Karlsruhe Institute of Technology

Test, Validation, Analysis

Family-Based Vulnerability Discovery for Software Product Lines

Master's Thesis
of

Tim Bächle

at the Department of Informatics
Institute of Information Security and Dependability
Test, Validation and Analysis (TVA)

Reviewer:	Prof. Dr.-Ing. Ina Schaefer
Second Reviewer:	Prof. Dr. Ralf Reussner
Advisors:	M.Sc. Christoph König
	M.Sc. Tobias Pett

Completion period: 28. May 2024 – 28. November 2024

Zusammenfassung

Die Ausnutzung von Software-Schwachstellen durch Angreifer kann katastrophale Konsequenzen mit sich bringen. Entdeckungen wie *Heartbleed* haben eindrucksvoll aufgezeigt, dass schwerwiegende Schwachstellen trotz dieser ernsthaften Gefahr regelmäßig unbemerkt bleiben. Dieses Problem wird durch die Einführung von *Softwareproduktlinien (SPLs)*, hochkonfigurierbaren Softwaresystemen, welche zu einer Vielzahl an potenziell verwundbaren Softwareprodukten führen, weiter verschärft. Die Anwendung konventioneller Lösungen der statischen Codeanalyse, welche häufig als Hilfsmittel für die Erkennung von Schwachstellen genutzt werden, trifft bei diesen Systemen auf Skalierbarkeits- und Vollständigkeitsprobleme. Um diese Probleme zu adressieren, wurde durch Forschende das Konzept der *familienbasierten* Analysen eingeführt. Diese zielen darauf ab, SPLs als Ganzes und nicht nur einzelne Softwareprodukte zu analysieren. Bislang wurden familienbasierte Analysen für verschiedenste Ziele vorgestellt. Es existieren jedoch kaum Arbeiten mit dem Fokus der familienbasierten Schwachstellenerkennung. Besonders existiert bislang keine Lösung, welche die Vorteile von *Query-Based Static Application Security Testing (Q-SAST)*, also der Möglichkeit, Schwachstellenmuster in Anfragen zu kodieren und für die Analyse zu verwenden, ausnutzt. Entsprechende Werkzeuge bieten viele Vorteile und ermöglichen die einfache Erkennung selbst komplexer Schwachstellen. In dieser Arbeit stellen wir einen familienbasierten Analyseansatz vor, welcher die Vorteile von Q-SAST für die Erkennung von Schwachstellen in SPLs ausnutzt. Zu diesem Zweck untersuchen und vergleichen wir zwei Strategien, durch welche ein handelsübliches Q-SAST-Werkzeug *geliftet*, das heißt für die Analyse von SPLs verwendet werden kann. Basierend auf den Ergebnissen dieses Vergleichs liften wir das handelsübliche Q-SAST-Werkzeug JOERN durch die Anwendung der Strategie des *Lifting by Simulation*. Bei dieser Strategie wird die Variabilität einer SPL in einem Prozess, welcher als *Variability Encoding* bekannt ist, in eine Form umgeschrieben, mit welcher das Analysewerkzeug arbeiten kann. Die resultierende Implementierung wird in die bestehende Analyseplattform VARI-JOERN integriert und auf Basis von drei praxisrelevanten SPLs hinsichtlich ihrer Effektivität bei der Identifizierung potenzieller Schwachstellen und ihrer Gesamteffizienz evaluiert. Die Ergebnisse zeigen vielversprechende Effektivität und eine durchweg hohe Effizienz. Es zeigt sich jedoch auch, dass sich der Erfolg von Lifting by Simulation auf die Verfügbarkeit von hochqualitativen Variability-Encoding-Lösungen stützt. In dieser Hinsicht weisen existierende, dem Stand der Technik entsprechende Werkzeuge noch erheblichen Raum für Verbesserungen auf.

Abstract

The exploitation of software vulnerabilities by attackers can have disastrous consequences. Discoveries like *Heartbleed* strikingly demonstrated that despite this grave danger, serious vulnerabilities frequently go unnoticed. This problem is exacerbated through the introduction of *Software Product Lines (SPLs)*, highly configurable software systems that give rise to a vast array of potentially vulnerable software products. Applying conventional static source code analysis solutions, which are a common aid for the discovery of vulnerabilities, to these systems faces scalability and completeness issues. To address these issues, researchers introduced the concept of *family-based* analyses, which aim to analyze an SPL in its entirety rather than individual products. While family-based analyses have been proposed for various objectives, previous work dedicated to family-based vulnerability discovery has been limited. Notably, there is no solution that leverages the benefits of *Query-Based Static Application Security Testing (Q-SAST)*, which allows vulnerability patterns to be codified into queries controlling the analysis. Corresponding tools provide many benefits and enable convenient detection of even sophisticated vulnerabilities. In this thesis, we propose a family-based analysis approach that leverages the benefits of Q-SAST for vulnerability discovery in SPLs. To this end, we examine and compare two strategies through which an off-the-shelf Q-SAST tool can be *lifted*, i.e., employed for the analysis of SPLs. Based on the results of this comparison, we lift the off-the-shelf Q-SAST tool JOERN by applying the strategy of *lifting by simulation*. This strategy entails rewriting the variability of an SPL into a form on which the analysis tool can operate in a process known as *variability encoding*. The resulting implementation is integrated into the analysis platform VARI-JOERN and evaluated on three real-world SPLs with regard to its effectiveness in identifying potential vulnerabilities and its overall efficiency. The results showcase promising effectiveness while also demonstrating high overall efficiency. However, they also indicate that the success of lifting by simulation hinges on the availability of high-quality variability-encoding solutions. In this regard, existing state-of-the-art tooling exhibits considerable room for improvement.

Contents

List of Figures

List of Tables

Listings

Acronyms

AST	Abstract Syntax Tree
CFG	Control Flow Graph
CPG	Code Property Graph
CPGQL	Code Property Graph Query Language
DB	Database
JSON	JavaScript Object Notation
NQ-SAST	Non-Query-Based Static Application Security Testing
PDG	Program Dependence Graph
SAST	Static Application Security Testing
SPL	Software Product Line
Q-SAST	Query-Based Static Application Security Testing
VIV	Variability-Induced Vulnerability
XML	Extensible Markup Language

1. Introduction

As discoveries like the Heartbleed [24n] vulnerability strikingly demonstrated, vulnerabilities in a software project's source code can easily be overlooked, lying dormant for years before being discovered. Crucially, the exploitation of even a single vulnerability by an attacker can lead to severe consequences, ranging from compromising sensitive user data to causing harm to virtual or even physical infrastructure [Zho+21]. Accordingly, a fundamental objective of any development effort aiming for high software quality, especially in terms of security, is the identification and subsequent removal of vulnerabilities [Fel+16]. To this end, static source code analysis acts as an essential aid for the identification of corresponding structures across large systems, for which manual code reviews do not scale [Shi+22]. There are numerous so-called Static Application Security Testing (SAST) tools [Fel+16] dedicated to this purpose. These tools can be categorized into Query-Based Static Application Security Testing (Q-SAST) and Non-Query-Based Static Application Security Testing (NQ-SAST) tools. NQ-SAST tools rely solely on searching for built-in vulnerability patterns, typically contributed by their developers or maintainers [Li+24]. Q-SAST tools, on the other hand, share the key design principle of allowing vulnerability patterns to be freely codified into queries [Li+24]. Usually, these queries are contributed by the community and collected in a central query Database (DB), ultimately facilitating both reuse and knowledge sharing [Li+24].

Popular SAST tools, such as CBMC [24c], Clang Static Analyzer [24d], CodeQL [24e], or Infer [24g], often face limitations when it comes to the analysis of Software Product Lines (SPLs). In principle, an SPL is a family of related software products (i.e., programs) that relies on a common code base [Sch+22; Thü+14]. While individual products of this family are related in that they all share a common core [Aba+17], they are distinguished by the set of features (i.e., end-user-visible behavior [Ape+11; Cas+21; SRS13]) they provide [Ape+13b; Thü+14]. Considering that SPLs are finding their way into ever more safety- and mission-critical areas of application [Ape+13b; Pet+19], being able to analyze these systems for the presence of vulnerabilities is of great interest. Yet, from a technical viewpoint, conventional SAST tools cannot cope with the variable source code found within an SPL [Lie+13; Pat23]. Thus, they are generally referred to as *variability-oblivious* [Mor+19; Pat+22; Pat23] and considered insufficient [Aba+17].

Variability-aware analysis approaches, on the other hand, take the variability of an SPL into account [Ape+13a; Tol+24]. In its simplest form, a variability-aware analysis can be realized through a *product-based* strategy [Ape+13b; Thü+14]. This strategy comprises deriving a number of software products from an SPL (thus resolving variability) and analyzing each product individually using variability-oblivious analysis techniques [Thü+14]. In the case of an *exhaustive* product-based strategy, this entails the analysis of all products derivable from the SPL [Thü+14; von+16]. Since the number of derivable products can grow exponentially with the number of features, in practice product-based analyses are often augmented by a sampling strategy for the selection of a subset of products to analyze [Lie+13; Pet+19; Thü+14; von+18]. This revised strategy is referred to by Thüm et al. [Thü+14] as an *optimized* product-based strategy. The optimized strategy improves scalability, given that only a fraction of the large number of possible products is considered [Lie+13; Pet+23; Thü+14; von+18]. However, it also renders the overall analysis incomplete [Ape+13a; Ape+13b; Thü+14; von+16; von+18]; that is, the results (e.g., the identified vulnerabilities) are not necessarily identical to the ones produced by an exhaustive product-based strategy [Ape+13a; von+16; von+18]. Specifically, findings that are confined to products not considered in the sample, might be missed [Ape+13a; Thü+14]. Therefore, especially for the analysis of SPLs in safety- and mission-critical areas, where a comprehensive analysis is highly desirable, optimized product-based approaches can only be a suboptimal solution.

An alternative is to follow a *family-based* strategy [Thü+14]. This strategy entails operating directly on an entire SPL rather than analyzing the generated software products individually [Ape+13b; Ios+17; Lie+13]. As a result, it avoids the redundant computations that result from analyzing many products sharing great similarities individually [Ape+13b; DBW19; Lie+13; Thü+14]. The family-based strategy therefore promises to maintain scalability and precision. However, it also necessitates dedicated analysis approaches capable of handling the feature and variability information encountered when operating on an SPL directly [Thü+14]. In this regard, traditional off-the-shelf tools used for the analysis of single products often face limitations and cannot be applied to an SPL's source code [Bod+13; Lie+13; Thü+14]. Accordingly, one solution is to develop new variability-aware analysis tools [Pat23; Thü+14]. Evidently, developing a novel tool from the ground up can incur a substantial engineering effort. Additionally, established variability-oblivious tools have often been improved over decades [Ios+17; Pat23]. Thus, achieving the same level of quality with a new implementation can be difficult and akin to playing catch-up with a field that is still continuously advancing [GW19; Pat+22]. As an alternative solution, the operation of existing variability-oblivious tools can be adapted, enabling them to work with variable code [Lie+13; Thü+14]. Within the literature, this process is generally referred to as *lifting* [Bod+13; Cas+21; DBW19].

Considering the field of family-based analyses, there are approaches dedicated to a variety of different objectives, such as the calculation of software metrics [Kra19; Lie+10], semantic slicing [GS19; GS20], type checking [Ken+10], model checking [Ape+13b; Cla+10; Mei+14; Oh+21; PS08], or data-flow analysis [Bod+13; Bra+12; Sch+22; von+18]. However, research on family-based analyses dedicated solely to the identification of vulnerabilities in source code has been limited. This is unfortunate, considering that SPLs are particularly popular in fields such as embedded software, system-level software, frameworks, development platforms, and web solutions [DBW19]. Systems within these fields form parts of our most critical infrastructure [GW19; Pat+22], where the exploitation of vulnerabilities can have particularly severe consequences. Trying to address this problem, previous work by Patterson [Pat23] focused on applying off-the-shelf NQ-SAST tools to SPLs. However, many common vulnerability types, such as SQL injections or buffer overflows, constitute taint-style vulnerabilities [Yam+15] whose identification can require knowledge about certain characteristics of the considered system (such as the functions realizing sensitive operations). Detecting these vulnerabilities with NQ-SAST tools, whose analysis cannot usually be tailored to a specific system [Li+24], can be difficult. In contrast, by allowing vulnerability patterns to be freely specified as queries, Q-SAST tools enable each user to contribute new queries, as well as customize existing ones for a specific use case [Li+24]. However, currently, there is no family-based vulnerability discovery approach leveraging the benefits of Q-SAST.

Contributions

In view of the research gap outlined above, the overall goal of this thesis is to provide a family-based static analysis approach dedicated to the discovery of vulnerabilities in the source code of real-world SPLs. Addressing the limitations of earlier work, the focus of this approach is placed on leveraging the benefits of Q-SAST. Considering the substantial engineering effort associated with developing a novel approach from the ground up, the concrete aim is to alter the operation of an off-the-shelf Q-SAST tool for the analysis of whole SPLs. More specifically, we aim to lift the analysis tool Joern [24h] to the domain of SPLs, given its extensive vulnerability discovery capabilities and open-source nature.

In summary, we make the following contributions:

- We describe the design implications of two frequently proposed lifting strategies and discuss their individual benefits and drawbacks.
- We present a family-based analysis approach realized by applying the strategy of lifting by simulation using variability encoding to the Q-SAST tool Joern.
- We implement our presented analysis approach and integrate it alongside the product-based analysis strategy already available within the analysis platform Vari-Joern.
- Finally, we evaluate our family-based analysis approach on three real-world SPLs, demonstrating not just its promising effectiveness compared to a product-based baseline but also its overall efficiency, which makes it suitable for practical use.

Outline

The remaining parts of this thesis are organized as follows: Sections 1.1 and 1.2 complete the introduction by expanding on the research problem this thesis seeks to address and defining its scope, respectively. Following the introduction, Chapter 2 provides an overview of fundamental concepts relating to the fields of SPLs, conditional compilation, and static source code analysis. Next, Chapter 3 details two common strategies through which a Q-SAST tool can be lifted to the domain of SPLs, discusses their strengths and weaknesses, and describes our choice. Based on this choice, Chapter 4 describes the design of our analysis approach, the implementation of which is detailed in Chapter 5. Chapter 6 describes the evaluation of our approach on real-world SPLs, including an overview of threats to its validity. Considering previous research relating to individual components of our analysis approach as well as its overall goal, Chapter 7 surveys the existing work and discusses its relation to this thesis. Finally, Chapter 8 concludes the thesis and gives an outlook on potential future work.

1.1 Problem Statement

The ISO/IEC 27000 standard [ISO18] defines a *vulnerability* as a weakness that can be exploited by one or more threats. In the context of software, exploited weaknesses in design or implementation can allow malicious users to inflict various types of harm. These can range from compromised sensitive data to unauthorized resource consumption and even damage to physical or virtual infrastructure [Zho+21]. Considering the high costs exploited weaknesses can cause [Fel+16; LL05], analyzing software systems for the presence of vulnerabilities is often an indispensable activity in the development process. Especially for large systems, for which extensive manual code reviews are often too costly [Shi+22], this activity typically involves applying static analysis. Although static analysis is limited to providing approximate results,[1] it remains popular as it allows for a proactive approach. This means that vulnerabilities can be identified early on, before the source code is ever actually run, thus minimizing both their impact and cost [Bra+13; LL05; Sha+01]. Q-SAST tools further expand on this advantage. Since they typically encourage the community to contribute their own vulnerability queries, newly discovered vulnerabilities can be rapidly incorporated into the analysis, allowing for their timely combat [Li+24].

However, off-the-shelf Q-SAST tools, as they are often used in industry for the static analysis of conventional software systems, generally face limitations when it comes to the analysis of SPLs [Lie+13; Pat23]. In this context, an SPL can be understood as a family of related software products (i.e., programs) that relies on a common code base [Sch+22; Thü+14]. This code base comprises two parts: (1) a common core reflecting the functionality shared by all software products derivable from the SPL [Aba+17] and (2) a set of variable features representing increments in user-visible functionality [Ape+11; Cas+21; SRS13], whose selection distinguishes the individual products of the SPL [Ape+13b; Thü+14]. The problem that arises from this implementation lies in the fact that most off-the-shelf Q-SAST tools are variability-oblivious and hence cannot cope with the variability introduced by the set of features [Lie+13; Pat+22; Pat23]. The simple C function in Listing 1.1 exemplifies the underlying problem on a smaller scale and will serve as a running example in the following chapters. The function follows common practice in many industrial SPLs [Aba+17; AK09; Bra+13; DBW19] and implements variability by use of conditional compilation via C preprocessor conditionals (`#ifdefs`). By itself, it can thus be seen as an example of a tiny SPL comprising two possible products: one with the feature `PROCESS_INPUT`[2] enabled and one with it disabled.

[1]According to Rice's theorem [Ric53], the identification of non-trivial program properties, such as vulnerabilities [Yam15], is undecidable and thus requires approximations in practice [Aba+17; ACB16; CM04].

[2]We follow the naming convention used in the Linux kernel [Aba+17; ABW14; Kra19; Tar+12] and prefix preprocessor macros relating to features of the SPL with `CONFIG_`.

```
 1  void foo() {
 2      int x = source(); // Potentially 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          #endif
 8          sink(y); // Security-sensitive operation.
 9      }
10  }
```

Listing 1.1: A variable C function inspired by the example provided by Yamaguchi et al. [Yam+14] that exhibits a Variability-Induced Vulnerability (VIV)

Applying conventional Q-SAST tools to the tiny SPL of Listing 1.1 in search of vulnerabilities poses a number of challenges. These challenges are based on the fact that formally preprocessor annotations (such as the ones found in lines 5 and 7 of Listing 1.1) function as a metalanguage, adding lightweight metaprogramming capabilities to C [Ape+13a; KGO11; Lie+10; Tar+12]. As a result, they are not part of the C language itself [Mor+19]. From a technical perspective, this means that unpreprocessed source code does not conform to the formal C grammar [GJ03] and thus cannot be parsed by standard parsers. For conventional Q-SAST tools, which typically operate on abstract source code representations such as Abstract Syntax Trees (ASTs) [Lie+13; von+18], the construction of which requires parsing, this constitutes a major problem. Accordingly, many tools either settle for the analysis of a default product by stripping variable code of all variability (e.g., by applying the preprocessor leaving all configuration-related preprocessor macros undefined) or refuse to operate on unpreprocessed code altogether. Evidently, both of these solutions prove insufficient, since only a tiny portion of the SPL is analyzed, if at all.

The `foo` function of Listing 1.1 illustrates the fatal consequences the aforementioned solutions can have. Even if a default product, where the `PROCESS_INPUT` feature is left unselected, is analyzed, the `foo` function will never be considered with the assignment of line 6 in place. Crucially, it is this line of code that completes a taint-style vulnerability [Yam+15], allowing user-controlled (i.e., tainted) data from a potentially attacker-controlled source (line 2) to reach a security-sensitive sink operation (line 8) without previously undergoing proper sanitization (line 3). Failing to identify this vulnerability could allow an attacker to exploit it to detrimental effect. For instance, assuming that the sink uses the tainted data as the number of bytes to copy between two buffers via a call to `memcpy`, a negative integer would be interpreted as unsigned.[3] This, in turn, would lead to a large copy operation that may exceed the size of either buffer. Evidently, this could cause the program to crash. Even worse, it could allow arbitrary access to the program's heap memory, as was the root problem in the case of the well-known Heartbleed vulnerability [Yam+15]. Consequently, relying solely on conventional SAST tools proves inadequate, especially when modern SPLs boast an enormous number of products, each having the possibility to contain vulnerabilities not found in any other product.

