrary identifiers. This allows us to exclude the identifiers found in the queries from the JOERN query database [31] from renaming. During our investigation, we found that this is not always sufficient and that structural changes found within the variant simulators (e.g., additional nesting around identifiers) still caused some queries to miss matches. This highlights a common pitfall with query-based static analysis tools, such as JOERN. Queries that are too lenient lead to false positives, whereas queries that are too rigid lead to false negatives once there are slight deviations in the syntax of the modeled code pattern [39]. Striking the right balance between these extremes poses a key challenge [39], especially when analyzing variability-encoded (i.e., heavily transformed) source code.

Following our inspection of the sample of Type 2 (P-Based) vulnerability warnings for TOYBOX and BUSYBOX, we also performed an examination of the corresponding vulnerability warnings for AXTLS and found that their causes were consistent with the two categories outlined above. While the sheer number of raised vulnerability warnings did prevent us from performing a full examination for TOYBOX and BUSYBOX, an extended sample further confirmed the above-mentioned observations. We can therefore conclude RQ_2 as follows:

Conclusion RQ_2 : Our family-based analysis and the optimized product-based ground truth report nearly equivalent sets of vulnerability warnings for AXTLS. For TOYBOX and BUSYBOX, the ground truth unexpectedly reports substantially more vulnerability warnings, indicating that our family-based approach suffers from many false negatives. Closer inspection revealed that this behavior is not a conceptual issue but is caused by limitations exhibited by current variability-encoding solutions and the challenges associated with analyzing variability-encoded source code. While we tried to address both aspects in our implementation, each poses a challenge that requires continued attention from the research community.

Considering the required execution time (cf. Table 2 and the results section above), our family-based approach is positioned between the optimized product-based strategy using 2-wise and 3-wise FIS for AXTLS and BUSYBOX. For TOYBOX, it falls behind the optimized product-based strategy using 3-wise FIS by a factor of over two. We could therefore not extend earlier findings [6, 40, 60], where a family-based strategy often clearly outperformed an optimized product-based strategy using either 2-wise or 3-wise FIS. Examining the reason for this behavior, we found that although the time spent on the actual analysis using JOERN is typically within the same range, the family-based approach requires significantly more time for variability encoding than the optimized product-based strategy does for variant sampling. For instance, on average, the family-based approach requires 01:08:41 (*hh:mm:ss*) for variability encoding of TOYBOX, whereas 3-wise FIS requires only 00:04:42. Conversely, the corresponding JOERN analysis times are similar with 00:48:42 and 00:51:03 for the family-based and optimized product-based strategy, respectively. We can thus conclude RQ_3 as follows:

Conclusion RQ₃: In terms of the required execution time, our family-based approach is positioned between the optimized product-based analysis using 2-wise and 3-wise FIS for AXTLS and BUSYBOX, but slower than the optimized product-based analysis using 3-wise FIS on TOYBOX. While our approach is therefore largely comparable to an optimized product-based analysis using 2-wise and 3-wise FIS, our results do not confirm earlier findings suggesting higher efficiency of family-based analyses. This can be attributed to variability encoding of real-world C code being significantly more resource demanding than comparable t -wise FIS.

4.5 Threats to Validity

Internal Validity. Using the optimized product-based analysis as ground truth, we compared the results of the family-based approach without verifying whether vulnerability warnings refer to actual vulnerabilities (i.e., true positives in the general sense [21]). Our classification and the results for RQ_2 might therefore be biased by irrelevant vulnerability warnings (i.e., false positives). As the optimized product-based ground truth is potentially incomplete (a result of applying sampling techniques [55, 57, 60]), our results might additionally be biased by valid vulnerability warnings that were not raised by either analysis strategy (i.e., false negatives in the general sense [21]). As accurately determining complex program properties is undecidable [52], properly mitigating these threats requires a manual examination of all vulnerability warnings and the input source code. Considering the large number of vulnerability warnings (cf. Table 3) and the size of the subject systems (cf. Table 1), this is beyond the scope of this paper.