[3]Passing a signed integer to `memcpy` will lead to an implicit conversion to an unsigned integer, as the function expects the number of bytes to copy between buffers to be unsigned.

In general, vulnerabilities and bugs that only manifest in certain products and result from particular feature configurations of an SPL (cf. Listing 1.1) are not a merely theoretical problem. In fact, they have been investigated in the context of multiple real-world SPLs [Aba+17; ABW14; Mor+19]. Corresponding bugs are often referred to as *variability bugs* [Aba+17; ABW14; Mor+19]. Inspired by this term, we refer to vulnerabilities only present in certain feature configurations as *Variability-Induced Vulnerabilitys (VIVs)*. A prominent example of a VIV is the aforementioned Heartbleed vulnerability that was identified in the OpenSSL[4] cryptography library in 2014. This vulnerability only manifested if a certain feature of the library was enabled [von+16] and allowed attackers to compromise sensitive user data [24n].

For the systematic identification of VIVs, one solution is to resolve the variability of the SPL by deriving every possible product and analyzing each one individually with a standard Q-SAST tool (i.e., following an exhaustive product-based strategy [Thü+14]). However, given that the number of products derivable from an SPL can grow exponentially with the number of features [Lie+13; Thü+14; von+18], this approach is impractical for many real-world SPLs, often containing hundreds to thousands of features [Ape+13b; Pet+19]. Considering SPLs of this magnitude, even sampling and analyzing just a representative subset of the possible products (i.e., following an optimized product-based strategy [Thü+14]) can be challenging. Determining a minimal sample with regard to popular coverage criteria often constitutes an NP-complete problem [Lie+13; von+18] and can yet result in large sample sizes. Thus, it is hardly surprising that sampling algorithms often face scalability issues when faced with large SPLs [Pet+19]. Additionally, since the optimized product-based strategy does not consider every product of an SPL in the analysis, there is once again a risk of missing many VIVs.

Unfortunately, as instances like the Heartbleed vulnerability showed, missing VIVs until they are released to customers is not uncommon in practice. Additionally, given the vast size of modern SPLs, developers themselves may not always be fully aware of the impact a certain feature or interplay thereof can have on products [ABW14]. For instance, in 2012 Apple accidentally shipped a version of its operating system Mac OS X with logging functionality enabled, not noticing that this led to users' passwords being stored in clear text [Sch+22]. Both aspects underscore the benefits a family-based analysis strategy provides. In this regard, the goal of a family-based strategy is to operate on an entire SPL at once rather than analyzing the generated software products individually [Ape+13b; Ios+17; Lie+13]. This is usually achieved by using dedicated techniques for the analysis of source code still containing variability information [DBW19; Lie+13]. Besides avoiding the redundant computations for parts shared between multiple products [Ape+13b; Lie+13; Thü+14], this approach allows the entirety of an SPL to be analyzed without incurring prohibitive costs, identifying VIVs and other structures of interest more reliably. While there are family-based analyses for various objectives, the work on family-based analyses dedicated to vulnerability discovery has been limited and currently there is no approach that leverages the benefits of Q-SAST. This leads to the main research question (***MRQ***) of this thesis shown below.

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

MRQ: *How can a Q-SAST tool be lifted to the domain of SPLs, i.e., be employed for a family-based analysis that is both effective and efficient?*

To answer the main research question, we focus on adapting the operation of an existing Q-SAST tool, allowing it to work on SPLs (i.e., lifting), instead of creating novel tooling from scratch. This is due to the enormous effort associated with engineering an analysis for a fully-fledged programming language from the ground up. Additionally, considering the vast amount of work and expertise that went into improving many off-the-shelf tools, often over the duration of multiple decades [Pat23], achieving a similar level of quality would be beyond the reach of this thesis.

1.2 Scope

Software Product Lines and Preprocessor Usage

For the scope of this thesis, we focus on the analysis of SPLs implemented in C. We focus on C since it is widely used for the implementation of SPLs, representing some of the largest and most configurable systems found in practice [GS20; GW19]. Furthermore, a rich collection of open-source C SPLs is available, ranging from a few to millions of lines of code [Lie+10]. Considering that C is an inherently unsafe programming language [Aho+07; GG12; Li+24; Wag+00; Yam15], we also believe that mechanisms for detecting vulnerabilities in corresponding systems can be particularly helpful. Despite its popularity in the SPL domain, the C language itself lacks the support for software variability required for implementing SPLs [GW19]. Thus, developers typically resort to using conditional compilation via the C preprocessor [Ape+13a; Lie+10; Sch+22]. This practice is generally criticized for its error-prone nature and adverse effect on the ease with which software can be comprehended, debugged, and maintained [Ape+13a; GW19]. However, given its lightweight, easy-to-learn, and easy-to-adopt nature [Ape+13a], it remains a popular implementation technique used among many industrial SPLs [Aba+17; AK09; Bra+13; DBW19; Lie+13]. Accordingly, to provide a meaningful analysis of C SPLs, our approach focuses on the analysis of source code containing preprocessor annotations. In this regard, two types of preprocessor usage can be distinguished [Ape+13a; LKA11]. Disciplined preprocessor annotations respect the structure of the embedding (i.e., host) language and wrap entire language constructs, such as entire statements or functions [Ape+13a; LKA11]. Undisciplined annotations, on the other hand, wrap an arbitrary number of consecutive tokens without respecting the syntax of the host language [LKA11]. Limiting the analysis to source code using only disciplined annotations would simplify essential tasks, such as parsing, since disciplined annotations can be mapped directly to entire subtrees in a program's AST [Ape+13a]. However, considering that undisciplined annotations represent a considerable share of all preprocessor annotations in many software projects [LKA11], this would pose a major threat to the effectiveness of our approach on real-world SPLs. Consequently, we do not limit ourselves to a particular preprocessor usage pattern and aim to support both disciplined and undisciplined annotations.

Static Source Code Analysis

While numerous off-the-shelf SAST tools are available today, especially sophisticated taint-style vulnerabilities can be modeled comprehensively using Q-SAST tools. To add to their benefits, Q-SAST tools facilitate knowledge sharing, enable systematic reuse, and were found to outperform NQ-SAST (i.e., non query-based) tools in terms of vulnerabilities detected [Li+24]. Moreover, they enable timely combat of diverse and emerging vulnerabilities, considering that not only developers and maintainers but also the community can contribute queries [Li+24]. Consequently, for our objective of providing a family-based analysis approach for vulnerability discovery, we focus on lifting a Q-SAST tool to the domain of SPLs. Q-SAST tools typically center around a query-friendly source code representation, on which queries can be executed. Code Property Graphs (CPGs) represent one such representation of source code that can be efficiently queried for patterns, such as ones modeling sophisticated vulnerabilities, using graph traversals. The idea of a CPG is to combine three classic source code representations, commonly found in the compiler domain [Aho+07], into one joint structure. To this end, a program's CPG combines its AST, Control Flow Graph (CFG), and Program Dependence Graph (PDG) into a single graph with appropriately labeled edges [Yam+14]. Given its expressive nature, numerous studies [BD23; Cao+24; Du+20; GS19; GS20; Hao+21; JDL19; Shi+22; Zho+21] have adopted this representation. We follow this trend and focus on a tool built around CPGs. Specifically, we choose JOERN [24h], a versatile and well-established Q-SAST tool originally introduced by Yamaguchi et al. [Yam+14] in 2014. As opposed to many other popular Q-SAST tools [Li+24], JOERN's implementation is entirely open-source. For the design of our family-based analysis approach, this enables us to explore even lifting strategies that would otherwise be restricted by limited access to the tool's internals. As an added benefit, JOERN already supports the analysis of multiple popular programming languages besides C [24h]. It therefore allows our analysis approach to be expanded to other programming languages in the future.

2. Background

In this chapter, we provide an overview of fundamental concepts relating to three fields. First, Section 2.1 describes the concept of SPLs, including the associated terminology, advantages, implementation strategies, and analysis. Second, Section 2.2 outlines the idea behind conditional compilation as realized through preprocessors like the C preprocessor CPP. Lastly, Section 2.3 describes query-based vulnerability discovery based on the concept of CPGs as one form of static source code analysis.

2.1 Software Product Lines

An SPL is a family of related software *products* (also referred to as *variants* [Lie+10]) that rely on a common code base and are distinguished by the features (i.e., end-user-visible behavior [Ape+11; Cas+21; SRS13]) they provide [Ape+13b; Sch+22; Thü+14]. Accordingly, the implementation of an SPL comprises two parts. The *common core* encapsulates the functionality shared by all products [Aba+17]. Additionally, a set of *features* encapsulates the individual increments in functionality that are visible to stakeholders [SRS13] and specific to certain products. Selecting a set of features creates a *configuration* [Aba+17; Bra+12; Pet+19; Thü+14], which can be seen as a specification of a potential product [Cla+10]. The size of the configuration space (i.e., set of possible configurations) may be exponential in the number of features [Ape+13b; DBW19; Thü+14]. However, usually not all configurations in this space specify meaningful and desired products [Ape+13a; Thü+12; Thü+14]. For instance, features relating to the target platform of a product can be mutually exclusive, meaning that only a single platform (such as Windows or Linux) can be selected simultaneously [Thü+12]. Thus, a *variability model* typically introduces constraints over the features of an SPL, defining the set of valid configurations [Aba+17; Thü+14; von+16] and, consequently, the actual products of the SPL [Ape+13a; Pet+19]. Variability models can be expressed in various forms. Most popular are *feature models* that document the features of an SPL and their relationships [Ape+13a]. Feature models are frequently represented graphically using *feature diagrams* [KA08; Kui+22; Pet+23]. As an example, Figure 2.1 illustrates the feature diagram of a small SPL formed around the `foo` function of our running example (cf. Listing 1.1). It shows the optional `PROCESS_INPUT` feature of the `foo` function, together with two mutually exclusive features (WINDOWS and LINUX) for the specification of the target platform and another optional feature (INCLUDE_CHECK). Feature models are not bound to graphical representations and can also be represented textually, such as in the form of a KCONFIG model [ABW14; Oh+19]. In this regard, KCONFIG is a tool for managing the variability of an SPL [Oh+21]. It was originally developed for the LINUX kernel [24o] but has since also been adopted by other systems [Aba+17; Lie+13; Oh+19; Oh+21].

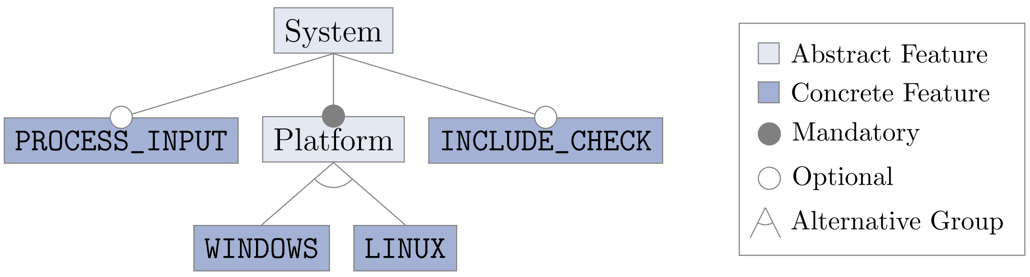


Figure 2.1: An example of a feature diagram specifying the variability model of a small SPL

Advantages

In general, SPLs serve the trend towards mass customization in the modern economy [Ape+13a; DBW19] and have become commonplace in many fields ranging from system-level software to web solutions [DBW19]. Although the upfront engineering effort required for the creation of an SPL is higher than for the creation of just a single software product, the effort for deriving products from the SPL is comparatively small [Ape+13a]. If many similar products need to be created (e.g., by tailoring products towards individual customers), this can result in significant cost savings compared to designing and developing each product from the ground up [Ape+13a; Cla+10; Mar+13]. Additionally, due to the systematic reuse of the common core and shared features, SPLs can reduce the time to market and allow for quick reactions to changing market conditions [Ape+13a; Mar+13; SRS13].

Implementation Strategies

Considering the implementation of SPLs in practice, three distinct strategies can be distinguished [SRS13]. *Annotative* approaches (such as conditional compilation using a preprocessor) merge the code of all features into a coherent code base and annotate sections belonging to certain features or combinations thereof [Ape+13a; KAK08]. For product derivation, code not required for the specific feature selection is either discarded during compile-time or ignored during run-time [Ape+13a; Cas+21]. *Compositional* approaches (such as feature-oriented programming) separate the code of features or feature combinations into dedicated files, containers, or modules, which can then be composed to form concrete products [AK09; Ape+13a; Cas+21; SRS13]. Lastly, *transformational* approaches (such as delta-oriented programming) rely on base assets and a set of transformations on these base assets [Ape+13a; Cas+21]. Concrete products are derived by applying transformations modifying the base assets in a step-wise manner [Ape+13a; SRS13].

Analysis

As empirical evidence shows [Aba+17; ABW14; Mor+19], SPLs, just like any other software system, are prone to programming errors. However, since their implementation describes not one but a wide range of different software products, errors can be confined to select configurations. These errors are commonly referred to as *variability bugs* [Aba+17; ABW14; Mor+19]. Similarly, there can be vulnerabilities only present under certain feature configurations. We refer to such vulnerabilities as *VIVs*. Since the number of products derivable from an SPL may be exponential in the number of features [Ape+13b; DBW19; Thü+14], automated analysis for variability bugs and VIVs represents a challenging task. In this regard, two general analysis strategies are often distinguished. A *product-based* analysis derives individual products from an SPL and analyzes them individually using conventional analysis techniques [Thü+14]. This can involve all products derivable from the SPL (*exhaustive* product-based) or only a representative subset (*optimized* product-based) [Thü+14]. A *family-based* analysis, on the other hand, does not analyze individual products of an SPL. Instead, it tries to analyze an SPL's implementation still containing variability through dedicated analysis techniques [Ape+13b; Ios+17; Lie+13].

2.2 Preprocessor-Based Conditional Compilation

The central idea behind conditional compilation is to selectively remove certain source code fragments before compilation [Ape+13a]. This allows a single code base to be transformed into a wide range of programs, making conditional compilation a common mechanism for implementing variable software (cf. Section 2.1) [Ape+13a]. With programming languages like C or C++, this mechanism is typically realized using annotations of the C preprocessor CPP. CPP is a stand-alone tool that adds lightweight metaprogramming capabilities to C, allowing source code to be manipulated before compilation [Ape+13a; Lie+10; LKA11; Tar+12]. Besides enabling conditional compilation, CPP also provides file inclusion and textual (i.e., macro) substitution capabilities [Ape+13a; KR88; LKA11; Pat+22]. Focusing on the use case of conditional compilation, developers can annotate (i.e., wrap) sections of source code with dedicated directives, often simply referred to as `#ifdef`s [Ape+13a; LKA11]. These directives are associated with a condition in the form of a compile-time expression, whose evaluation by the preprocessor decides upon the inclusion of the annotated code fragment [Ape+13a; KR88; LKA11]. The condition is therefore often referred to as the *presence condition* of the annotated code [GG12; Ken+10].

Preprocessor Discipline

From a technical perspective, the preprocessor CPP is token-based, meaning that it operates only on individual tokens and is oblivious to the structure of the enclosing (i.e., host) programming language [Ape+13a; GG12; LKA11]. Accordingly, preprocessor conditionals are not part of the syntax of the host language [Mor+19], which allows them to enclose arbitrary tokens of the source code [GG12; KR88]. Two types of preprocessor usage can thus be distinguished based on their alignment with regard to the code structure of the host language [Ape+13a; LKA11]. *Disciplined* preprocessor annotations align with the syntax of the host language and span entire language constructs, such as entire statements, functions, type definitions, or elements within type definitions [Ape+13a; LKA11]. As a result, disciplined annotations can be mapped to subtrees of a program's AST [Ape+13a]. *Undisciplined* preprocessor annotations, on the other hand, disregard the syntax of the host language and span only a number of arbitrary consecutive tokens [Käs+11; LKA11].

Listing 2.1 illustrates the difference between disciplined and undisciplined preprocessor annotations. The disciplined annotations shown in Listing 2.1a span entire statements, while the undisciplined annotations shown in Listing 2.1b span only select tokens. Since undisciplined annotations can wrap arbitrary source code regions, they allow for fine-grained variability, avoiding redundancy among non-variable code fragments. For instance, the undisciplined annotations in Listing 2.1b can wrap only the code segments relating to the variable if-statement, while the disciplined annotations in Listing 2.1a are forced to repeat the assignments to variables `x` and `y` in order to wrap complete statements.

```
#ifdef CONFIG_INCLUDE_CHECK
  if(A){
    x = readX();
    y = readY();
  }
#else
  x = readX();
  y = readY();
#endif
```

(a) Disciplined preprocessor annotations

```
#ifdef CONFIG_INCLUDE_CHECK
  if(A){
#endif
    x = readX();
    y = readY();
#ifdef CONFIG_INCLUDE_CHECK
  }
#endif
```

(b) Undisciplined preprocessor annotations

Listing 2.1: Examples for disciplined and undisciplined preprocessor annotations

Although undisciplined annotations lead to more compact code, they quickly complicate comprehension and maintainability of the program [Ape+13a]. In addition, annotations at token-level are prone to introduce subtle, hard-to-find syntax errors and complicate many tasks, such as parsing or refactoring unprocessed code [Ape+13a; LKA11]. As a result, the use of undisciplined annotations is generally discouraged and even restricted in certain projects, such as the Linux kernel [Ape+13a]. Nonetheless, as Liebig et al. [LKA11] found out during their analysis of 40 C projects, undisciplined annotations remain common in many projects and, on average, account for approximately 16% of all preprocessor annotations. Except for ill-formed annotations (i.e., annotations that can cause syntax errors or have mismatching numbers of opening and closing directives), all undisciplined annotations can be transformed into disciplined form [GJ03; Ken+10; LKA11]. This can be achieved by expanding annotations until they wrap entire host language constructs [GJ03; LKA11] (cf. Listing 2.1). However, when encountering nested annotations, in the worst case, this can lead to an exponential number of code clones [Ken+10; LKA11].

2.3 Static Source Code Analysis

Complementing the field of dynamic testing, static testing allows artifacts to be tested without execution [Fel+16]. For source code, this enables a proactive approach, identifying flaws prior to execution, thereby minimizing both costs and impact [Bra+13; LL05; Sha+01]. Among the most prominent examples of such static testing approaches are static analysis and manual reviews [Fel+16]. Although manual reviews can be a powerful tool for identifying conceptual faults that cannot be detected automatically, they are usually time-consuming and associated with high costs [EL02; LL05; Shi+22]. In addition, the quality of manual reviews strongly depends on the expertise of the reviewers [EL02; LL05]. In contrast, static analysis provides an automated way of leveraging expert knowledge codified in dedicated tools and thus represents a valuable aid for large projects [EL02].

The field of static source code analysis is extensive, as it comprises various approaches with different objectives. For taint-style vulnerabilities in particular, query-based analysis approaches provide comprehensive and extensible means for modeling and detecting vulnerabilities [Li+24; Yam+14]. Query-based approaches typically involve parsing source code into query-friendly representations and enabling the community to contribute and exchange queries on these representations [Li+24]. Concrete implementations of this idea are referred to as so-called Q-SAST tools [Li+24]. In the following sections, we describe taint-style vulnerabilities (Section 2.3.1), introduce CPGs as one popular query-friendly source code representation (Section 2.3.2), and demonstrate how taint-style vulnerabilities can be identified in a CPG using graph traversal-based queries (Section 2.3.3). The `foo` function (cf. Listing 1.1) is used as a running example throughout these sections. Since conventional source code analysis takes place on concrete products rather than on SPLs, we assume that the product incorporating feature `PROCESS_INPUT` is selected for analysis (e.g., through favorable sampling). Listing 2.2 shows the code of this product, derived from the SPL by preprocessing the `foo` function with the `CONFIG_PROCESS_INPUT` macro defined.

```
void foo() {
    int x = source();
    if(x < MAX){
        int y = 0;
        y = 2 * x;
        sink(y);
    }
}
```

Listing 2.2: The `foo` function of the running example (cf. Listing 1.1) preprocessed with the `CONFIG_PROCESS_INPUT` macro defined

2.3.1 Taint-Style Vulnerabilities

Taint-style vulnerabilities refer to software flaws regarding the propagation of data through a program as identified by taint analysis [Yam+15]. One objective of taint analysis is to identify data flows exhibiting the following three characteristics [Li+24; Yam+15]: (1) data is read from a potentially attacker-controlled source, (2) the data read from the source is not sanitized, and (3) the unsanitized data is passed to a sensitive sink. Data flows exhibiting these characteristics indicate a taint-style vulnerability. As an example, consider the `foo` function (cf. Listing 2.2). Let us assume the `source` function reads data from a potentially user-controlled source (e.g., command line parameters, UI input fields, or network). Let us furthermore assume that the internals of the `sink` function use the passed argument `y` as part of a sensitive operation (e.g., as the number of bytes to copy between buffers or the row in a database). If the user-controlled (i.e., tainted) data is less than `MAX`, it is possible to track a data flow from its respective source through variable `x`, to variable `y`, and lastly to the sensitive sink. Since this flow only sanitizes the data with regard to values greater than or equal to `MAX`, malicious data (e.g., negative values) can still reach the sensitive `sink` function. Thus, all three characteristics outlined above are fulfilled, indicating that the `foo` function contains a taint-style vulnerability. Depending on the implementation of the `sink` function and the program's memory layout, a malicious user could exploit this vulnerability to crash the program or read arbitrary values from its memory.

Although not all vulnerabilities can be represented using a taint-style description of sources, sanitizers, and sinks [Li+24], a plethora of common flaws can be described in this manner [Yam+15]. These include buffer overflows, SQL injections, and missing authorization checks, among others [Yam+14; Yam+15]. Besides only manifesting itself for certain feature configurations, the well-known *Heartbleed* vulnerability of the OpenSSL cryptography library constituted a taint-style vulnerability [Yam+15]. Similar to the example involving the `foo` function, the Heartbleed vulnerability was caused by a function reading data from a potentially user-controlled source (a network stream) and passing it on to a sensitive sink (the `memcpy` function) without sanitization [Yam+15]. In practice, this allowed attackers to gain access to sensitive regions of heap memory, leaking information such as secret keys for certificates, login credentials, and emails [24n; Yam+15].

2.3.2 Code Property Graphs

Fundamentally, a CPG is a graph-based representation of source code, capturing both structural and behavioral properties. The main idea behind this representation is to combine the unique views provided by three classic program representations, commonly known from the compiler domain [Aho+07], into a joint structure [Yam+14]. To this end, a CPG captures a program's static structure as expressed through an AST, the underlying control-flow semantics as expressed through a CFG, and the exhibited control and data dependencies as expressed through a PDG. In the following, we describe the three program representations of AST, CFG, and PDG in more detail and discuss how they are integrated into a joint CPG. To exemplify the different representations, we use the `foo` function (cf. Listing 2.2).

Abstract Syntax Trees (ASTs)

In technical terms, an AST is a tree data structure that is typically generated by a parser during syntax analysis [Aho+07]. It captures the syntactic structure of a program [Aho+07] and exposes its composition through elements such as definitions, declarations, statements, and expressions [Shi+22; Yam+14]. Nodes of the tree represent constructs in the source program (e.g., operations) while the children of a node represent the components of a construct (e.g., operands of an operation) [Aho+07; Du+20; Yam+14]. In addition, the tree is ordered [Yam+14] to preserve the ordering of program elements with respect to the original program. As an example, Figure 2.2, shows the AST for the `foo` function (cf. Listing 2.2). What differentiates ASTs from parse trees (i.e., concrete syntax trees) is that they do not represent the concrete syntax of the program as derived from a grammar [Aho+07; Yam+14]. Considering an if-statement such as the one extending from line 3 to line 7 of Listing 2.2, a corresponding parse tree would contain child nodes for the `if`-keyword, the opening and closing parenthesis, the guarding predicate, and the guarded statement. An AST, on the other hand, would omit details of program formulation, such as the child nodes for the keyword and the parentheses (cf. Figure 2.2), as they have no effect on the semantics of the program [Yam15]. Besides playing an important role in the front-end of a classic compiler architecture [Aho+07], ASTs serve as the foundation for many static analyses and code representations alike [Lie+13; von+18; Yam+14]. Their expressiveness is, however, limited given that control flow and data dependencies are not made explicit [Shi+22; Yam+14].

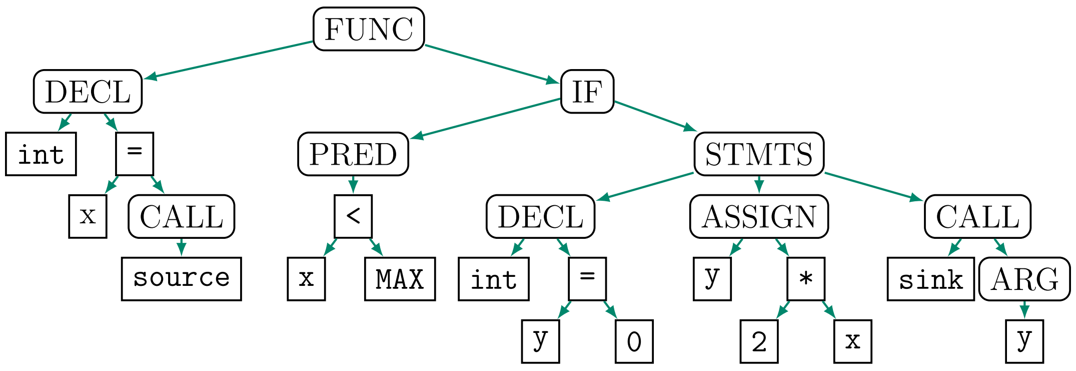


Figure 2.2: The AST for the preprocessed `foo` function (cf. Listing 2.2)

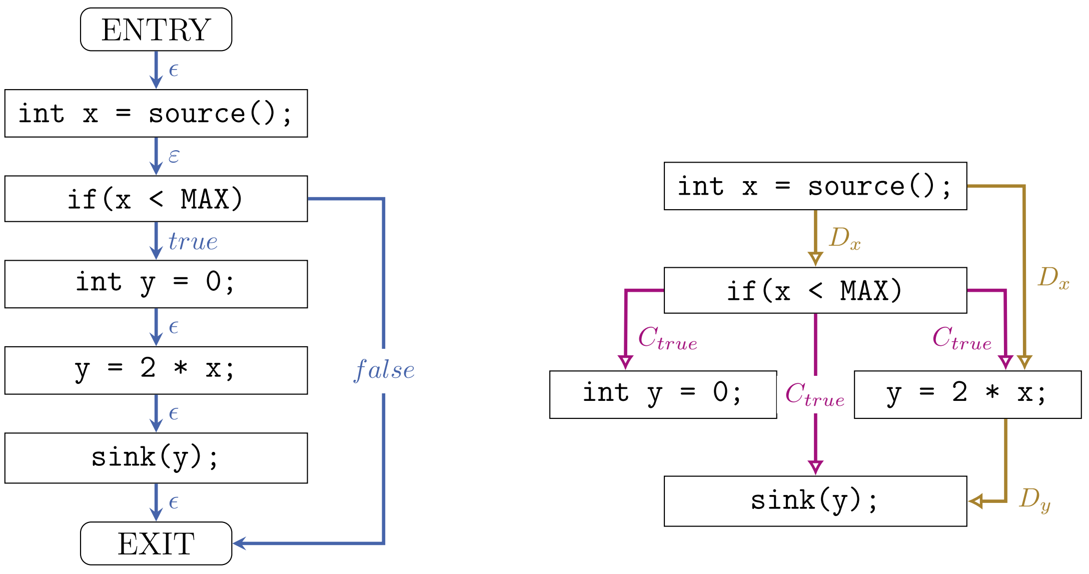
Control Flow Graphs (CFGs)

A CFG is a graph data structure that can be derived from a program's AST [Yam+14] and describes the possible flow of control through the program [Aho+07]. More specifically, a CFG is a directed graph over a program's statements and the conditions deciding upon their execution [Yam+14]. Accordingly, individual statements and predicates represent the nodes of the graph, while the possible control flow is represented by directed edges [Bra+13; DBW19; Yam+14]. Given that nodes can possess multiple outgoing edges due to branching control flow, each edge is assigned a label of ϵ, *true*, or $false$ [Yam+14]. Trivially, statement nodes each have one outgoing edge labeled with ϵ [Yam+14], indicating that the control flow always continues along the edge.[1] Predicate nodes, on the other hand, possess two outgoing edges labeled *true* and $false$, indicating the possible flows of control incurred by the predicate evaluating to true and false, respectively [FOW87; Yam+14]. To define a clear start and end to the flow of control, two nodes that do not correspond to any source code regions are typically added to a CFG [Aho+07; FOW87; Muc97]. The *Entry* node is connected to the first executable node of the graph via an ϵ-edge. Additionally, the *Exit* node receives outgoing edges from all nodes at which the flow of control can terminate. To illustrate this code representation, Figure 2.3a shows the CFG for the `foo` function (cf. Listing 2.2). In the compiler domain, CFGs play a vital role in data-flow analyses, such as reaching definitions [Muc97]. However, information about a program's possible control flow has also proven useful in other fields, such as security [Yam+14]. Nonetheless, as CFGs do not provide information about data dependencies, tracking user-controlled (i.e., tainted) data is not straightforward [Yam+14].

Program Dependence Graphs (PDGs)

Similar to a CFG, a PDG is a directed graph data structure over a program's statements and predicates [FOW87]. The fundamental idea of this graph is to make

[1]In the compiler domain, nodes connected by ϵ-edges are typically collapsed to so-called basic blocks [Aho+07; Muc97]. Although this simplifies the CFG, it complicates its mapping to the CPG.



(a) Control Flow Graph (CFG) (b) Program Dependence Graph (PDG)

Figure 2.3: The CFG and PDG for the preprocessed `foo` function (cf. Listing 2.2)

both the data and control dependencies of a program explicit [FOW87]. Accordingly, a PDG contains two types of edges [FOW87; Yam+14] that can be derived from a program's CFG [Yam+14]. A data dependency edge between nodes signifies that the operation of the receiving node depends on a data value computed by the originating node [FOW87; Yam+14]. Conversely, a control dependency edge indicates that the execution of the receiving node depends on the outcome of the originating predicate [FOW87; Yam+14]. Figure 2.3b shows the PDG for the `foo` function (cf. Listing 2.2). Data dependency edges are shown in brown and are labeled with $D_{\texttt{x}}$ to indicate a data dependency on the value of a variable `x`. Control dependency edges are shown in purple and labeled C_{true} and C_{false} to indicate the cases in which the predicate evaluates true and false, respectively. Historically, PDGs have been utilized to speed up traditional code optimizations [FOW87]. Another context in which PDGs are useful is program slicing [FOW87], i.e., determining the set of statements that may influence a particular statement [Wei81].[2] A program slice can be obtained simply by traversing the PDG backwards from the statement of interest and collecting the visited nodes [FOW87]. Even though a PDG makes dependencies between statements and predicates explicit, it does not encode the order in which statements are executed [Yam+14].

Code Property Graphs (CPGs)

As the name implies, CPGs are built on the formalism of *property graphs* [RN10]. A property graph $G = (V, E, \lambda, \mu)$ is defined as a 4-tuple characterizing a directed, edge-labeled, and attributed multigraph [Ang18; RN10; Yam+14]. In this structure, V represents the set of nodes and $E \subseteq (V \times V)$ represents the set of directed edges between nodes. The edge labeling function $\lambda : E \to \Sigma$ assigns a label from an alphabet Σ to each edge. Additionally, the function $\mu : (V \cup E) \times K \to S$ assigns

[2]For brevity, the description refers only to backward slicing. PDGs can just as well be used for forward slicing, i.e., determining the set of statements that may be influenced by a particular statement.

property values from S to nodes and edges under specific property keys from K. In this regard, S represents an infinite set of property values, while K represents an infinite set of property keys [Ang18]. Since a single property graph can model various types of relationships using typed and labeled edges [RN10], the fundamental idea of a CPG, as originally introduced by Yamaguchi et al. [Yam+14], is to integrate AST, CFG, and PDG into a unified property graph with appropriately labeled edges. To this end, the three representations are first represented as individual property graphs and then merged into a joint CPG [Yam+14].

An AST can be represented as a property graph $G_{AST} = (V_{AST}, E_{AST}, \lambda_{AST}, \mu_{AST})$, with V_{AST} representing the nodes of the tree, E_{AST} representing the edges of the tree, and the function λ_{AST} labeling the edges in E_{AST} as AST edges. Furthermore, the function μ_{AST} assigns properties *code* and *order* to each node in V_{AST} to indicate the represented source code construct and order within the tree, respectively. Similarly, a CFG can be represented as a property graph $G_{CFG} = (V_{CFG}, E_{CFG}, \lambda_{CFG}, \cdot)$ with V_{CFG} representing the nodes in the CFG (i.e., *Entry*, *Exit*, and the nodes in V_{AST} where *code* is set either to $STMT$ or $PRED$ representing statements and predicates, respectively) and E_{CFG} representing the edges in the CFG. Additionally, λ_{CFG} assigns a label out of $\Sigma_{CFG} = \{\varepsilon, true, false\}$ to each edge in E_{CFG} as in the conventional CFG, while the function μ_{CFG} is not needed. Lastly, a PDG can be represented as a property graph $G_{PDG} = (V_{CFG}, E_{PDG}, \lambda_{PDG}, \mu_{PDG})$. The set of nodes in this property graph is identical to the one of G_{CFG}, given that CFGs and PDGs share the same nodes. E_{PDG} contains all edges of the PDG and λ_{PDG} assigns a label out of $\{D, C\}$ to all edges in E_{PDG} to indicate data dependency and control dependency edges, respectively. The function μ_{PDG} assigns a property *symbol* to all data dependency edges, indicating the relevant symbol, and a property *condition* to all control dependency edges, indicating the relevant evaluation of the originating predicate.

Drawing from the definition provided by Yamaguchi et al. [Yam+14], the three property graphs G_{AST}, G_{CFG}, and G_{PDG} can be combined into a corresponding CPG $G = (V, E, \lambda, \mu)$ as defined below:

$$V = V_{AST} \cup \{Entry, Exit\} \qquad \lambda = \lambda_{AST} \cup \lambda_{CFG} \cup \lambda_{PDG}$$
$$E = E_{AST} \cup E_{CFG} \cup E_{PDG} \qquad \mu = \mu_{AST} \cup \mu_{PDG}$$

In the context of this definition, the union of two functions (e.g., $\lambda_{AST} \cup \lambda_{CFG}$) denotes a new function operating over the merged domains of the initial functions. Furthermore, for the sake of brevity, the definition focuses on intraprocedural CPGs, i.e., CPGs spanning only the length of a single function. However, similar to other program representations, CPGs can also cover multiple functions, making them interprocedural. In its simplest form, this can be achieved by joining the CPGs of individual functions through edges that connect the arguments of callers to the parameters of their callees and return statements back to their call sites [Yam+15].

Figure 2.4 shows the CPG for the `foo` function (cf. Listing 2.2). Edge labels set by the λ function are shown on the respective edge. Additionally, the properties *symbol* and *condition* are shown as indices to the corresponding edge labels, while the *code* property is shown inside the AST nodes. For clarity, the labels applied to the AST edges by λ_{AST} are omitted. Furthermore, the *order* property set by μ_{AST} is not explicitly shown but represented by the order of the AST children starting from the left.

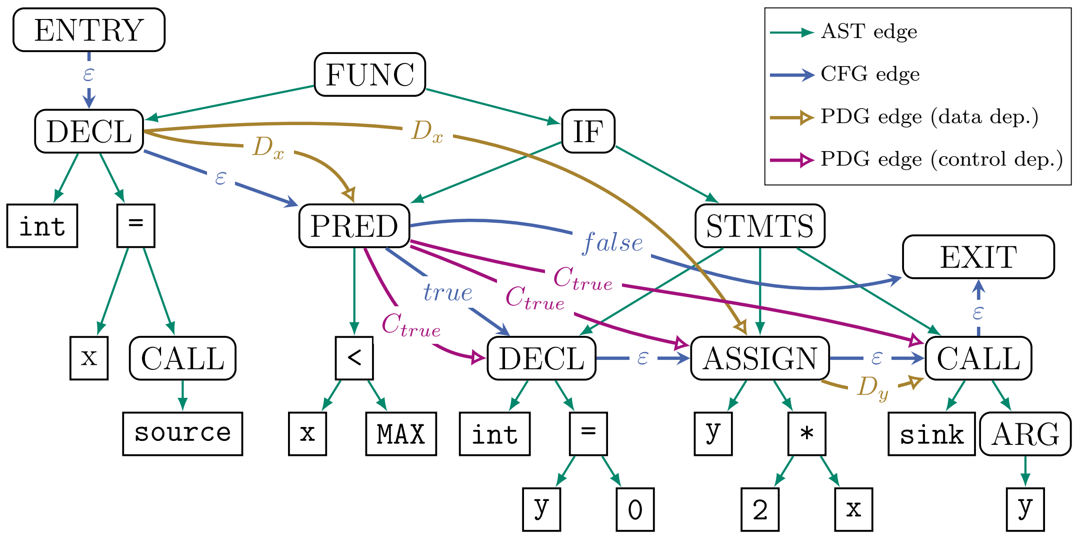


Figure 2.4: The CPG for the preprocessed `foo` function (cf. Listing 2.2)

2.3.3 Query-Based Vulnerability Discovery

The fundamental idea of query-based vulnerability analysis is to transform source code into representations that enable convenient and efficient detection of vulnerability patterns through queries [Li+24]. Property graphs (cf. Section 2.3.2), supported by many popular graph databases [Ang18; Yam+14; Yam+15], serve as one such representation. To extract information from property graphs, users typically interact with graph databases via *graph traversals* that navigate through the graph based on its labels and properties [RN10; Yam+14]. Using traversals, vulnerability patterns can be codified in a comprehensive way. This not only facilitates knowledge sharing by allowing experts to incorporate domain knowledge and historical data, but also enables systematic reuse across different software systems [Li+24; Zho+21]. Graph traversals are often specified using dedicated query languages like GREMLIN.[3] However, given that these languages vary in their syntax and are not universally supported by all graph databases alike, in the following we focus on a generic definition and syntax introduced by Yamaguchi et al. [Yam+14]. Given a property graph $G = (V, E, \lambda, \mu)$, a graph traversal is a function $\mathcal{T} : \mathcal{P}(V) \to \mathcal{P}(V)$ that maps one set of nodes to another, both represented as elements of the power set $\mathcal{P}(V)$ [Yam+14].[4] To enable more sophisticated traversals while maintaining comprehensibility, two traversals $\mathcal{T}_0$ and $\mathcal{T}_1$ can be chained together to $\mathcal{T}_0 \circ \mathcal{T}_1$ using conventional function composition [RN10; Yam+14].

Using a CPG as the property graph, graph traversals make it possible to efficiently query for many common vulnerabilities. Since a CPG contains all information of the corresponding AST, CFG, and PDG, patterns with regard to syntax-related, control flow-related, or data flow-related vulnerabilities can be modeled by considering only the corresponding subgraphs. However, for modeling taint-style vulnerabilities (cf. Section 2.3.1), all three subgraphs need to be considered. For instance, identifying the vulnerability in the `foo` function (cf. Listing 2.2) involves considering the calls to `source` and `sink`, the possible control flow executing the assignments to `y` and

[3]https://tinkerpop.apache.org/gremlin.html.