In addition, with Type 2 (P-Based) vulnerability warnings showing a significant increase for larger subject systems (cf. Table 3), we focused on this category of our classification without considering Type 3 (F-Based) vulnerability warnings in detail. While this threatens the internal validity of our results, we argue that the impact is small. The share of Type 3 (F-Based) vulnerability warnings was always below 7% and showed only minor differences across subject systems. Furthermore, an examination of a small sample of Type 3 (F-Based) vulnerability warnings showed that the underlying reason was not the promised completeness of the family-based strategy (cf. Section 2.2). Instead, the code duplication and renaming applied by variability encoding introduced potentially problematic source code patterns not present in the original source code.

External Validity. We only considered a set of three subject systems for the evaluation. Our findings may therefore not generalize to highly-configurable software systems as a whole. While this threat could only be conclusively addressed by selecting a larger number of subject systems, we tried to mitigate the risk by selecting systems frequently used in recent literature and thus serving as popular representatives of highly-configurable software (cf. Section 4.1).

Another threat is that we employed queries from the community-maintained JOERN query database [31] for the analysis of the subject systems with VARI-JOERN. Besides its partial susceptibility towards source code changes (cf. Section 4.4), this database is limited in size and its queries do not always target

vulnerability patterns. As a consequence, our results may not provide conclusive insights into the actual vulnerability discovery capabilities of our family-based approach. While using a larger vulnerability-focused set of queries would address this threat, there currently is no corresponding solution that is publicly available. Furthermore, as pointed out by Bächle et al. [8], creating such a set would require detailed expert knowledge of common vulnerability patterns and the intricacies of the subject systems that is beyond the scope of this paper.

5 Related Work

This paper introduces a family-based vulnerability discovery approach and evaluates it against a product-based analysis strategy. Consequently, it relates to family-based analysis approaches, product-based analysis approaches, and studies focusing on comparisons between family- and product-based analyses.

Family-Based Analysis Approaches. Motivated by the promised benefits in completeness and efficiency [40, 57, 60], numerous studies [5, 10, 11, 28, 35, 48, 49, 53, 55, 58] proposed family-based analysis approaches for configurable software. Focusing on the work most closely related to ours, Iosif-Lazar et al. [28] and Patterson et al. [48] developed variability encoding solutions and evaluated their capabilities by applying off-the-shelf static bug detection tools to the variability-encoded C code. Patterson [49] continued this line of work by introducing the SUGARLYZER framework. This framework automated the application of a static analysis tool to variability-encoded source code and served as the basis for our family-based vulnerability discovery approach. Despite the fact that family-based analysis approaches have been proposed for various different use cases, to the best of our knowledge, our approach is the first to target vulnerability discovery.

Product-Based Analysis Approaches. There are two recent studies [8, 43] that propose product-based vulnerability discovery approaches. Medeiros et al. [43] developed an approach that uses different sampling techniques to resolve a configurable system’s variability, allowing the off-the-shelf static analysis tool CP-CHECK [18] to be used to identify configuration-related code weaknesses. Bächle et al. [8], on the other hand, proposed the analysis platform VARI-JOERN to investigate how different sampling strengths (i.e., t -values) of t -wise FIS affect vulnerability discovery. We integrated our family-based approach into VARI-JOERN (cf. Section 3.2) and used its existing product-based analysis strategy (cf. Section 3.1) for the evaluation. What distinguishes the approaches of Medeiros et al. [43] and Bächle et al. [8] from ours is that we propose a family-based vulnerability discovery approach, whereas they focus on product-based solutions using different sampling techniques and parameterizations thereof.