[4]If the sets of nodes should be allowed to contain duplicates (e.g., to weight or rank results), traversals can be defined over power multisets instead of conventional power sets [RN10].

the call to `source`, as well as the data dependencies between `x` and `y`. Additionally, the absence of proper sanitization of the data has to be considered. All this can be achieved using the high-level graph traversal shown below:

$$\textsc{Match}_p \circ \textsc{Unsanitized}_{\{\mathcal{T}_s\}} \circ \mathrm{ARG}^1_{\texttt{sink}}$$

The $\mathrm{ARG}^1_{\texttt{sink}}$ traversal first identifies nodes corresponding to the first arguments passed to the `sink` function. Next, the $\textsc{Unsanitized}_{\{\mathcal{T}_s\}}$ traversal collects all nodes representing statements that produce data for these arguments. Additionally, the traversal only keeps nodes whose produced data does not undergo proper sanitization as described by the traversal $\mathcal{T}_s$ (i.e., comparisons for greater than or equal to zero and less than `MAX`). Finally, $\textsc{Match}_p$ gathers all child nodes of the data-producing statements and selects those satisfying the predicate p, which returns *true* if a node represents a call to the `source` function. For brevity, we omit the formal definitions of the individual traversals used. A detailed description of the composition of sophisticated traversals can be found in the publication of Yamaguchi et al. [Yam+14].

3. Comparison of Common Lifting Strategies

Conventional (i.e., variability-oblivious) SAST tools operate on single products and cannot fully cope with the variability introduced into source code by preprocessor annotations (cf. Section 1.1). To overcome this limitation, two general strategies for lifting a conventional SAST tool to the domain of SPLs can be distinguished [Ios+17; Pat23; Thü+12; von+16]:

(1) **Lifting by Extension**: Lift the entire SAST tool by extending its internals (e.g., parsing, data structures) to support immediate processing of an SPL in its entirety rather than just its individual software products.
(2) **Lifting by Simulation**: Do not lift the SAST tool itself but the variability of the SPL by means of a preprocessing step that transforms the compile-time variability of the SPL into run-time variability. The SAST tool can then operate on the transformed source code without the need for modification.

Since Q-SAST tools represent a special class of SAST tools, both strategies can be applied towards our goal of lifting the off-the-shelf Q-SAST tool Joern. Accordingly, Section 3.1 first outlines the general structure of a Q-SAST tool. Sections 3.2 and 3.3 then describe lifting by extension and lifting by simulation in more detail and outline how the strategies can be applied in the context of a Q-SAST tool, such as Joern. Lastly, Section 3.4 discusses the strengths and weaknesses of both strategies and explains the rationale behind our choice.

3.1 Query-Based Static Application Security Testing

As described in Section 2.3, the fundamental concept of a Q-SAST tool revolves around parsing source code into query-friendly representations and enabling the community to contribute and exchange queries modeling issues (e.g., vulnerabilities) within these representations. From a technical perspective, Q-SAST tools typically comprise three distinct components [Li+24]: (1) a query-friendly source code representation, along with its parsing infrastructure and database storage, (2) a query

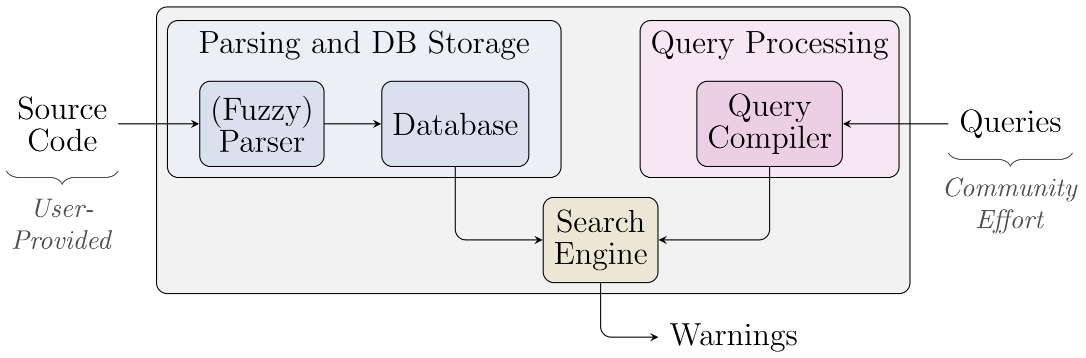


Figure 3.1: Typical structure of a Q-SAST tool as described by Li et al. [Li+24]

language with its compilation toolchain, and (3) a search engine. The structure resulting from these three overarching components is illustrated in Figure 3.1.

As input, a Q-SAST tool accepts source code that a user wishes to analyze. Additionally, it receives a number of queries that control the analysis. A common strategy is to reuse and possibly adapt queries from a community-maintained query database [Li+24]. For the analysis, the tool parses the provided source code into a source code representation (e.g., a CPG in the case of JOERN), which is typically maintained in a dedicated database [Li+24]. To enable the analysis of code without a working build environment (e.g., incomplete code), a Q-SAST tool may also employ approximate (fuzzy) parsing [Li+24; Yam15]. In addition to the creation and storage of the source code representation, the tool compiles the queries that should be used for the analysis into a suitable format. The search engine then executes the compiled queries on the source code representation stored in the database and reports the found matches as warnings.

3.2 Lifting by Extension

An intuitive strategy for lifting a variability-oblivious Q-SAST tool is to extend its internals to support processing of whole SPLs. We refer to this strategy as *lifting by extension*. Extending a tool's internals enables it to receive the preprocessor-annotated code of an SPL as input and to produce vulnerability warnings relating to specific configurations as output. While some of the tool's components and inputs can remain unmodified to achieve this functionality, others must be adjusted. Figure 3.2 depicts the general structure of a Q-SAST tool (cf. Figure 3.1), emphasizing the extent to which the individual components and inputs require alterations. For a more detailed discussion of the individual components, in the following, we focus on the three overarching components of source code representation, query language, and search engine identified by Li et al. [Li+24]. These components are indicated by 1, 2, and 3 in Figure 3.2, respectively. Specifically, we detail the roles of the contained components, the challenges associated with accommodating variability, and necessary alterations to overcome these challenges.

3.2.1 Source Code Representation

In general, the automated analysis of source code hinges on the availability of one or more source code representations that expose certain properties of the analyzed

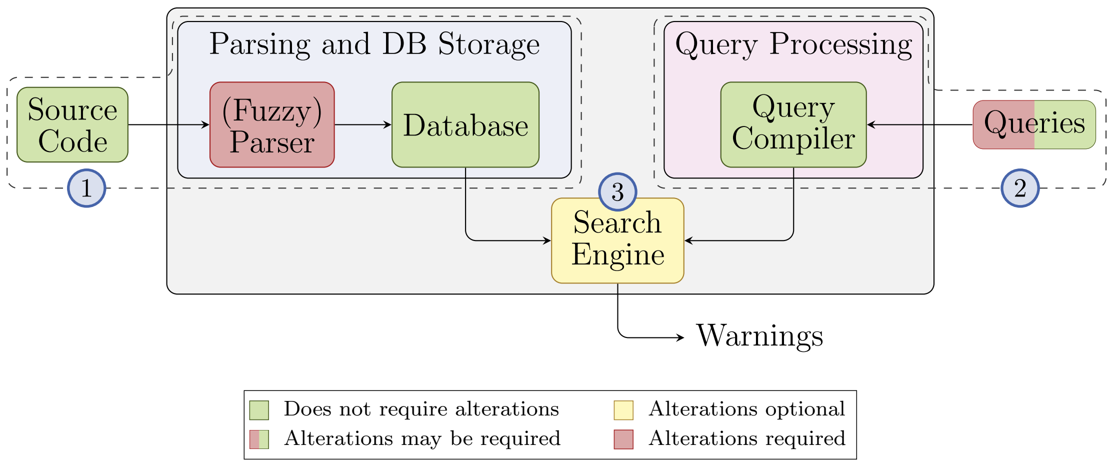


Figure 3.2: Structure of the lifting by extension strategy

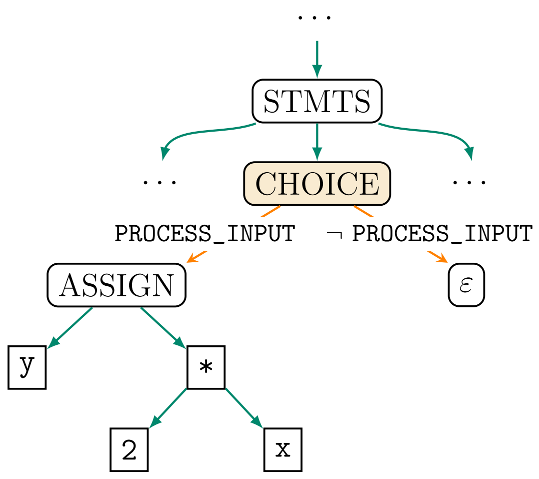
program [Yam15]. For a comprehensive analysis of an SPL, these representations have to consider variability. This means that the properties of every product derivable from the SPL have to be exposed by the representations [Lie+13; von+18]. Popular source code representations, such as conventional ASTs, CFGs, PDGs, or CPGs, only relate to the source code of a single product. Therefore, conventional representations are insufficient and need to be made variability-aware, taking into account all possible products of an SPL.

3.2.1.1 Variability-Aware Source Code Representation

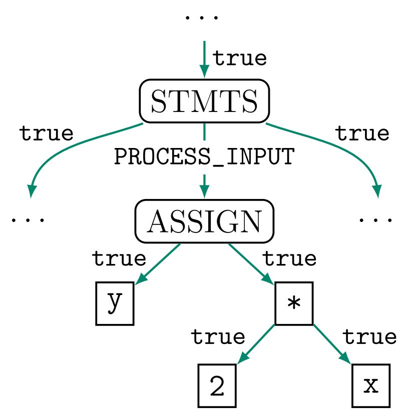
Before variable source code can be parsed and stored in a corresponding database, a variability-aware version of the conventional (i.e., variability-oblivious) source code representation used needs to be developed. To this end, there are multiple ways through which variability can be incorporated into a source code representation. Evidently, these ways vary depending on the concrete form of source code representation considered. Given the query-friendly nature of graph-based representations and the resulting popularity among prevalent Q-SAST tools [Li+24], in the following we focus on incorporating variability into this form of representation. Additionally, while others [Wal+14] considered variability-aware representations on a level close to implementation, we focus on a conceptual examination. Specifically, we concentrate on the two common techniques of incorporating special choice nodes into the representation and annotating edges with dedicated presence condition labels.

Choice Nodes

One solution for incorporating variability into a graph-based source code representation is to introduce dedicated nodes (typically referred to as *choice* nodes [Lie+13; von+18]). In essence, these nodes correspond to preprocessor conditionals and indicate variable sections of the representation through outgoing edges [von+18]. For instance, in the context of ASTs, choice nodes indicate choices between two or more alternative subtrees [Lie+13; von+18]. This is illustrated in Figure 3.3a, which shows the partial variability-aware AST for the `foo` function of the running example (cf. Listing 1.1). Since inclusion of the second assignment to `y` is subject to selection of the `PROCESS_INPUT` feature, a choice node selects between the assignment (`PROCESS_INPUT` selected) and an empty statement (`PROCESS_INPUT` unselected).



(a) Using a choice node



(b) Using presence condition labels

Figure 3.3: Variability-aware AST subtrees for the body of the if statement of the `foo` function (cf. Listing 1.1)

Presence Condition Labels

An alternative to the usage of choice nodes is to connect all variable sections to the rest of the representation directly. This avoids the introduction of artificial nodes and thus leads to a more compact representation. It does, however, necessitate labeling each edge with its corresponding presence condition to indicate the configurations under which the relationship is valid. For edges between nodes, whose associated SPL fragments require their mutual inclusion, the presence condition is trivially `true`. All remaining edges are labeled with the condition under which the relationship (structural inclusion, control flow, etc.) is present in products derived from the SPL. Figure 3.3b illustrates this solution for the AST of the `foo` function (cf. Listing 1.1). Given that the assignment statement is only part of the compound statement if the `PROCESS_INPUT` feature is selected, the corresponding edge receives the label `PROCESS_INPUT`. All other edges are assigned the trivial presence condition, `true`, since for all products of the SPL, inclusion of the parent node always implies inclusion of the child node.

3.2.1.2 Parsing

Parsing aims at creating instances of a given source code representation. It thus constitutes an indispensable component of most Q-SAST tools (cf. Figure 3.2). However, parsing unpreprocessed source code into a variability-aware representation (cf. Section 3.2.1.1) is a non-trivial task that cannot be solved by standard parsers [Käs+11; Ken+10; Lie+13]. In essence, the reason for this lies in the fact that preprocessor annotations are not part of the C language [Mor+19]. As a result, unpreprocessed C does not conform to the formal C grammar [GJ03] (cf. Section 1.1). To address this challenge, the parsing performed by a Q-SAST tool needs to be adjusted. In this regard, there are two general directions. Using a naive solution, the existing parser can be reused. However, as a result, the resulting source code representation quickly becomes impractically large. More advanced solutions solve this problem. In return, they require profound changes to the existing parser. Both alternatives are described in more detail below.

Naive Variability-Aware Parsing

Targeting a representation using choice nodes (cf. Section 3.2.1.1), a trivial approach involves reusing the existing parser employed by the Q-SAST tool to build the variability-oblivious representation for every product of the SPL [GJ03]. In this regard, every possible product is first derived from the SPL to eliminate variability and subsequently parsed. The resulting representations can then be joined using a single choice node that selects between the representations of the individual products based on the associated feature configuration [Lie+13; von+18]. This solution is simple and requires only minor adjustments to the parsing performed by a tool. However, it leads to redundancies for all elements shared between multiple products. Consequently, the variability-aware representation becomes unnecessarily large, rendering any analysis performed on it inefficient. Moreover, given the theoretically exponential number of products derivable from an SPL [Lie+13; Thü+14; von+18], creating a separate representation for every product leads to prohibitive costs.

Advanced Variability-Aware Parsing Solutions

Besides the aforementioned trivial strategy, there are two advanced solutions that were proposed as part of prior research on parsing unpreprocessed C [GG12; GJ03; Ken+10]. The first solution is straightforward and involves extending the grammar of the host language (i.e., C) with dedicated productions for preprocessor conditionals [GJ03; Ken+10; LKA11]. Based on such a modified grammar, a new parser can be generated by parser generators such as BISON.[1] The generated parser can then be used to build the corresponding variability-aware representation. An alternative strategy to changing the host language's grammar is to extend the parser of a tool to a fork-merge parser [GG12]. Upon encountering a preprocessor conditional, a fork-merge parser forks its state into subparsers that parse the code of the individual branches of the conditional [GG12]. When all subparsers reach the end of their branch, they are merged, and parsing continues as usual [GG12]. The source code representations created by the subparsers can then be incorporated into the overall representation by connecting them to choice nodes or using dedicated edges labeled with the corresponding presence conditions. While both solutions try to avoid redundancies and thus create more compact representations than the naive strategy, they evidently require deeper changes to the parsing performed by a Q-SAST tool.

3.2.1.3 Requirements Imposed by Parsing Variable Source Code

Depending on the used parsing solution (cf. Section 3.2.1.2), the variable source code of an SPL may need to conform to a number of requirements before it can be parsed into a variability-aware representation. Establishing any property on the input source code by hand can quickly become infeasible considering the size of real-world software. As a result, required properties are typically enforced through a preprocessing step of the variability-aware parser. This ensures that variable source code can be used as input without it having to undergo alterations (cf. Figure 3.2).

The trivial parsing technique of using a single choice node to select between individual variability-oblivious representations removes all variability prior to parsing. It does therefore not impose any special requirements on an SPL's source code. More

[1]https://www.gnu.org/software/bison/.

sophisticated solutions, such as an extended grammar or a fork-merge parser, often impose two requirements [GG12; Ken+10]: (1) preprocessor conditionals must be in disciplined form and (2) macro substitutions and file inclusions have to be already resolved. Below, we describe these requirements in more detail.

Preprocessor Discipline

First, to enable a clear mapping from variational sections of source code to elements of the target representation, preprocessor conditionals must be in disciplined form [Ken+10; LKA11]. In most software projects, the vast majority of preprocessor annotations is already in disciplined form [LKA11]. The remaining annotations can generally be transformed into disciplined form, albeit in the worst case resulting in an exponential number of code clones [Ken+10; LKA11] (cf. Section 2.2). Alternatively, to avoid transformation of the source code, it is possible to parse each configuration of a structure containing an undisciplined annotation in isolation using dedicated subparsers [GG12].

Macro Substitutions and File Inclusions

Second, to avoid false parse errors, macro substitutions and file inclusions have to be resolved before parsing while keeping preprocessor conditionals (i.e., `#ifdefs`) intact [GG12; Ken+10]. This is usually realized by a configuration-preserving preprocessor (also referred to as a partial preprocessor [Ken+10]) [GG12]. Without prior expansion of macros, code relying on macro substitution for syntactic correctness can raise false syntax errors. Not expanding file inclusions can, additionally, withhold information from the parser that is indispensable for the syntactic correctness of the source file. Listing 3.1 exemplifies the need for these steps in the context of a small C program. To successfully parse the program, the `P` macro defined in Listing 3.1b has to be expanded. Without this expansion, Lines 4 and 5 in Listing 3.1a lack a terminating semicolon, thus preventing them from being parsed as the intended consecutive statements. Since the definition of the `P` macro is located in a separate header file, expanding the corresponding file inclusion is vital. Otherwise, the definition of the macro will remain unknown, making its substitution impossible.

```
1 #include "SimpleHeader.h"
2
3 void main() {
4     P("Hello\n")
5     P("World\n")
6 }
```

(a) The main program

```
1 // SimpleHeader.h
2 #include <stdio.h>
3
4 #define P(message) \
5     printf(message);
```

(b) The included header file

Listing 3.1: A simple C program inspired by the example of Kenner et al. [Ken+10] that can only be parsed after header inclusion and macro substitution has been completed

3.2.1.4 Database Persistency

Q-SAST tools typically maintain instances of their source code representations in dedicated databases [Li+24]. In this regard, graph databases provide a convenient and efficient way to store and query graph-based representations. Most of these databases support versatile graph data models, such as property graphs [Ang18]. Considering that these representations allow various types of relationships and nodes to be expressed within a single graph [RN10], storing a variability-aware representation requires no changes to the database component (cf. Figure 3.2).

However, depending on whether earlier stages of the tool under consideration already built and maintained the source code representation in a format supported by the corresponding graph database, a transformation into a suitable format might need to be established. For this purpose, the existing transformation for the variability-oblivious representation can be extended to handle constructs such as presence condition labels or choice nodes. Considering property graphs as the target format, this represents a simple task. Adding presence condition labels only requires adding a new property containing the presence condition to the corresponding edges. Similarly, to distinguish between conventional nodes and choice nodes, a new property can be added to nodes indicating their type.

3.2.2 Query Language

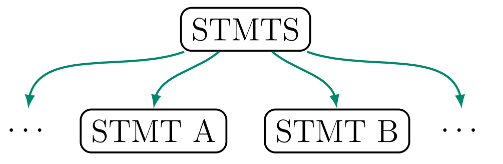
For the specification of queries, Q-SAST tools often make use of general-purpose query languages supported by the utilized database solution. For instance, the popular but partially closed-source Q-SAST tool CodeQL [24e] relies on queries written in a Datalog variant [Li+24]. Joern, on the other hand, based its queries on Gremlin but has since transitioned to the Scala-based Code Property Graph Query Language (CPGQL) supported by the employed graph database flatgraph[2] [24f; 24i; Yam+14]. Given their general-purpose nature, these query languages typically merely act as interfaces for providing access to the utilized source code representations. As a result, they generally do not pose limitations with regard to modeling patterns in variability-aware representations. The query language and associated compilation toolchain of a Q-SAST tool, therefore, require no modifications to extend the tool's applicability to the domain of SPLs (cf. Figure 3.2). Nonetheless, the internal structure of a variability-aware representation can differ significantly from representations based on single products. Therefore, the question arises to what extent users must take variability into account when specifying a query. In this respect, it is possible to distinguished between variability-oblivious and variability-aware queries.

Variability-Oblivious Queries

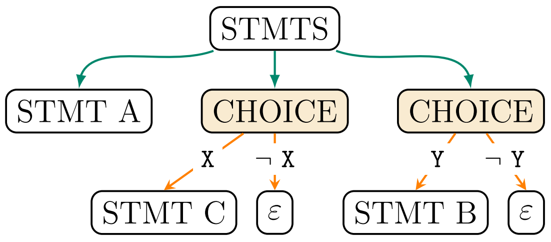
Variability-oblivious queries do not take variability into account when modeling specific source code patterns. As a result, their specification is simple, and they remain effective for the analysis of programs employing alternative patterns of variability. Although keeping queries oblivious to variability offers great benefits, it carries a risk of missing a considerable number of valid matches. This risk is exemplified by Figure 3.4. Assuming that the consecutive execution of statements of types A and B introduces a vulnerability, a query matching simple instances of this pattern can be formulated on ASTs. The query searches the AST for nodes representing a statement list (denoted by STMTS) and checks the associated child

[2]https://github.com/joernio/flatgraph.

nodes for consecutive occurrences of statements of types A and B. The AST pattern matched by this query is illustrated in Figure 3.4a. Considering a variability-aware AST of an SPL, such as the one partially illustrated in Figure 3.4b, this query leads to missed matches.[3] Due to the presence of the variable statement C, the statements of types A and B are not consecutive child nodes of the corresponding statement list. Additionally, the statement of type B is variable itself and thus not a direct child of the statement list. As a result, the query is unable to match an occurrence of the pattern, despite the fact that the configuration $\neg \mathtt{X} \wedge \mathtt{Y}$ would lead to a manifestation. To detect this manifestation, the query needs to account for variability, i.e., it needs to be variability-aware.



(a) AST pattern matched by the query



(b) Variability-aware AST

Figure 3.4: An AST pattern matched by a variability-oblivious query and a possible variability-aware AST exhibiting the pattern

Variability-Aware Queries

Variability-aware queries take variable structures into account when modeling particular source code patterns. This allows patterns to be matched, even when being affected by variability structures. For instance, using a variability-aware query, it is possible to match the consecutive execution of statements of type A and B in the variability-aware AST depicted in Figure 3.4b. To this end, the query has to allow for an optional choice node in between the two statements. If this choice node is present, the query must ensure that there is at least one path from this node to a leaf node representing the empty statement ε. Additionally, the query cannot simply rely on the two statements being direct child nodes of the statement list, as they can both be subject to variability. The query therefore additionally has to allow for choice nodes in between the statements and the statement list.

In general, variability-aware queries not only lead to high precision but also simplify the task of the search engine, given that they closely describe the structures to be searched within the source code representation. Nonetheless, they also require users to be aware of the usage of variability in the analyzed code base when specifying queries. For large programs encompassing millions of lines of code, this is obviously challenging. Furthermore, making variability explicit requires queries to be in line with the variability patterns of a given code base. This, in turn, can reduce the effectiveness of reusing queries for the analysis of other programs.

[3]For brevity, we focus on a variability-aware AST realized using choice nodes. The same effect can, however, also be observed for variability-aware ASTs realized using presence condition labels on its edges.

Practical Considerations

As previously described, variability-oblivious queries can result in missing obvious matches to the modeled source code patterns. Variability-aware queries are not a perfect solution either. Not only are they harder to specify, but they cannot easily be reused in the context of other systems. Therefore, in practice, the correct form has to be chosen depending on the modeled pattern. Queries modeling source code patterns that are not affected by variability (e.g., the presence of a call to a specific function) can be kept variability-oblivious. Queries that model patterns prone to variability interactions (cf. Figure 3.4) should, on the other hand, be specified variability-aware to provide meaningful results. When trying to reuse existing queries specified for the analysis of non-variable software, this means that queries need to be inspected and adjusted accordingly (cf. Figure 3.2).

3.2.3 Search Engine

The search engine of a Q-SAST tool is responsible for scanning the constructed source code representation for matches to the vulnerability patterns modeled by the provided queries [Li+24]. While the database containing the source code representation usually does not have to be rebuilt in between queries of an analysis run [Li+24], the search engine is executed for every query. The performance of a Q-SAST tool therefore heavily depends on its search engine. As a result, it is not surprising that tools, such as JOERN, rely on the built-in solution of the database technology used [24i]. While this presents a way that is less labor-intensive than implementing a dedicated search engine from scratch, it makes the internals difficult to access and modify.

Fortunately, in the context of popular graph-based representations and corresponding graph databases, introducing variability into the search engine does not necessarily require changes to its internals. The query languages used by popular Q-SAST tools are typically expressive enough to capture variability within their queries, if needed (cf. Section 3.2.2). Accordingly, the query language itself does not require changes and the underlying structure of the queries passed on to the search engine remains unchanged. A similar pattern can be observed for incorporating variability into the used source code representation. While the representation itself requires changes (e.g., in the form of choice nodes or dedicated edges with presence condition labels), the data models used by graph databases are typically expressive enough to incorporate changes without the need for adjustment (cf. Sections 3.2.1.1 and 3.2.1.4). As the underlying structure of the data contained within the database remains unchanged, the search engine is able to operate on variability-aware representations without the need for modifications.

Optional Improvements

As described above, changes to the search engine can be difficult to realize and are not necessarily required to support the analysis of SPLs. However, they would offer an opportunity to address certain problems arising from the use of variability-oblivious queries on variability-aware representations (cf. Section 3.2.2).

By leaving the search engine unchanged, it is the user's responsibility to ensure that queries whose pattern can be affected by variability incorporate the right degree of variability-awareness. If a user is unaware of this, matches for a query can be missed (cf. Figure 3.4), leaving potential vulnerabilities undiscovered. To counteract simple instances of this problem, the internals of the search engine could be altered. Focusing on an implementation using choice nodes for the sake of brevity, this could look as follows: If a choice node is encountered during the matching process, the search engine automatically descends from the choice node, searching for either an ε node or the next structure expected by the query. Once an ε node or the next expected structure is found, the search engine ascends again and continues matching as before. Otherwise, the matching process at the current position is aborted. To exemplify this strategy, consider the example in Figure 3.5, where the goal is again to identify a consecutive execution of statements of types A and B. After matching the node of statement A, the search engine would descend into the choice node parenting statement C. Since this leads to the discovery of an ε node, the search engine would ascend again and follow the same procedure for the choice node parenting statement B. Upon discovering the node of statement B, the searched pattern would be completed and the match returned.

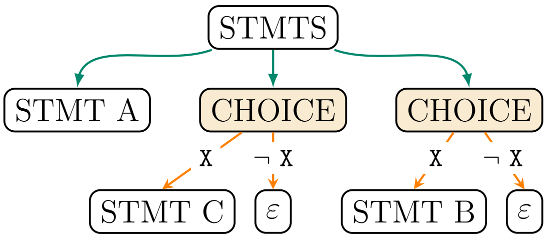


Figure 3.5: An adjusted version of the variability-aware AST of Figure 3.4, where statement A is directly succeeded by statement B only in an invalid product

The simple strategy outlined above can help prevent missed matches to a query. However, its disregard for the presence conditions associated with the matched structures can lead to matches relating to invalid products of the SPL. This is again exemplified in Figure 3.5. While the strategy would report a match to a query modeling the consecutive execution of statements A and B, this match would relate to the configuration $\neg X \wedge X$. This configuration constitutes a logical contradiction and thus describes an invalid product of the SPL. Matches relating to invalid products can be filtered out after the analysis. As a more efficient solution, it is also possible to employ satisfiability checks within the search engine to detect invalid feature configurations at an early stage. For this purpose, the presence condition encountered when traversing a new choice node can be joined with the ones collected earlier during the matching attempt via a logical conjunction. The resulting formula can then be passed on to a corresponding solver. If the formula is not satisfiable, the current (partial) match relates to an invalid product. In this case, the search engine would need to either discard the match entirely or backtrack, removing corresponding parts from the formula until it is satisfiable again. For the example in Figure 3.5, this would mean that upon encountering the node of statement B, the formula would become $\neg X \wedge X$. As this formula is not satisfiable, the search engine would be forced to ignore the match.

3.3 Lifting by Simulation

Another strategy for lifting a variability-oblivious Q-SAST tool to the domain of SPLs is to transform problematic (i.e., variable) input into a form with which the tool can operate. Figure 3.6 depicts the structure of this strategy, which we refer to as *lifting by simulation*. Contrary to lifting by extension, the goal is not to consider the internals of the chosen Q-SAST tool (cf. Section 3.1) but to treat it as a black box that can be used without requiring alterations. To achieve this goal, two components are introduced as additional pre- and post-processing steps surrounding the tool. The *variability encoding* component builds on the identically named concept [Ape+11; von+16] and is responsible for transforming variable source code into a so-called *product simulator* (i.e., non-variable C) before its analysis by the Q-SAST tool. While variability encoding enables an off-the-shelf Q-SAST tool to operate on SPLs [Ios+17; von+16], it can severely change the structure of the analyzed source code [Pat+22]. In addition, as a variability-oblivious tool is used for the analysis of the transformed code, reported warnings do not retain any variability information [Pat23]. The *warning mapping* component addresses these challenges. It maps the location of a warning reported by the Q-SAST tool (i.e., a *raw* warning) back to its corresponding position in the unpreprocessed source code. Furthermore, it associates a presence condition indicating the configurations under which the warning manifests. The resulting *mapped* warnings are then reported to the user. In the following, we describe the two components of variability encoding and result mapping in more detail. We omit a detailed description of the utilized Q-SAST tool, considering that a standard Q-SAST tool (cf. Section 3.1) can be used without modifications to its internals. However, since variability encoding can lead to severe changes in the analyzed code (and hence the source code representation), it is important to point out that the queries of the Q-SAST tool might need to be adjusted to account for these changes (cf. Section 3.2.2).

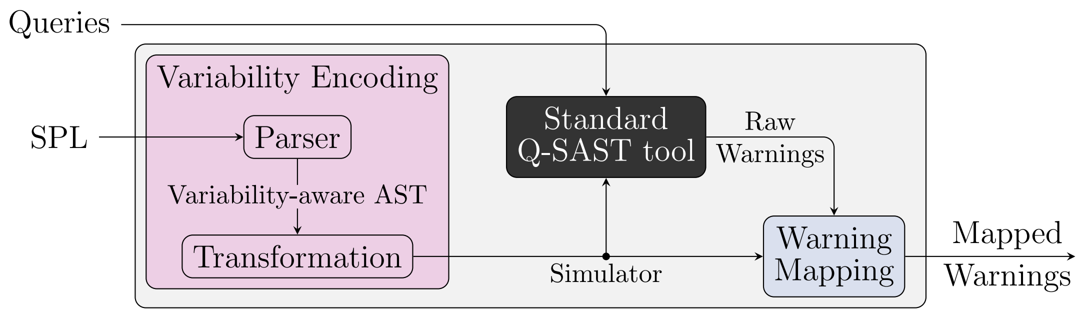


Figure 3.6: Structure of the lifting by simulation strategy

3.3.1 Variability Encoding

Before a standard Q-SAST tool can operate on variable code, the code must be transformed into plain host code. Formally, this process is referred to as *variability encoding* [Ape+11; von+16]. It is based on the concept of configuration lifting initially proposed by Post and Sinz [PS08] and aims at enabling an efficient and comprehensive analysis of SPLs [Ape+11]. To achieve this, variability encoding tries to avoid having to consider every derivable product individually by creating a so-called *product simulator* [Ape+11]. A product simulator (also known as variant simulator [von+16], metaproduct [Thü+12], or metaprogram [PS08]) is a single program incorporating the behavior of all products derivable from an SPL [Ape+13b; Ios+17; Thü+12; von+16]. Considering that the simulator contains only source code of the host language (i.e., C), it can be analyzed using off-the-shelf Q-SAST tools, which do not support unpreprocessed source code out of the box [Ios+17; Pat+22; von+16]. In general, transforming a variable program into a corresponding product simulator represents a challenging task. Below, we therefore describe the fundamental process, its typical implementation, and some of the associated challenges in more detail.

3.3.1.1 General Concept

From a technical perspective, for the construction of a product simulator, variability encoding transforms compile-time variability into run-time variability [Ios+17; Pat+22]. This means that preprocessor conditionals, which are usually evaluated during compile-time, are transformed into variability constructs of the host language (e.g., if statements). These constructs are, in turn, evaluated during the run-time of the program. While the concrete transformations used for realizing variability encoding differ depending on the host language and implementation strategy of the SPL considered, the underlying concept remains the same. Each feature is represented as a global boolean *feature variable* that models the presence or absence of the associated feature [Ape+11].[4] To ensure that all feature configurations remain feasible and are considered during analysis, feature variables are not assigned a fixed value [Ape+11]. Instead, they are either declared as externally defined or initialized non-deterministically [Ape+11; Ios+17]. This ensures that an analysis tool applied to the product simulator cannot assume that certain execution paths are never taken by applying optimizations, such as constant propagation [Aho+07; Ape+11]. Source code that is included under certain feature configurations (i.e., code that is not part of the common core of the SPL) is then encapsulated in a conditional block guarded by a boolean formula over the corresponding feature variables [Ape+11].

```
 1  void foo() {
 2      int x = source();
 3      if(x < MAX){
 4          int y = 0;
 5          #ifdef CONFIG_PROCESS_INPUT
 6          y = 2 * x;
 7          #endif
 8          sink(y);
 9      }
10  }
```

(a) Before variability encoding

```
 1  extern bool PROCESS_INPUT;
 2  void foo() {
 3      int x = source();
 4      if(x < MAX){
 5          int y = 0;
 6          if(PROCESS_INPUT){
 7            y = 2 * x;
 8          }
 9          sink(y);
10      }
11  }
```

(b) After variability encoding

Listing 3.2: The `foo` function (cf. Listing 1.1) before and after variability encoding

The general concept outlined above is exemplified by the `foo` function in Listing 3.2. Line 6 of the unpreprocessed function (Listing 3.2a) is only incorporated into a product if the `PROCESS_INPUT` feature is selected. Accordingly, in the corresponding product simulator (Listing 3.2b) the line needs to be enclosed in an if statement (lines 6-8). The guard of this if statement is a newly introduced `PROCESS_INPUT` feature variable (line 1) that corresponds to the selection of the identically named feature. Based on the value assigned to this variable, the product simulator is able to simulate the behavior of both products derivable from the variable `foo` function at run-time.

[4]For dealing with numerical features, features could be represented as numerical variables or transformed into sets of logical features representing the selection of specific feature values.

3.3.1.2 Variability Encoding Strategies

Based on the general concept of variability encoding described in Section 3.3.1.1, two strategies for constructing a product simulator can be differentiated. These strategies are outlined below.

Naive Variability Encoding

In general, the source code of an SPL can always be transformed into a corresponding product simulator [von+16]. In this regard, similar to parsing variable code into a variability-aware source code representation (cf. Section 3.2.1.2), the trivial approach is to derive every possible product from the SPL. An additional wrapper can then dispatch between the different products at program start depending on the values assigned to the feature variables via run-time parameters [von+16]. For instance, in the context of the `foo` function (cf. Listing 3.2a), an additional function could dispatch between both products derivable from `foo` based on the value of a `PROCESS_INPUT` feature variable. Since the number of products derivable from an SPL may be exponential in the number of its features [DBW19], the naive strategy results in product simulators of impractical size. Within a simulator, it leads to redundancies for every fragment of source code shared between products. Such redundancies not only lead to an inefficient analysis but also defeat the purpose of a family-based strategy, aimed at exploiting the similarities between products [von+16].

Compact Variability Encoding

A more compact and thus practical product simulator can be constructed by transforming each point of variability within the unpreprocessed code in isolation. This allows individual variability patterns to be handled differently. In fact, it is the strategy that was applied for the introductory example shown in Listing 3.2. Since the preprocessor conditional used within the `foo` function is found at statement level (i.e., within a function body) and specified in disciplined form, it could be substituted with a conventional if statement. The guard (i.e., condition) of this if statement was a newly introduced feature variable corresponding to the `CONFIG_PROCESS_INPUT` macro found in the original code. Compared to the naive strategy outlined above, the local transformation of the preprocessor conditional avoids having to include both variants derivable from the variable `foo` function in the product simulator. While certain preprocessor conditionals, such as the one found in the `foo` function, can be handled easily, others demand a more sophisticated transformation. As a last resort, it is always possible to deal with problematic preprocessor conditionals by applying the naive strategy at a finer granularity. However, for certain variability patterns, there are also more advanced solutions [Ios+17; von16] that aim at reducing the resulting code duplication while preserving the behavior of all products in the simulator.

3.3.1.3 Variability Encoding in Practice

So far, we described the concept behind variability encoding and two strategies how a product simulator can be constructed. In general, variability encoding is not a novel concept and has seen considerable research [Gar17; Ios+17; Pat+22; PS08; von+16; von16]. In this section, we thus focus on how variability encoding is commonly realized in practice. In this regard, we first outline how a compact product simulator can be created from variable source code. In addition, we briefly discuss how variability bugs found during the transformation can be incorporated in the product simulator, further preserving the overall behavior of the SPL.

Simulator Construction

From a technical perspective, transforming the code of an SPL into a corresponding product simulator typically involves two steps (cf. Figure 3.6) [Gar17; Ios+17; von+16]. First, the source code of the SPL is parsed into a variability-aware AST. As we have discussed in the context of lifting by extension, in itself, this represents a non-trivial task whose efficient implementation requires sophisticated concepts like a custom grammar or a fork-merge parser (cf. Section 3.2.1.2). In addition, in the case of preprocessor-annotated C code, a configuration-preserving preprocessor [GG12] has to be applied to resolve macro substitutions and file inclusions. Without this step, syntactical structures might be incomplete or identifiers might not be resolved (cf. Section 3.2.1.3).

As a second step, transformations tailored to specific variability patterns can be applied to variable subtrees of the constructed AST to create the product simulator. In this regard, it is possible to transform the source code represented by a subtree into a corresponding part of the product simulator directly [Gar17; von16]. Alternatively, a variability-oblivious AST can be created using tree transformations as an intermediary step [Ios+17]. As a third alternative, the two steps of parsing and transformation can be interwoven by executing special semantic actions[5] during parsing [Pat+22]. This has the benefit that the transformation can take place during parsing, avoiding the computational effort of building a full AST beforehand.

Preserving Variability Bugs

Not every product of an SPL is guaranteed to be type-safe or even syntactically sound [Bra+12; Pat+22]. Therefore, to fully preserve the behavior of the SPL, variability bugs that can be identified during parsing of the variable code, such as type and syntactic errors, need to be incorporated into the product simulator. Even though these kinds of errors can typically be easily detected by the front ends of modern compilers, recall that the potentially exponential number of products derivable from an SPL often prohibits an exhaustive analysis [Aba+17; Pat23]. Thus, in practice, compiling every product individually is not feasible. As a result, it is not uncommon that some products of an SPL exhibit obvious errors [Aba+17], which can even prevent them from compiling [Bra+12; Pat+22]. One way of incorporating these errors into a product simulator is to transform compile-time errors into run-time errors [Pat+22]. To achieve this, upon encountering an error during parsing, an additional statement can be introduced into the corresponding location within the simulator. The statement then indicates the error at run-time by throwing an exception, ultimately aborting the execution of the simulator.