Comparative Studies. Using an extended version of their model-checking tool chain SPLVERIFIER [5], Apel et al. [6] compared product-based and family-based strategies with regard to their verification time and the ability to identify defects

on six case studies. They found that the family-based strategy was nearly always superior to a product-based strategy using t -wise FIS in both regards. Focusing on real-world systems, von Rhein et al. [60] compared product- and family-based strategies for seven specific control- and data-flow analyses. They extended prior findings of Liebig et al. [40], showing that although a family-based analysis is often slower than analyzing a single variant, it is typically faster than using popular sampling techniques while still reporting findings for all variants. Similar to the work on family-based analysis approaches, what distinguishes these studies from ours is that they focus on comparisons for other use cases (verification, control- and data-flow analyses, etc.), rather than focusing on vulnerability discovery.

6 Conclusion and Future Work

In this paper, we introduced a family-based vulnerability discovery approach based on lifting by simulation [28] and evaluated it against the de facto standard of an optimized product-based analysis strategy. The results show that our approach can achieve efficiency largely comparable to a product-based analysis using 2-wise and 3-wise FIS. Furthermore, it can match the effectiveness of such an analysis for small subject systems. For more complex subject systems, it became apparent that the limitations exhibited by state-of-the-art variability encoding solutions and the challenges that arise when analyzing variability-encoded source code currently prevent an effective analysis. While efficient and effective vulnerability discovery using lifting by simulation is thus possible, both aspects require continued attention from the research community before the approach becomes viable for complex software systems. This would not only allow variability-induced vulnerabilities, such as the infamous HEARTBLEED vulnerability, to be identified early, but would also open up opportunities for other analyses on the variability-encoded source code.

In the future, we aim to continue our efforts in addressing the limitations of variability encoding solutions and the challenges encountered when analyzing variability-encoded source code. In addition, we want to extend the set of queries used for the analysis in VARI-JOERN by investigating whether vulnerability-targeting queries of other analysis tools can be translated into the query language used by JOERN. To gain more qualitative insights, we also plan to analyze properties of individual vulnerability warnings (e.g., the number of involved features) and to compare them across analysis strategies. Finally, since lifting by simulation represents only one way to implement a family-based strategy, we want to explore alternative implementations for family-based 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.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abal, I., Brabrand, C., Wasowski, A.: 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 421–432. ASE '14, ACM, New York, NY, USA (Sep 2014). <https://doi.org/10.1145/2642937.2642990>, <https://dl.acm.org/doi/10.1145/2642937.2642990>
2. Abal, I., Melo, J., Stănculescu, Ș., Brabrand, C., Ribeiro, M., Wasowski, A.: Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology* **26**(3), 1–34 (Jul 2017). <https://doi.org/10.1145/3149119>, <https://dl.acm.org/doi/10.1145/3149119>
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Berlin Heidelberg, Berlin, Heidelberg, 1 edn. (2013). <https://doi.org/10.1007/978-3-642-37521-7>, <https://link.springer.com/10.1007/978-3-642-37521-7>
4. Apel, S., Kästner, C.: An Overview of Feature-Oriented Software Development. *The Journal of Object Technology* **8**(5), 49–84 (2009). <https://doi.org/10.5381/jot.2009.8.5.c5>, http://www.jot.fm/contents/issue_2009_07/column5.html
5. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Detection of Feature Interactions using Feature-Aware Verification. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 372–375. ASE '11, IEEE, USA (Nov 2011). <https://doi.org/10.1109/ASE.2011.6100075>, <http://ieeexplore.ieee.org/document/6100075/>
6. Apel, S., von Rhein, A., Wendler, P., Groslinger, A., Beyer, D.: Strategies for Product-Line Verification: Case Studies and Experiments. In: 2013 35th International Conference on Software Engineering. pp. 482–491. ICSE '13, IEEE, USA (May 2013). <https://doi.org/10.1109/ICSE.2013.6606594>, <http://ieeexplore.ieee.org/document/6606594/>
7. axTLS Embedded SSL (Apr 2026), <https://axtls.sourceforge.net/>
8. Bächle, T., Hofmayer, E., König, C., Pett, T., Schaefer, I.: Investigating the Effects of T-Wise Interaction Sampling for Vulnerability Discovery in Highly-Configurable Software Systems. In: Proceedings of the 29th ACM International Systems and Software Product Line Conference. SPLC '25, ACM, New York, NY, USA (Aug 2025). <https://doi.org/10.1145/3744915.3748462>
9. Benavides, D., Sundermann, C., Feichtinger, K., Galindo, J.Á., Rabiser, R., Thüm, T.: UVL: Feature modelling with the Universal Variability Language. *Journal of Systems and Software* **225** (Jul 2025). <https://doi.org/10.1016/j.jss.2024.112326>, <https://linkinghub.elsevier.com/retrieve/pii/S0164121224003704>
10. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: SPL^{LIFT}: Statically Analyzing Software Product Lines in Minutes Instead of Years. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 355–364. PLDI '13, ACM, New York, NY, USA (Jun 2013). <https://doi.org/10.1145/2491956.2491976>, <https://dl.acm.org/doi/10.1145/2491956.2491976>
11. Brabrand, C., Ribeiro, M., Tolêdo, T., Borba, P.: Intraprocedural Dataflow Analysis for Software Product Lines. In: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development. pp. 13–24. AOSD '12, ACM, New York, NY, USA (Mar 2012). <https://doi.org/10.1145/2162049.2162052>, <https://dl.acm.org/doi/10.1145/2162049.2162052>

12. Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., Borba, P.: Intraprocedural Dataflow Analysis for Software Product Lines. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Leavens, G.T., Chiba, S., Tanter, É. (eds.) *Transactions on Aspect-Oriented Software Development X*, vol. 7800, pp. 73–108. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36964-3_3, http://link.springer.com/10.1007/978-3-642-36964-3_3
13. BusyBox (Apr 2026), <https://busybox.net/>
14. Castro, T., Teixeira, L., Alves, V., Apel, S., Cordy, M., Gheyi, R.: A Formal Framework of Software Product Line Analyses. *ACM Transactions on Software Engineering and Methodology* **30**(3), 1–37 (Jul 2021). <https://doi.org/10.1145/3442389>, <https://dl.acm.org/doi/10.1145/3442389>
15. Chess, B., McGraw, G.: Static Analysis for Security. *IEEE Security and Privacy Magazine* **2**(6), 76–79 (Nov 2004). <https://doi.org/10.1109/MSP.2004.111>, <http://ieeexplore.ieee.org/document/1366126/>
16. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and Adequacy in Software Product Line Testing. In: *Proceedings of the ISSSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*. pp. 53–63. ROSATEA '06, ACM, New York, NY, USA (Jul 2006). <https://doi.org/10.1145/1147249.1147257>, <https://dl.acm.org/doi/10.1145/1147249.1147257>
17. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering* **34**(5), 633–650 (Sep 2008). <https://doi.org/10.1109/TSE.2008.50>, <http://ieeexplore.ieee.org/document/4564473/>
18. Cppcheck (Apr 2026), <https://cppcheck.sourceforge.io/>
19. Dimovski, A.S., Brabrand, C., Wąsowski, A.: Finding suitable variability abstractions for lifted analysis. *Formal Aspects of Computing* **31**(2), 231–259 (Apr 2019). <https://doi.org/10.1007/s00165-019-00479-y>, <https://dl.acm.org/doi/10.1007/s00165-019-00479-y>
20. Evans, D., Larochelle, D.: Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software* **19**(1), 42–51 (Aug 2002). <https://doi.org/10.1109/52.976940>, <http://ieeexplore.ieee.org/document/976940/>
21. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Chapter One - Security Testing: A Survey. In: *Advances in Computers*, vol. 101, pp. 1–51. Elsevier, Cambridge, MA, USA (2016). <https://doi.org/10.1016/bs.adcom.2015.11.003>, <https://linkinghub.elsevier.com/retrieve/pii/S0065245815000649>
22. Ferreira, G., Malik, M., Kästner, C., Pfeffer, J., Apel, S.: Do `#if`defs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. pp. 65–73. SPLC '16, ACM, New York, NY, USA (Sep 2016). <https://doi.org/10.1145/2934466.2934467>, <https://dl.acm.org/doi/10.1145/2934466.2934467>
23. Gazzillo, P., Grimm, R.: SuperC: Parsing All of C by Taming the Preprocessor. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 323–334. PLDI '12, ACM, New York, NY, USA (Jun 2012). <https://doi.org/10.1145/2254064.2254103>, <https://dl.acm.org/doi/10.1145/2254064.2254103>
24. Gazzillo, P., Wei, S.: Conditional Compilation is Dead, Long Live Conditional Compilation! In: *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. pp. 105–108. ICSE-NIER '19,

- IEEE, USA (May 2019). <https://doi.org/10.1109/ICSE-NIER.2019.00035>, <https://ieeexplore.ieee.org/document/8805666/>
25. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* **24**(2), 674–717 (Apr 2019). <https://doi.org/10.1007/s10664-018-9635-4>, <http://link.springer.com/10.1007/s10664-018-9635-4>
 26. The Heartbleed Bug (Apr 2026), <https://heartbleed.com/>
 27. Heradio, R., Fernandez-Amoros, D., Galindo, J.Á., Benavides, D., Batory, D.: Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering* **27**(2), 44 (Mar 2022). <https://doi.org/10.1007/s10664-021-10102-5>, <https://link.springer.com/10.1007/s10664-021-10102-5>
 28. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective Analysis of C Programs by Rewriting Variability. *The Art, Science, and Engineering of Programming* **1**(1), 1 (Jan 2017). <https://doi.org/10.22152/programming-journal.org/2017/1/1>, <http://programming-journal.org/2017/1/1>
 29. ISO/IEC: ISO/IEC 27000:2018(E) - Information technology - Security techniques - Information security management systems - Overview and vocabulary (Feb 2018)
 30. Joern - The Bug Hunter's Workbench (Apr 2026), <https://joern.io/>
 31. Joern Query Database (Apr 2026), <https://queries.joern.io/>
 32. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In: Whittle, J., Clark, T., Kühne, T. (eds.) *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems. MODELS '11*, vol. 6981, pp. 638–652. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_47, http://link.springer.com/10.1007/978-3-642-24485-8_47
 33. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: *Proceedings of the 30th International Conference on Software Engineering*. p. 311. ICSE '08, ACM, New York, NY, USA (2008). <https://doi.org/10.1145/1368088.1368131>, <http://portal.acm.org/citation.cfm?doid=1368088.1368131>
 34. Kconfig Language (Apr 2026), <https://docs.kernel.org/kbuild/kconfig-language.html>
 35. Kenner, A., Kästner, C., Haase, S., Leich, T.: TypeChef: Toward Type Checking #ifdef Variability in C. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. pp. 25–32. FOSD '10, ACM, New York, NY, USA (Oct 2010). <https://doi.org/10.1145/1868688.1868693>, <https://dl.acm.org/doi/10.1145/1868688.1868693>
 36. Krieter, S.: 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*. pp. 1–4. SPLC '20, ACM, New York, NY, USA (Oct 2020). <https://doi.org/10.1145/3382025.3414989>, <https://dl.acm.org/doi/10.1145/3382025.3414989>
 37. Krieter, S., Thüm, T., Schulze, S., Ruland, S., Lochau, M., Saake, G., Leich, T.: T-Wise Presence Condition Coverage and Sampling for Configurable Systems (May 2022). <https://doi.org/10.48550/arXiv.2205.15180>, <http://arxiv.org/abs/2205.15180>
 38. Krieter, S., Thüm, T., Schulze, S., Saake, G., Leich, T.: YASA: Yet Another Sampling Algorithm. In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. pp. 1–10. VaMoS '20, ACM, New York, NY, USA (Feb 2020). <https://doi.org/10.1145/3377024.3377042>, <https://dl.acm.org/doi/10.1145/3377024.3377042>