[5]In the context of a syntax-directed translation scheme, semantic actions refer to program fragments that are associated with the productions of a grammar and executed upon a successful match [Aho+07].

3.3.1.4 Prominent Challenges

As mentioned in Section 3.3.1.2, not every variability pattern can be translated into simulator code by replacing the associated preprocessor annotation with an if statement. In this section, we give a brief overview of two general patterns that an implementation aiming for the construction of a compact product simulator needs to address. For general-purpose programming languages like C, there are, however, many other variability patterns whose behavior-preserving transformation into a compact product simulator poses a challenge. As empirical evidence shows, these patterns can generally be encoded using a form of code duplication and renaming [Ios+17; von+16]. For the sake of brevity, we omit a detailed discussion of how this can be achieved and refer the interested reader to the publications of von Rhein et al. [von+16], Iosif-Lazar et al. [Ios+17], and Patterson et al. [Pat+22].

Undisciplined Preprocessor Conditionals

Undisciplined conditionals do not surround entire host language constructs but only an arbitrary sequence of tokens (cf. Section 2.2). Translating them into corresponding host language variability, which naturally can only wrap entire constructs, is therefore not possible. They also severely complicate the parsing of variable source code, which typically forms the first step of a variability encoding implementation (cf. Section 3.3.1.3). A straightforward solution to this problem is to transform the conditionals into disciplined form before trying to translate them into host language constructs. With the exception of ill-formed conditionals, this can automatically be achieved for all undisciplined conditionals [GJ03; Ken+10; LKA11] (cf. Section 2.2). However, in the worst-case scenario, it may result in an exponential number of code clones [Ken+10; LKA11].

Preprocessor Conditionals on Unsupported Scopes

A serious challenge for transforming unpreprocessed C into a corresponding product simulator is the presence of preprocessor conditionals situated at levels other than statement level. Conditionals of this kind are not uncommon and represent a typical way to realize portable code [von+16]. Programming languages like C do, however, not support variability at levels other than statement level (i.e., inside function bodies) [Pat+22; von16]. To address this limitation, it may be necessary to introduce a certain degree of code duplication during variability encoding [Ios+17; Pat+22; von+16]. This is illustrated in Listing 3.3.

Listing 3.3a shows an example of portable code that uses variability to choose the correct data type in a struct based on the target system architecture. Since the variability is found at the level of field declarations, simply substituting the preprocessor conditionals for if statements is not possible. While it would be possible to follow the naive strategy (cf. Section 3.3.1.2), this would lead to a near identical duplicate for both the `C` struct and the `baz` function. Listing 3.3b shows a possible improvement that only duplicates the struct while applying identifier renaming and host code variability to the `baz` function.[6]

[6]A detailed discussion of the improvement and the reason why it does not suffice to only duplicate the variable field inside the struct can be found in Appendix A.1.

```c
// #include ...
struct C {
  char* x;
  #ifdef CONFIG_64BIT
  int64_t y;
  #else
  int32_t y;
  #endif
} C;

double baz() {
  struct C c = {"A", 1};
  int sY = sizeof(c.y);
  int sC = sizeof(c);
  return (double) sY / sC;
}
```

(a) Before variability encoding

```c
// #include ...
extern bool CONFIG_64BIT;

struct C_64BIT {
  char* x;
  int64_t y;
} C_64BIT;

struct C_No_64BIT {
  char* x;
  int32_t y;
} C_No_64BIT;

double baz() {
  struct C_No_64BIT c1 = {"A", 1};
  struct C_64BIT c2 = {"A", 1};

  int sY = CONFIG_64BIT ?
    sizeof(c2.y) : sizeof(c1.y);
  int sC = CONFIG_64BIT ?
    sizeof(c2) : sizeof(c1);
  return (double) sY / sC;
}
```

(b) After variability encoding

Listing 3.3: Portable C code inspired by the example of von Rhein et al. [von+16], before and after variability encoding

In general, applying local code duplication and renaming to transform variability (cf. Listing 3.3b) can, in the worst case, degenerate to the naive strategy. As a result, it may cause the transformed program to exhibit exponential growth in the number of features involved in a particular variability pattern [Ios+17; von+16]. However, as empirical evidence shows [Ios+17; von+16; von16], the resulting blowup can mostly be kept local. The majority of code shared between products is therefore incorporated into the product simulator only once.

3.3.2 Warning Mapping

As a product simulator comprises only host code, it effectively enables off-the-shelf Q-SAST tools to operate on an SPL [Ios+17; Pat+22; von+16]. However, two problems are typically associated with the vulnerability warnings raised on a product simulator [Pat23]:

(1) Warnings relate to locations within the product simulator rather than the unpreprocessed SPL source code.
(2) Warnings are not assigned a presence condition indicating under which configurations of the SPL they manifest.

To provide meaningful information about potential vulnerabilities in an SPLs, these problems need to be addressed. This is the task of the *warning mapping* component. Accordingly, in Sections 3.3.2.1 and 3.3.2.2 we provide more context on the two problems and explore possible options how they can be addressed.

3.3.2.1 Mapping Warning's Locations

Transforming an SPL into a corresponding product simulator is a challenging task that involves careful consideration of many possible variability patterns (cf. Section 3.3.1). Furthermore, to ensure behavior preservation, extensive code duplication and renaming may be required [von+16]. Listing 3.3 illustrated the effect this can have on the structure of source code. Not only can source code increase in length, but the locations of the structures contained within can change. For large source files containing thousands of lines of code, this can mean that the layout of the file changes drastically. Accordingly, when applying a Q-SAST tool to a product simulator, reported warnings may relate to completely different lines of code, compared to the location in the original SPL source code from which they originated [Pat23]. This is not desirable since changes addressing the reported warnings are made to the source code of the SPL and not to individual products or the product simulator. Consequently, warnings need to be mapped back to the locations in the SPL that caused them. For smaller SPLs, whose source files are of limited size, it might be feasible to do this manually. However, for larger systems, a Q-SAST tool may report numerous warnings. Mapping these warnings by hand would represent a very labor-intensive task, taking up valuable time better spent investigating and fixing the underlying causes. Accordingly, to ensure viability of lifting by simulation in practice, it is crucial that an automated process mapping warnings reported on variability-encoded source code to the corresponding location within the SPL is provided.

Employing Inverse Transformations

In theory, it would be possible to automatically determine the location of a warning in the unpreprocessed SPL source code by inverting the transformations applied during variability encoding. The lines of code to which a particular warning relates could then be tracked through the inverse transformations and their final location reported to the user. Evidently, this represents a complicated solution. Not only can it be very labor-intensive to specify inverse transformations for every transformation used for variability encoding, but operations of the partial preprocessor applied during parsing (cf. Section 3.2.1.3) have to be inverted as well. In addition, depending on its implementation, variability encoding may produce similar output for different inputs (e.g., because certain variability patterns are handled identically). Guaranteeing that an inverse transformation recreates the exact structure of the unpreprocessed source code can therefore be challenging. However, if the resulting mapping is not precise enough, interpreting warnings can be even harder than before. Lastly, since variability encoding can be computationally demanding [Pat+22; Sch+22], executing the same transformations in reverse would cause a noticeable performance overhead.

Expanding the Role of Variability Encoding

To avoid the drawbacks of employing inverse transformations, an alternative is to follow the solution proposed by Patterson et al. [Pat+22; Pat23]. This solution involves moving the responsibility of establishing a mapping between the unpreprocessed code and the code of the product simulator to the transformation that takes place during variability encoding. For this purpose, the variability encoding tooling keeps track of where unpreprocessed lines of code end up during the transformation.

Special comments are added after each line of code in the simulator, indicating the corresponding line numbers in the unpreprocessed code [Pat+22]. The task of mapping the locations of warnings then becomes trivial. For every warning, the involved lines of code in the simulator are scanned for special comments inserted during variability encoding. The original line numbers can then be read from the comments and associated with a particular warning.

3.3.2.2 Associating Presence Conditions

Another issue that arises from applying a Q-SAST tool to a product simulator is that warnings are not associated with presence conditions (i.e., formulas specifying the configuration options required for their manifestation). This results from the fact that lifting by simulation employs a variability-oblivious tool for the analysis of what is effectively variable source code. To the tool, there is no difference between conventional if statements and the ones introduced during variability encoding to codify the (un)selection of one or more features. Consequently, the alarms raised retain no variability information [Pat23] and do not indicate the configurations of the SPL to which they apply. For developers, knowing which configuration options (i.e., features) are involved in a warning can, however, be a valuable aid in the interpretation and classification of a warning. Let us consider a scenario where a warning indicating a severe vulnerability requires the selection of a developer feature dedicated to enabling easier debugging. Such a vulnerability probably poses only a minor threat, if any at all. It is created intentionally to aid developers during development and will never manifest in products shipped to customers.[7]

The simplest solution to this problem is to require users to identify presence conditions to the reported warnings by hand, examining the surrounding code regions in either the simulator or the SPL source code. As with establishing a mapping of warnings' locations, this can be very labor-intensive. Alternatively, the internals of the employed Q-SAST tool can be altered in a way such that it emits presence conditions for the warnings raised. However, this contradicts the idea of using off-the-shelf variability-oblivious tools for the analysis, which is central to lifting by simulation. Not only would the overall maintenance effort be increased but changing a particular Q-SAST tool for another would require additional effort. Lastly, the idea of moving large parts of the task to the variability encoding component as proposed for mapping warnings' locations can be applied. To this end, guards of if statements introduced by variability encoding can be specified in a fixed format (e.g., as calls to artificial functions returning boolean values [Pat23]) such that they can easily be differentiated from conventional if statements. As an additional improvement, guards of outer configuration-related if statements can be integrated into the guards of inner configuration-related if statements through conjunction [Pat23]. This avoids having to ascend through all enclosing scopes to identify a complete presence condition. Instead, the guard of the first if statement encountered when ascending though the conditionals introduced by variability encoding already reflects the complete presence condition.

[7]Evidently, this hinges on the assumption that strict checks of the selected features are performed before shipping code out to customers. An assumption that, as illustrated by Apple's FileVault vulnerability in 2012 [Sch+22], does not always hold in practice.

3.4 Discussion

The strategies of lifting by extension and lifting by simulation both exhibit certain strengths and weaknesses. Since these aspects often mirror each other, deciding between the two strategies is not simple. The individual strengths and weaknesses need to be carefully judged and prioritized with regard to the context in which the strategy should be applied. In the following, we consider the criteria of precision, performance, maintenance, extensibility, and required implementation effort. Table 3.1 shows a high-level comparison of the two strategies with regard to these criteria. This comparison is discussed in more detail in Sections 3.4.1 to 3.4.5. Lastly, with focus on the scope of this thesis, Section 3.4.6 elaborates on the rationale behind our choice, which is in favor of lifting by simulation.

Criterion	Lifting by Extension	Lifting by Simulation
Precision	Avoids imprecision due to analysis of altered source code	Analysis of altered source code can introduce imprecision [Pat23]
Performance	Immediate analysis of variable code	Additional pre- and post-processing
Maintainability	High maintenance effort [GW19]	Moderate maintenance effort
Extensibility	Heavily dependent on the tool to be lifted	Nearly independent of the tool to be lifted
Implementation Effort	Idea applied only conceptually	Applied in the context of conventional SAST tools

Table 3.1: A high-level comparison of lifting by extension and lifting by simulation, focusing on the criteria of precision, performance, maintainability, extensibility, and implementation effort

3.4.1 Precision

While any static analysis approach dedicated to the identification of complex vulnerabilities is limited to providing approximate results,[8] overall precision can vary between different analysis approaches. The ultimate goal is to be as precise as possible. Since an increase in precision is typically traded with a decrease in performance, a more realistic goal is to be as precise as possible while maintaining adequate performance. In the context of this thesis, we aim to leverage the capabilities and performance of Q-SAST for the analysis of variable code by lifting an off-the-shelf Q-SAST tool to the domain of SPLs. It is not the goal to reengineer the fundamentals of the approach employed by the analysis tool to improve its overall precision. As a result, imprecision introduced by the analysis tool and its analysis approach are inevitable for both strategies. However, apart from this imprecision, there can be a difference between employing lifting by extension and lifting by simulation.

Lifting by Extension

Using lifting by extension, the Q-SAST tool operates directly on a variability-aware source code representation. Such a representation allows the source code of an

[8]Recall that this a direct consequence of Rice's theorem [Ric53].

entire SPL to be captured in a compact structure, relying only on lightweight constructs such as choice nodes or presence condition labels for encoding variability (cf. Section 3.2.1.1). Therefore, assuming that insufficient variability-awareness of queries is not a problem (cf. Section 3.2.2), the matching process performed by the tool's search engine should remain as precise as for non-variable code. Nonetheless, a variability-aware source code representation is typically larger in size than its variability-oblivious counterpart. Limitations of internal components of the analysis tool can therefore surface, negatively affecting precision. As the goal of lifting by extension is to extend the capabilities of the tool by altering its internals, these limitations could be addressed by adapting the corresponding component to suit the new conditions. Lastly, the use of variability-oblivious queries for modeling patterns that can be affected by variability can lead to missed matches (cf. Section 3.2.2). As demonstrated in Section 3.2.3, simple instances of this problem can, however, be addressed by adjusting the search engine.

Lifting by Simulation

The idea of lifting by simulation is to use an off-the-shelf Q-SAST tool for the analysis of a product simulator. While this allows the analysis tool to be reused without requiring alterations to its internals, analysis of a product simulator can introduce a certain amount of imprecision [Ios+17]. To emulate the behavior of the whole SPL, the product simulator encompasses additional host language conditionals, effectively introducing new execution paths into the code to be analyzed (cf. Section 3.3). Since the newly added paths encapsulate feature-specific code not contained in the SPL's common core, their guards (i.e., presence conditions) can be complex, involving numerous feature variables. Therefore, for analysis tools it can be challenging to reliably identify mutually exclusive paths, leading to false vulnerability warnings once accidentally ignored [Pat23]. Moreover, to preserve the behavior of the individual products of an SPL, the product simulator often has to rely on code duplication and renaming of identifiers (cf. Section 3.3). In practice, this can cause the source code to expand to a multiple of its original, unpreprocessed size [Ios+17; Pat+22], a fact reflected in the source code representation of the utilized analysis tool. Limitations of internal components can therefore quickly surface. Addressing these limitations would require alterations to the analysis tool's internals. However, not only would this contradict the idea of lifting by simulation, but it would also eliminate one of its main benefits in being nearly independent of the analysis tool used. A similar limitation can be observed for the use of queries that do not account for structural changes in the product simulator. While simple instances of this problem could be addressed by adjusting the search engine (cf. Section 3.2.3), for lifting by simulation this is not an option. Consequently, an additional translation component responsible for translating queries would need to be introduced.

Summary

While both lifting strategies inherit the imprecision of the chosen Q-SAST tool and its analysis approach, lifting by extension promises higher precision. It avoids imprecision resulting from analyzing variability-encoded source code, which tends to incorporate many redundancies, and allows limitations relating to internal components and queries to be addressed directly.

3.4.2 Performance

Large real-world SPLs tend to receive many commits per day and thus continuously evolve [Pet+23]. As a result, receiving warnings about potential vulnerabilities days or even weeks after the analysis was started is anything but ideal. By then, the SPL's code might have significantly changed, making the warnings of an analysis tool hard to interpret or even obsolete. Performance is therefore crucial to the viability of an analysis approach in practice. Considering that typically not all parts of an SPL's code base can be integrated into a single product (e.g., due to mutual exclusive features or feature model constraints) [ABW14; Gar17], analyzing an SPL using a family-based strategy (i.e., as a whole) is generally more resource demanding than analyzing any single product by itself. As with precision, the degree to which additional overhead is introduced can vary depending on the lifting strategy.

Lifting by Extension

With lifting by extension, variable source code is parsed into a variability-aware source code representation that is immediately analyzed for matches to a user's queries. As the source code representation is constructed directly from an SPL's variable source code, it contains only limited redundancies that could make finding matches to queries more computationally demanding (e.g., those required for transforming undisciplined annotations into a disciplined form). Additionally, the analysis does not require separate pre- or post-processing steps, as all associated tasks can be carried out within the tool during the analysis through extensions of the corresponding components. For instance, presence conditions can be assigned to vulnerability warnings by considering the choice nodes or presence condition labels involved in the matched parts of the source code representation. In this regard, it would even be possible to construct presence conditions during the matching process by tracking the traversed choice nodes or encountered presence condition labels. Similarly, a mapping between elements in the variability-aware source code representation and regions in the unpreprocessed source code can already be established during parsing.

Lifting by Simulation

Lifting by simulations aims at the analysis of a previously created product simulator with conventional Q-SAST tooling. Similar to lifting by extension, this process involves parsing variable code into a variability-aware representation (often an AST [Gar17; Ios+17; von+16]). However, this represents only one part of variability encoding. To generate the product simulator, extensive transformations typically need to be executed on the representation (cf. Section 3.3.1.3). The analysis tool then takes the simulator as input and parses it into its own source code representation (e.g., CPGs in the context of JOERN). Parsing is thus effectively performed twice. In addition, a product simulator may contain many redundancies with the aim of preserving the behavior of all products of the SPL (cf. Section 3.3.1). Since redundancies are reflected in the source code representation of the analysis tool, identifying matches to a user's queries can become even more computationally demanding [Thü+12]. Lastly, with lifting by simulation, the necessary tasks of mapping warnings to their original location and assigning them presence conditions can only be performed after the analysis tool finished its execution. Otherwise, alterations to the tool would be required, which would contradict the idea of lifting by simulation.

Summary

In essence, lifting by extension promises greater performance than lifting by simulation. Not only is parsing performed only once, but the analysis takes place on a compact variability-aware source code representation that does not contain unnecessary redundancies. Furthermore, computationally demanding pre- and post-processing steps are not required.

3.4.3 Maintainability

As emphasized by Lehman's first law of software evolution [Leh80], to remain relevant in practice, a software system that is used and that reflects some real-world activity needs to be continuously improved and maintained. This insight also applies to static analysis approaches, given that they are frequently used to automate the analysis of ever-evolving software systems during their associated development and evolution activities. Considering maintainability in the context of lifting by extension and lifting by simulation, two aspects can be compared. First, popular Q-SAST tools, such as JOERN or CODEQL, are themselves continuously improved and maintained. It is therefore of interest how well updates to the chosen analysis tool can be incorporated into the analysis approach using the two lifting strategies. Second, on a broader scale, it is relevant how easily the overall approach resulting from an implementation of the strategies can be maintained.

Lifting by Extension

Since lifting by extension aims to alter the internals of the chosen Q-SAST tool, updates to its original (i.e., unaltered) version cannot always be simply integrated. If an update relates only to unchanged sections of the tool's source code and does not alter newly introduced dependencies, the updated code can simply be merged. However, if an update relates to sections of the tool's code for which alterations were necessary, resulting conflicts have to be manually resolved. This can not only be labor-intensive but also error-prone. Furthermore, with lifting by extension, all aspects of the analysis are performed within the lifted analysis tool. While Q-SAST tools are typically structured into distinct components (cf. Section 3.1), their internal complexity can nonetheless be high. Performing maintenance activities can thus be challenging.