39. Li, Z., Liu, Z., Wong, W.K., Ma, P., Wang, S.: Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Transactions on Dependable and Secure Computing* **21**(5), 1–18 (2024). <https://doi.org/10.1109/TDSC.2024.3354789>, <https://ieeexplore.ieee.org/document/10400834/>
40. Liebig, J., Von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C.: Scalable Analysis of Variable Software. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. pp. 81–91. ESEC/FSE '13, ACM, New York, NY, USA (Aug 2013). <https://doi.org/10.1145/2491411.2491437>, <https://dl.acm.org/doi/10.1145/2491411.2491437>
41. Livshits, V.B., Lam, M.S.L.: 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, p. 18. USENIX Association, USA (Jul 2005), <https://dl.acm.org/doi/10.5555/1251398.1251416>
42. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A Comparison of 10 Sampling Algorithms for Configurable Systems. In: *Proceedings of the 38th International Conference on Software Engineering*. pp. 643–654. ICSE '16, ACM, New York, NY, USA (May 2016). <https://doi.org/10.1145/2884781.2884793>, <https://dl.acm.org/doi/10.1145/2884781.2884793>
43. Medeiros, F., Ribeiro, M., Gheyi, R., Braz, L., Kästner, C., Apel, S., Santos, K.: An Empirical Study on Configuration-Related Code Weaknesses. In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. pp. 193–202. SBES '20, ACM, New York, NY, USA (Oct 2020). <https://doi.org/10.1145/3422392.3422409>, <https://dl.acm.org/doi/10.1145/3422392.3422409>
44. Mordahl, A., Oh, J., Koc, U., Wei, S., Gazzillo, P.: 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*. pp. 50–61. ESEC/FSE '19, ACM, New York, NY, USA (Aug 2019). <https://doi.org/10.1145/3338906.3338967>, <https://dl.acm.org/doi/10.1145/3338906.3338967>
45. Oh, J., Batory, D., Heule, M., Myers, M., Gazzillo, P.: Uniform Sampling from Kconfig Feature Models. Technical Report 19, The University of Texas at Austin, Department of Computer Science, Austin, Texas (Sep 2019), <https://api.semanticscholar.org/CorpusID:92985846>
46. Oh, J., Yildiran, N.F., Braha, J., Gazzillo, P.: Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. pp. 893–905. ESEC/FSE '21, ACM, New York, NY, USA (Aug 2021). <https://doi.org/10.1145/3468264.3468578>, <https://dl.acm.org/doi/10.1145/3468264.3468578>
47. OpenSSL (Apr 2026), <https://www.openssl.org/>
48. Patterson, Z., Zhang, Z., Pappas, B., Wei, S., Gazzillo, P.: SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C. In: *Proceedings of the 44th International Conference on Software Engineering*. pp. 2056–2067. ICSE '22, ACM, New York, NY, USA (May 2022). <https://doi.org/10.1145/3510003.3512763>, <https://dl.acm.org/doi/10.1145/3510003.3512763>
49. Patterson, Z.: Toward Applying Variability-Oblivious Static Analyses to Software Product Lines. Ph.D. thesis, The University of Texas at Dallas, Dallas (Dec 2023), <https://utd-ir.tdl.org/items/1623bed4-684c-44e7-94c2-f20cfeb7c976>
50. Pett, T., Heß, T., Krieter, S., Thüm, T., Schaefer, I.: Continuous T-Wise Coverage. In: *Proceedings of the 27th ACM International Systems and Software Product Line*