Lifting by Simulation

Using lifting by simulation, the chosen Q-SAST tool is treated as a black box that does not necessitate any changes to its internals. Incorporating updates to the analysis tool is therefore simple and limited to replacing the old version with the new one and adjusting corresponding calls in case the call semantics of the tool have changed. In addition, considering the overall maintainability of the resulting analysis approach, lifting by simulation establishes a clear separation into the three distinct components of variability encoding, Q-SAST tool, and warning mapping (cf. Figure 3.6). Since these components are executed in sequence and act as individual processing steps, the overall structure bears resemblance to the classic pipes and filters architectural pattern [Bus+96]. This not only allows individual components to be exchanged and reused [Bus+96] but also eases maintenance given that each component represents a self-contained step in the analysis process.

Summary

Lifting by simulation offers distinct benefits over lifting by extension in terms of maintainability. Through its pipes and filters architecture, maintainability tasks can be performed more easily than on lifting by extension's tightly integrated structure. Furthermore, since the chosen Q-SAST tool is treated as a black box, updates can be seamlessly incorporated without the need to manually resolve conflicts between versions.

3.4.4 Extensibility

Another factor worth considering is that there are multiple Q-SAST tools that enjoy widespread adoption in industry [Li+24]. As these tools typically possess unique strengths, there is no single Q-SAST tool that suits all situations. It is therefore of interest how seamlessly an implementation of lifting by extension and lifting by simulation can integrate with other Q-SAST tools.

Lifting by Extension

While the alterations made to a Q-SAST tool's internals trough lifting by extension enable for variability to be handled as it is most suitable in that context, they are likely to be highly specialized to the internals of the chosen tool. Since other tools may be implemented in a different programming language or realized differently, reusing the alterations necessary for lifting one Q-SAST tool in the context of another can be difficult. Consequently, expanding to other tools with lifting by extension is generally labor-intensive and entails applying the strategy anew. In addition, as the strategy involves alterations to a tool's internals, it may only be applied to Q-SAST tools that are either open-source or provide access to sufficient parts of their implementation.

Lifting by Simulation

As lifting by simulation does not necessitate any changes to the chosen Q-SAST tool, another tool can simply be substituted. In this regard, most parts of the existing variability encoding and warning mapping components can be reused. Notably, only two minor adjustments might be necessary. First, if the call semantics of the new tool differ from those of the previously used tool, the parts issuing calls to the Q-SAST tool need to be adjusted. Additionally, the warning mapping component may need to be adjusted in case the new tool uses a different format to encode warnings (e.g., JavaScript Object Notation (JSON) vs. Extensible Markup Language (XML)). This near-independence from the chosen analysis tool provides the added benefit that even closed-source Q-SAST tools can be lifted using lifting by simulation.

Conclusion

Considering their expansion capabilities, lifting by simulation clearly outperforms lifting by extension. Through its near independence of the chosen Q-SAST tool, lifting by simulation offers simple expansion to other tools and enables even closed-source solutions to be lifted. Lifting by extension, on the other hand, applies extensive alterations to the chosen Q-SAST tool that make an expansion to other analysis tools labor-intensive and require extensive access to a tool's internals.

3.4.5 Implementation Effort

Lifting an off-the-shelf Q-SAST tool to the domain of SPLs represents a complex task that demands a significant amount of effort. Evidently, this effort plays a key role in the selection of a lifting strategy. In this regard, the effort needed for implementing lifting by extension and lifting by simulation differs.

Lifting by Extension

In essence, for its implementation, lifting by extension requires inspecting the internals of the chosen Q-SAST tool and making adjustments for accommodating variability. These adjustments mostly focus on the utilized source code representation and its associated parsing infrastructure (cf. Section 3.2). While altering a complex component of an analysis tool, such as the employed parser, can be a difficult task, the tool does provide a solid base to expand upon. For instance, altering the operation of the existing parser used in a Q-SAST tool is generally less labor-intensive than engineering a new parser from the ground up.

While there are solutions of prior research relating to the general idea of lifting by extension, these solutions follow a more conceptional approach and aim at lifting classic data-flow analyses specified via a CFG, lattice and transfer functions [Bra+12; Bra+13] or the IFDS framework [Bod+13]. For lifting an off-the-shelf tool, they are therefore not suitable and cannot be reused [Pat23]. In addition, while there are established solutions dedicated to creating variability-aware data structures [Wal+14] and ASTs [GG12; Ken+10], work on creating variability-aware variants of advanced source code representations, such as CFGs, PDGs, or CPGs, has been limited. Consequently, for extending a tool's parsing infrastructure, it might be necessary to create either a new variability-aware parser for the desired representation or establish a transformation starting from a variability-aware AST.

Lifting by Simulation

Contrary to lifting by extension, lifting by simulation does not provide a foundation to build upon and requires separate implementations for the variability encoding and warning mapping components. Especially the implementation of the variability encoding component can represent a serious undertaking (cf. Section 3.3.1). Not only does it typically require a variability-aware parser but also a set of behavior-preserving transformations that can handle most variability patterns encountered in real-world C.

From an objective point of view, lifting by simulation appears very labor demanding to implement. This is misleading, considering that there are practical solutions from previous work that can be reused for this purpose. For creating variability-aware ASTs from unpreprocessed C source code, there are tools such as TypeChef [Ken+10] or SuperC [GG12]. Built on these tools there are solutions like Hercules [Gar17; von+16], C Reconfigurator [Ios+17], and SugarC [Pat+22] that realize variability encoding. With Sugarlyzer [Pat23], there even exists a framework that augments SugarC's variability encoding with a warning mapping component and allows generic SAST tools to be utilized for an analysis via corresponding interfaces. For the implementation of the variability encoding and warning mapping components, it is therefore possible to draw from a number of existing solutions. This, in return, avoids having to implement both components from scratch, greatly reducing the overall effort required for implementing lifting by simulation.

Summary

When comparing the implementation of the two strategies, lifting by simulation with its requirements for dedicated variability encoding and warning mapping components appears to involve more effort than the alterations performed by lifting by extension. However, taking into account the results of previous work in this field, for lifting by simulation there are already practical solutions for both additional components. On the other hand, previous work relating to lifting by extension focuses on lifting general data-flow analyses and is not suitable for lifting an off-the-shelf tool. In addition, the work on the creation of variability-aware variants of source code representations popular among analysis tools has been limited. Consequently, when taking the potential reuse of existing solutions into account, the implementation of lifting by simulation generally requires less effort.

3.4.6 Our Choice

For the scope of this thesis, we aim to lift the off-the-shelf Q-SAST tool JOERN to the domain of SPLs. In this regard, we concentrate on SPLs implemented in C via an annotative implementation strategy by use of the C preprocessor CPP (cf. Section 1.2). In pursuit of this goal, our choice fell on the strategy of lifting by simulation. While lifting by extension promises greater precision and performance (cf. Sections 3.4.1 and 3.4.2), its drawbacks in terms of maintainability, extensibility, and implementation effort (cf. Sections 3.4.3 to 3.4.5) are particularly important in the context of this thesis.

Since JOERN is a tool that is still actively maintained and improved, being able to easily incorporate updates is an important factor for ensuring the long-term viability of our approach. Similarly, it is crucial to be able to easily maintain our overall approach in the future, considering that we do not have the same capacities as a commercially developed tool with dedicated developers and maintainers. In view of the lack of existing solutions, our aim is instead to provide a first concept for family-based vulnerability discovery leveraging the benefits of Q-SAST that can be continuously improved and expanded. This is also the main reason why we are prepared to compromise on the general precision and performance of the approach.

Additionally, while JOERN represents an established and powerful Q-SAST tool, other Q-SAST tools, such as CODEQL [24e] or SONARQUBE [24m], are widespread and possess unique advantages [Li+24]. As there subsequently is no single Q-SAST tool that suits every situation, it is important to maintain the option to expand to other tools in the future. Given that many popular Q-SAST tools are not completely open-source [Li+24], being able to expand to even closed-source solutions provides an added benefit.

Lastly, considering the size and complexity of JOERN,[9] implementing lifting by extension represents a daring task that can only be supported to a limited extent by reusing existing solutions (cf. Section 3.4.5). Most existing work focuses on general data-flow analyses and hence cannot be used to lift a concrete tool. Additionally,

[9]As of August 2024, JOERN consists of more than 180,000 lines of Scala code. Additionally, it relies on the open-source in-memory graph database FLATGRAPH, which itself adds Scala code in excess of 15,000 lines.

while Gerling and Schmid [GS19] introduced the concept of variability-aware CPGs to enable semantic slicing on SPLs, crucially, their solution cannot accurately handle undisciplined annotations [Kra19]. Since we set out to support both disciplined and undisciplined annotations (cf. Section 1.2), reusing Gerling and Schmid's solution would thus require non-trivial adjustments. Overall, implementing lifting by extension therefore represents a task whose extent exceeds the scope of this thesis.

4. Design

In the last chapter, we have described two strategies for lifting a variability-oblivious Q-SAST tool to the domain of SPLs: lifting by extension (cf. Section 3.2) and lifting by simulation (cf. Section 3.3). Taking into account the benefits and drawbacks of both strategies, for our goal of lifting the off-the-shelf tool Joern, we decided to pursue lifting by simulation. In this chapter, we now describe the design we chose for the implementation of the strategy.

4.1 Initial Considerations

Recall that the fundamental idea of lifting by simulation is to rewrite variable source code into plain host code before analysis, in a process known as variability encoding. In the context of this thesis, this means that unpreprocessed C (i.e., C code still containing preprocessor annotations) has to be transformed into plain C. This transformation can severely change the structure of the source code being analyzed. Thus, to provide useful insights, the vulnerability warnings reported by the analysis tool need to be mapped back to their original location within the unpreprocessed source code. Additionally, warnings need to be assigned a presence condition (i.e., a propositional formula over feature variables) describing the configurations of the SPL in which they manifest. The structure resulting from this approach was described in detail in Section 3.3. This structure is illustrated again in Figure 4.1, highlighting the components and inputs that can remain unchanged, those that may require alterations, and those that are newly introduced when aiming for the analysis of an SPL with a variability-oblivious Q-SAST tool and a set of queries. In the following sections, we describe our design for variability encoding (Section 4.2) and warning mapping (Section 4.3) in more detail since they represent newly required components. Moreover, we explain the source of the queries used by our approach and detail the extent to which these queries need to be adjusted (Section 4.4). We do not elaborate on the implementation of the Q-SAST tool because lifting by simulation theoretically enables any variability-oblivious tool to operate on an SPL's source code without requiring alterations to its internals. However, for the scope of this thesis, we chose the analysis tool Joern (cf. Section 1.2) as it not only allows us to leverage the benefits of Q-SAST but also provides extensive capabilities for detecting real-world vulnerabilities (cf. Section 2.3). Lastly, we also do not separately consider the SPLs that should be analyzed. In this regard, transforming their preprocessor-based variability theoretically allows any C-based SPL to be analyzed without requiring changes to its code base.

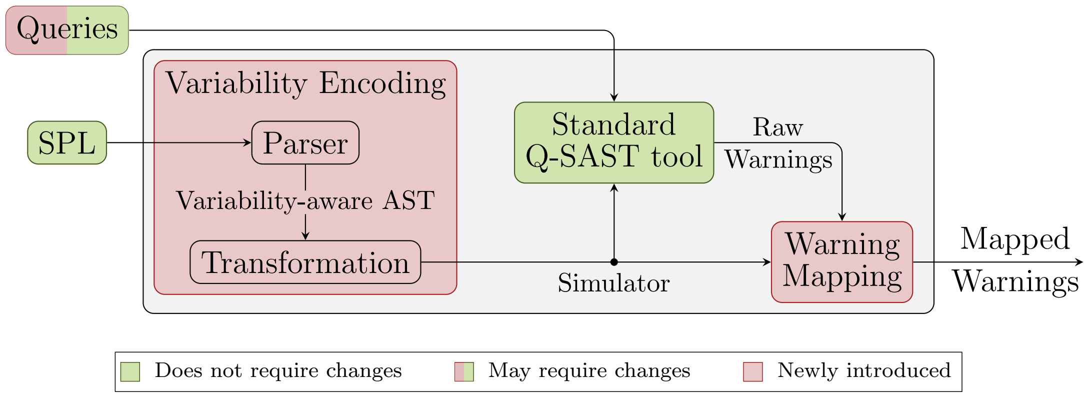


Figure 4.1: Categories of components in the lifting by simulation strategy

4.2 Variability Encoding

As we have described in Section 3.3, variability encoding is typically realized using a variability-aware parser in conjunction with a set of behavior-preserving transformations applied either during or after parsing. Furthermore, during variability encoding, it is possible to incorporate a mapping to presence conditions and unpreprocessed source code regions into corresponding sections of the simulator. Without this information, the warning mapping component would need to rely on applying inverse variability encoding to derive the necessary information (cf. Section 3.3). Overall, implementing variability encoding from scratch can thus be very labor-intensive. For our design, we thus aim to reuse an existing solution. In this regard, to the best of our knowledge, there are three existing solutions that form the state of the art in variability encoding of unpreprocessed C. These solutions are outlined below, together with the reasoning behind our choice.

Hercules

First, there is HERCULES [Gar17; von16]. HERCULES represents an extension to the variability-aware parser TYPECHEF [Ken+10] and aims at using variability encoding for the performance measurement of SPLs [Gar17]. Since this underlying objective differs from ours, certain changes to HERCULES' internals would be necessary. By default, HERCULES does not incorporate easily accessible information on line mappings and presence conditions into the product simulator. To avoid having to implement an inverse variability encoding inside the warning mapping component, this functionality would thus need to be added. Additionally, in accordance with its goal, HERCULES introduces performance measuring functions into the product simulator [Gar17]. These additions to the simulator have the potential to raise false warnings during an analysis and might need to be removed for the purpose of our approach. Yet, with HERCULES and TYPECHEF exhibiting an extended period without any maintenance to their source code (the most recent commits to their repositories[1] were issued in 2017 and 2021, respectively), implementing changes may face additional difficulties. Another drawback of Hercules is that by using TypeChef, it is only able to transform variable source code once it can be successfully parsed into a variability-aware AST without any type errors [Gar17; Pat23]. However, considering past observations in the context of large real-world SPLs [Aba+17], this is not always the case. Thus, using Hercules, it is questionable whether any real-world SPL could be analyzed without ensuring type correctness for every product beforehand. Lastly, the transformations applied are only informally described and do not cover many variability patterns found inside real-world C [Pat23].

C Reconfigurator

Another existing solution is C Reconfigurator [Ios+17]. Contrary to Hercules, C Reconfigurator builds on the variability-aware parser SuperC [GG12; Ios+17]. It was developed with the aim of enabling the identification of variability bugs in SPLs using variability-oblivious analysis tools [Ios+17]. While the underlying goal is thus closely related to ours,[2] the tool carries some of the same drawbacks as Hercules. First, the solution does not emit explicit information indicating line mappings or presence conditions into the product simulator. In addition, since C Reconfigurator is considered a prototype tool, its transformations are specified and proven on the idealized language IMP, which only represents a subset of C [Ios+17]. Moreover, variable source code can only be transformed into a corresponding product simulator if it does not exhibit any syntax errors [Pat+22]. Lastly, as with Hercules, development and maintenance on the tool have stalled. In fact, the most recent commit to C Reconfigurator's repository[3] dates back to 2017.

SugarC

The most recent solution for variability encoding of unpreprocessed C is SugarC [Pat+22]. Similar to C Reconfigurator, SugarC builds on SuperC as the underlying parsing framework and aims at enabling the application of variability-oblivious analysis tools to SPLs [Pat23]. However, contrary to C Reconfigurator, the resulting product simulator incorporates information on both line mappings and presence conditions in a form that can easily be accessed by a warning mapping component. For instance, to establish a mapping between line numbers, specially formatted C comments containing the original line ranges are appended to the simulator's lines of code [Pat+22]. In addition, SugarC is capable of handling syntax and type errors and preserves them in the product simulator as run-time errors [Pat+22]. While the applied transformations are not complete and their correctness is not formally proven, SugarC claims to support many complex cases found inside real-world C [Pat+22]. From a technical perspective, it is integrated into the repository of SuperC,[4] which saw frequent commits up until 2022.

Our Choice

Taking into account the three solutions for variability encoding outlined above, for our design, we chose SugarC. Not only does it represent the latest development dedicated

[1]https://github.com/joliebig/Hercules, https://github.com/ckaestne/TypeChef.
[2]Recall that we concentrate on the identification of VIVs using Q-SAST tools.
[3]https://github.com/itu-square/c-reconfigurator.
[4]https://github.com/appleseedlab/superc.

to variability encoding of unpreprocessed C but it exhibits the shortest period without regular maintenance to its source code. In addition, it is the only solution among the three that directly embeds information on line mappings and presence conditions into the product simulator. However, most crucially, comparing the tools head-to-head in terms of their ability to handle constructs found in unpreprocessed C code, Patterson et al. [Pat+22] demonstrated that SugarC outperforms both Hercules and C Reconfigurator. To this end, all three tools were applied to DesugarBench, a benchmark for variability encoding consisting of 108 small but variable programs spread across 12 categories [Pat+22]. While no tool was able to handle all 108 programs contained in the benchmark, SugarC passed 97 [Pat+22]. In comparison, Hercules passed 73 of the programs and C Reconfigurator, as a prototype tool, passed only 32 [Pat+22].

4.3 Warning Mapping

Since SugarC adds readily accessible information on line mappings and presence conditions to the product simulator, implementing a corresponding warning mapping component becomes straightforward. Accordingly, Patterson [Pat23] already provided an implementation as part of his Sugarlyzer[5] framework. In principle, Sugarlyzer can be seen as a framework facilitating the implementation of the lifting by simulation strategy (cf. Figure 4.1). To this end, it uses SugarC for variability encoding and provides dedicated interfaces for incorporating variability-oblivious SAST tools for the analysis of the resulting product simulator. Warnings reported by the analysis tool are passed to the warning mapping component, which, in turn, leverages the information embedded into the simulator by SugarC. Moreover, the warning mapping component already performs advanced postprocessing tasks. For instance, warnings relating to the same issue found within the unpreprocessed code (e.g., caused by local code duplication as often required for handling undisciplined annotations and ensuring behavior preservation) are deduplicated and their presence conditions joined by disjunction. In addition, the presence conditions tied to the reported warnings are checked for their satisfiability. Warnings whose presence condition constitutes a contradiction (e.g., because the code segments involved would require a simultaneous selection and deselection of a feature) can subsequently be pruned as they can never manifest in real products of the SPL.

Considering that Sugarlyzer already provides a capable warning mapping component tailored to SugarC for variability encoding, we decided to reuse this implementation for our design. In this respect, it would have been possible to reuse just the implementation of the warning mapping component. However, beyond variability encoding and warning mapping, the Sugarlyzer framework also provides fundamental abstractions required for the implementation of lifting by simulation. For instance, Sugarlyzer already provides classes for the warnings issued by a SAST tool or the subject systems to be analyzed. Accordingly, to benefit from the existing abstractions, we based our overall design on Sugarlyzer. This noticeably reduces the effort required for the implementation of our approach. Since Sugarlyzer supports the integration of other tools through corresponding interfaces, it additionally opens the door for an extension of our analysis approach to other Q-SAST tools.

[5]https://github.com/UTD-FAST-Lab/Sugarlyzer.

4.4 Queries

One of the main characteristics of lifting by simulation using a Q-SAST tool is that the analysis can be controlled through the provided set of queries (cf. Figure 4.1). To this end, Section 4.4.1 describes the source of the queries used for our design. Furthermore, as we have described in Section 3.3, existing queries might have to be adjusted to accommodate variability. Therefore, Section 4.4.2 describes the extent to which the used queries have to be adjusted.