- Conference - Volume A. pp. 87–98. SPLC '23, ACM, New York, NY, USA (Aug 2023). <https://doi.org/10.1145/3579027.3608980>, <https://dl.acm.org/doi/10.1145/3579027.3608980>
51. Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I.: Product Sampling for Product Lines: The Scalability Challenge. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A. pp. 78–83. SPLC '19, ACM, New York, NY, USA (Sep 2019). <https://doi.org/10.1145/3336294.3336322>, <https://dl.acm.org/doi/10.1145/3336294.3336322>
 52. Rice, H.G.: Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society* **74**(2), 358–366 (1953). <https://doi.org/10.1090/S0002-9947-1953-0053041-6>, <https://www.jstor.org/stable/1990888>
 53. Schubert, P.D., Gazzillo, P., Patterson, Z., Braha, J., Schiebel, F., Hermann, B., Wei, S., Bodden, E.: Static data-flow analysis for software product lines in C: Revoking the preprocessor’s special role. *Automated Software Engineering* **29**(35) (May 2022). <https://doi.org/10.1007/s10515-022-00333-1>, <https://link.springer.com/10.1007/s10515-022-00333-1>
 54. Schulze, S., Richers, O., Schaefer, I.: Refactoring Delta-Oriented Software Product Lines. In: Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development. pp. 73–84. AOSD '13, ACM, New York, NY, USA (Mar 2013). <https://doi.org/10.1145/2451436.2451446>, <https://dl.acm.org/doi/10.1145/2451436.2451446>
 55. Shahin, R., Chechik, M.: Automatic and Efficient Variability-Aware Lifting of Functional Programs. *Proceedings of the ACM on Programming Languages* **4**(OOPSLA), 1–27 (Nov 2020). <https://doi.org/10.1145/3428225>, <https://dl.acm.org/doi/10.1145/3428225>
 56. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: 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 (Aug 2001), <https://www.usenix.org/conference/10th-usenix-security-symposium/detecting-format-string-vulnerabilities-type-qualifiers>
 57. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys* **47**(1), 1–45 (Jul 2014). <https://doi.org/10.1145/2580950>, <https://dl.acm.org/doi/10.1145/2580950>
 58. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-Based Deductive Verification of Software Product Lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering. pp. 11–20. GPCE '12, ACM, New York, NY, USA (Sep 2012). <https://doi.org/10.1145/2371401.2371404>, <https://dl.acm.org/doi/10.1145/2371401.2371404>
 59. Toybox (Apr 2026), <https://www.landley.net/toybox/>
 60. von Rhein, A., Liebig, J., Janker, A., Kästner, C., Apel, S.: Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* **27**(4), 1–33 (Oct 2018). <https://doi.org/10.1145/3280986>, <https://dl.acm.org/doi/10.1145/3280986>
 61. von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., Apel, S.: Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* **85**(1), 125–145 (Jan 2016). <https://doi.org/10.1016/j.jlmp.2015.06.007>, <https://linkinghub.elsevier.com/retrieve/pii/S2352220815000577>

62. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and Discovering Vulnerabilities with Code Property Graphs. In: 2014 IEEE Symposium on Security and Privacy. pp. 590–604. S&P '14, IEEE, USA (May 2014). <https://doi.org/10.1109/SP.2014.44>, <http://ieeexplore.ieee.org/document/6956589/>
63. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In: 2015 IEEE Symposium on Security and Privacy. pp. 797–812. S&P '15, IEEE, USA (May 2015). <https://doi.org/10.1109/SP.2015.54>, <https://ieeexplore.ieee.org/document/7163061/>
64. Zhou, L., Huang, M., Li, Y., Nie, Y., Li, J., Liu, Y.: GraphEye: A Novel Solution for Detecting Vulnerable Functions Based on Graph Attention Network. In: 2021 IEEE Sixth International Conference on Data Science in Cyberspace. pp. 381–388. DSC '21, IEEE, USA (Oct 2021). <https://doi.org/10.1109/DSC53577.2021.00060>, <https://ieeexplore.ieee.org/document/9750460/>