4.4.1 Source and Integration

While the customizability enabled by using user-provided queries offers great benefits (cf. Section 2.3), for a transparent evaluation of the capabilities of our analysis approach, using a known set of representative queries is vital. Additionally, while SUGARLYZER theoretically allows any variability-oblivious SAST tool to be used for the analysis, its design is clearly based on classic NQ-SAST tools. By default, it does therefore not allow the user to provide inputs to the analysis tool and expects the execution of a fixed analysis. Taking both aspects into account, for our design, we depart slightly from the structure imposed by lifting by simulation (cf. Figure 4.1) and limit the analysis to a fixed set of queries. To this end, we make use of queries taken from the JOERN query database [24j]. This community-maintained database provides a collection of 27 queries modeling common issues found in C code as traversals on a CPG. More specifically, we focus on the 16 queries with the default tag. This represents the set of queries executed when using JOERN in NQ-SAST fashion, relying on the default query set without the ability to customize the analysis. Limiting the analysis to a fixed set of queries certainly restricts the capabilities of our analysis approach and eliminates many benefits of using a Q-SAST tool. However, it not only makes an evaluation of the approach more transparent but also decreases the overall implementation effort as the structure of SUGARLYZER does not have to be altered to allow user-provided queries to be passed to the analysis tool. Nonetheless, it is worth noting that there are no technical limitations that would prevent us from extending SUGARLYZER with such a functionality in the future.

4.4.2 Need for Adjustments

Variability encoding can severely alter the structure of the analyzed source code and thus the CPG used by JOERN. Consequently, queries that strictly match on particular structures inside a CPG have to be relaxed to provide useful results on a product simulator (cf. Section 3.3). As described above, for our design, we rely on the default query set for C taken from the JOERN query database [24j]. To determine whether these queries would need to be adjusted as part of our implementation, we reviewed the individual queries by hand and identified two categories: (1) queries whose scope is limited to a single statement and its substatement level and (2) queries whose scope exceeds a single statement.

Queries with a Limited Scope

For queries of category (1) (i.e., those with a limited scope), it is important to observe that variability encoding does not alter the internal structure of individual statements. Therefore, the effectiveness of such queries is not affected. In the worst case scenario, where a statement contains undisciplined preprocessor annotations, the statement will merely be duplicated for all its variants. These variants are subsequently incorporated into the CPG of the product simulator and correspond to the variants found in the individual products of the SPL. Since the query will thus be able to find the modeled pattern within the product simulator, it will remain just as effective as on conventional (i.e., non variability-encoded) source code. Adjustment to queries of category (1) are therefore not necessary.

Queries with an Unlimited Scope

Considering queries of category (2) (i.e., those with an unlimited scope), the situation is slightly more complex. Queries whose scope exceeds a single statement are particularly important for the discovery of complex vulnerabilities, as they allow patterns spread across vast regions of a program to be matched. However, with the transformations applied during variability encoding drastically changing the overall code structure, their effectiveness can be decreased. For instance, a query might not expect an extra conditional (e.g., an if statement) surrounding one of the modeled source code regions. The queries of the Joern query database [24j] circumvent this problem by avoiding describing the expected patterns solely by their syntactical structure. Instead, queries leverage the expressiveness of the CPG and use semantic traversals to establish relationships between program constructs. These often-predefined traversals, such as `dominatedBy` or `reachableBy`, follow semantic relationships, such as control flow domination or data flow, codified in the CPG. Since the goal of variability encoding is to preserve the behavior (i.e., the semantics) of the unpreprocessed program, these relationships should not be affected by the structural changes to the program (and hence the CPG). Adjustment to queries of category (2) are therefore not necessary either.

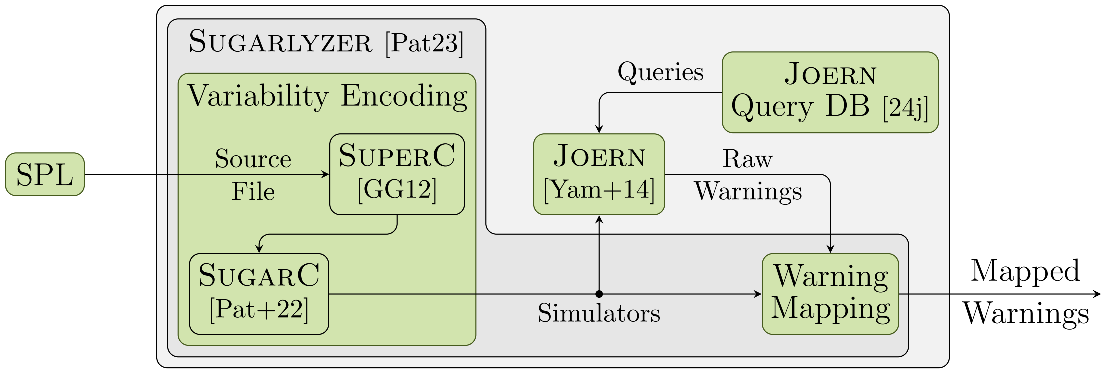
Conclusion

In summary, the changes to the source code and the associated source code representation incurred by variability encoding should have no effect on the effectiveness of the queries considered for our implementation. We can thus reuse the queries of the Joern query database without requiring alterations to their specification. In this regard, we want to emphasize that this is not universally the case. Accordingly, when incorporating newer versions of the query database or using different queries altogether, the queries have to be reevaluated.

4.5 Final Design

All the choices outlined in the previous sections culminate in the overall design of our analysis approach. This design is illustrated in Figure 4.2. Note that the Sugarlyzer framework operates on individual source files rather than the entire SPL at once. Source files are desugared and analyzed in isolation before the reported warnings of all files are collected just before warning mapping. Since the involved tools (i.e., SuperC, SugarC, and Joern) are hence repeatedly restarted, this approach can introduce additional overhead. However, it also allows for more flexibility. Since files are desugared and analyzed individually, several source files can be processed simultaneously, introducing a simple form of parallelism into the analysis approach.

Implementing a rudimentary caching functionality also becomes straightforward, as there are separate intermediary results for every file. Considering that Sugarlyzer already implements these improvements, we decided against changing this internal behavior. In this regard, it is important to highlight that it would also have been possible to alter the internals of Sugarlyzer to enable the analysis of an entire SPL at once. Furthermore, from a technical perspective, SugarC is part of SuperC and realized through special semantic actions executed during the parse process. As a result, the transformation into a product simulator takes place during parsing and is not performed afterward on a fully-built variability-aware AST. Given that there is still a clear distinction between the activities of parsing source code (i.e., matching productions of a grammar) and transforming it into a product simulator (i.e., executing semantic actions), we illustrate SuperC and SugarC as separate components of our design.



🟩 Does not require changes

Figure 4.2: Design of our implementation of lifting by simulation

While implementing lifting by simulation from the ground up is very labor-intensive and certainly out of the scope of this thesis, by reusing existing solutions, we can reduce the implementation effort to a minimum. As shown by Figure 4.2, no components remain to be newly developed. Thus, for implementing our design, the main task focuses on adding support for Joern to the Sugarlyzer framework.

5. Implementation

Among the two lifting strategies outlined in Chapter 3, for the scope of this thesis, we chose to implement the strategy of lifting by simulation. To this end, Chapter 4 detailed our design that resolves around the SUGARLYZER framework of Patterson [Pat23]. In this chapter, we describe the concrete implementation of this design. The aim of the implementation is to integrate our family-based analysis approach into VARI-JOERN,[1] an existing platform, allowing for an optimized product-based analysis of SPLs for vulnerabilities using JOERN. The result should be a platform offering simple access to both the product- and family-based analysis strategy.

5.1 Vari-Joern

VARI-JOERN is an existing analysis platform that employs an optimized product-based analysis strategy for the discovery of vulnerabilities in SPLs. It does therefore not analyze an SPL as a whole but uses different sampling strategies to derive a representative subset of products (cf. Section 2.1). This set of products is analyzed using JOERN. Therefore, while the methods may vary, the fundamental objective of utilizing the off-the-shelf Q-SAST tool JOERN for the analysis of SPLs remains identical to our approach. From a technical perspective, VARI-JOERN is realized as a Java application combining five distinct components. This application ships with its own Dockerfile and is intended to be run in a corresponding DOCKER[2] container. This not only ensures consistency across different execution environments, but also guarantees that VARI-JOERN's external dependencies can be resolved. The resulting structure is illustrated in simplified form in Figure 5.1.

For an analysis, the five components are executed in sequence. First, VARI-JOERN employs a feature model reader to extract the feature model of the provided SPL. This feature model is passed to a sampler that selects a representative subset of products according to a certain sampling strategy. A composer receives the provided SPL, together with the sample chosen by the sampler, and derives the corresponding products. These products are analyzed with JOERN using queries from the JOERN

[1]https://github.com/KIT-TVA/Vari-Joern.
[2]https://www.docker.com/.

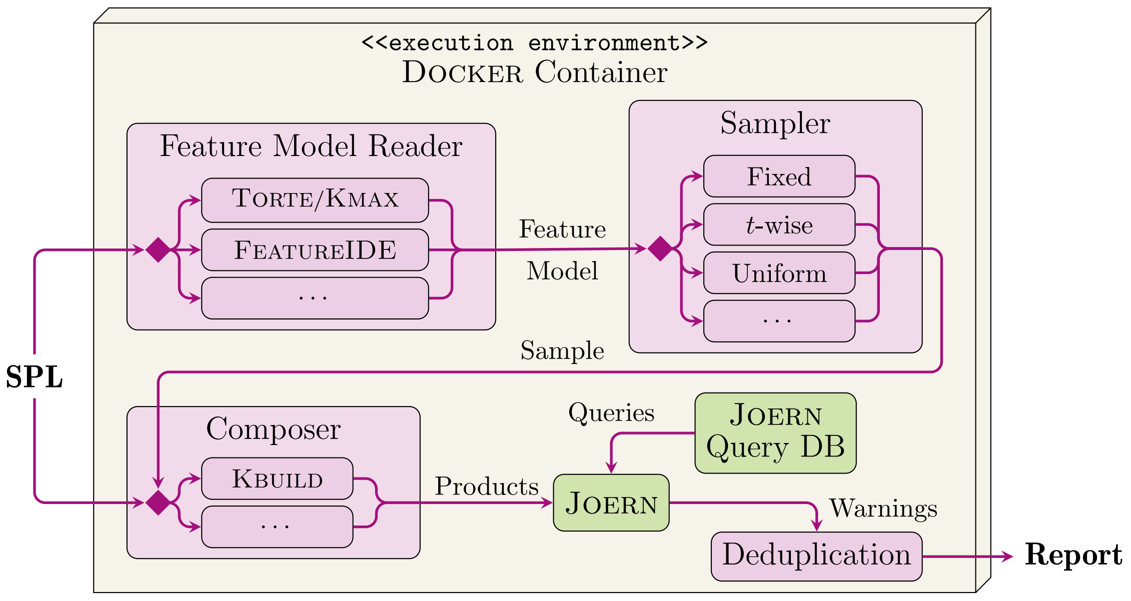


Figure 5.1: Initial structure of Vari-Joern

query database. Finally, the resulting warnings are deduplicated and reported to the user. Through the use of appropriate abstractions, Vari-Joern allows for multiple different implementations of the feature model reader, sampler, and composer components (cf. Figure 5.1). The analysis can thereby be tailored to certain subject systems and objectives. For instance, it is possible to read the feature model of a Kconfig-based system using the existing feature model reader built on Torte[3] and Kmax [Gaz17]. If another subject system employing the XML format used by FeatureIDE[4] to encode its feature model should be analyzed instead, the utilized feature model reader can simply be switched to the corresponding FeatureIDE implementation. Beyond the already available implementations of the components shown in Figure 5.1, a user can also add their own. Technically, this configurability of the approach extends to the analysis tool. However, as the name of the platform indicates, in this regard, Vari-Joern currently centers around Joern.

5.2 Sugarlyzer and its Integration into Vari-Joern

Integrating our design into Vari-Joern, the objective is to add the family-based strategy alongside the existing product-based strategy. As our design resolves around the Sugarlyzer framework of Patterson [Pat23], this requires a closer consideration of the framework's implementation.

The Sugarlyzer Framework

Sugarlyzer is implemented in Python and relies on Docker to maintain reproducibility across different execution environments [Pat23]. In summary, an analysis using Sugarlyzer is structured as follows: To start the analysis, the user interacts with a dispatcher that builds a dedicated Docker image for the user-specified SAST

[3]https://github.com/ekuiter/torte.
[4]https://featureide.github.io/.

tool (e.g., INFER [24g], or CLANG STATIC ANALYZER[24d]). This image not only contains a working installation of the SAST tool but also a full copy of SUGARLYZER. For the actual analysis, the dispatcher then starts a corresponding DOCKER container and forwards the user-specified analysis parameters to the SUGARLYZER installation contained within. Inside the container, the execution closely follows the idea of lifting by simulation. For variability encoding, SUPERC and SUGARC are applied to every C source file found within the system under analysis. The resulting variability-encoded source files are then analyzed using the chosen SAST tool and the output parsed into corresponding warnings. Lastly, these warnings are mapped back to their original location, assigned a presence condition, post-processed (e.g., to eliminate duplicates), and reported to the user as a single JSON report file.

Architectural Changes to Sugarlyzer

Since SUGARLYZER is implemented in Python, it cannot be integrated directly into VARI-JOERN's existing Java implementation. With more than 2,000 lines of code [Pat23], rewriting the framework in Java also involves considerable engineering effort. Consequently, we decided to add the code of SUGARLYZER alongside the existing Java implementation. As described above, by default, this code creates its own DOCKER container for the analysis. Considering this behavior, integrating SUGARLYZER into VARI-JOERN would require the creation of an additional sibling container. Since this can add additional overhead and complexity, we decided to slightly alter the structure of SUGARLYZER. Instead of interacting with the dispatcher, the analysis is now started by executing the main entry point of SUGARLYZER' analysis directly. While this small alteration eliminates the need for a sibling container, it places additional responsibility on VARI-JOERN. Before, the dispatcher always ensured a working installation of the chosen analysis tool within the used container. Since we bypass this functionality, this responsibility is now placed on the Dockerfile used by VARI-JOERN. However, given that VARI-JOERN already relies on JOERN, for the scope of this thesis, this does not necessitate any changes. Nevertheless, if other analysis tools should be supported by either of VARI-JOERN's analysis strategies in the future, their installation has to be ensured in the Dockerfile.

Integration into Vari-Joern

While the implementations of the two analysis strategies could be used separately from one another, for the user of the analysis platform, it would unnecessarily complicate the operation. Thus, to provide uniform and convenient access to both analysis approaches, we integrated support for SUGARLYZER and its family-based analysis into the command-line interface and configuration file used by VARI-JOERN's existing Java implementation. To achieve this, we reorganized the configuration file used for customizing the analysis, which was previously limited to the product-based strategy. Parameters specifying information on the subject system that is relevant to both analysis strategies (e.g., the path to the system's root directory) have been moved to a dedicated section of the file. In addition, for parameters that apply solely to one of the strategies, we established corresponding sections. To switch between the two strategies, we extended the command-line interface with a new mandatory argument. Selecting the product-based strategy through this argument, the remaining parts of the Java implementation are executed. Selecting the family-based strategy, on the

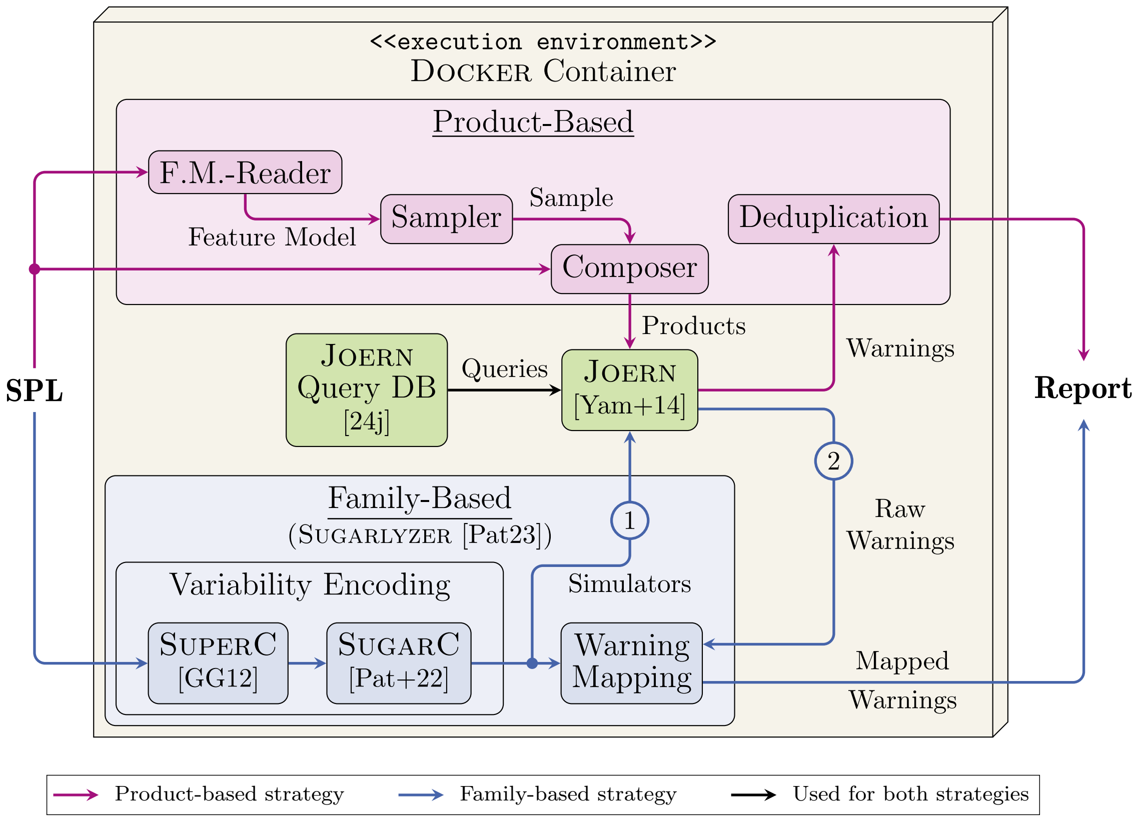


Figure 5.2: Updated structure of Vari-Joern

other hand, invokes Sugarlyzer with the information extracted from the relevant sections of the configuration file. The overall structure of Vari-Joern resulting from the integration of our family-based strategy is illustrated in Figure 5.2. As intended, this updated version of Vari-Joern provides convenient access to both analysis strategies, even though they are implemented in different programming languages.

5.3 Adding Support for Joern to Sugarlyzer

So far, we have detailed how we integrated our design, centered around the Sugarlyzer framework, into the existing analysis platform Vari-Joern. Since the Sugarlyzer framework currently only offers built-in support for the three popular NQ-SAST tools Clang Static Analyzer [24d], Infer [24g], and PhASAR[5] [24l; SHB19], another vital activity of our implementation focuses on adding support for Joern. Considering the implementation of the Sugarlyzer framework outlined in Section 5.2, adding Joern to the set of supported analysis tools involves two main activities[6] [Pat23]:

(1) A connector, connecting Sugarlyzer to Joern and allowing it to initiate an analysis on a variability-encoded source file, has to be supplied.
(2) A reader, responsible for parsing the output of Joern into corresponding objects, has to be provided.

[5]While PhASAR allows users to specify their own data-flow problems, this process is rather involved and requires implementing predefined interfaces [SHB19]. As this does not reflect the idea of Q-SAST as described by Li et al. [Li+24], we consider PhASAR a NQ-SAST tool.

[6]In the case of the unaltered Sugarlyzer framework, a Dockerfile specifying the container image used for the analysis with the new tool would also need to be added.

These activities directly correspond to establishing the edges highlighted with (1) and (2) in Figure 5.2. For both activities, the implementations already available in SUGARLYZER for the three supported NQ-SAST tools served as valuable inspiration.

Initiating an Analysis with Joern

We addressed (1) by introducing a new connector for JOERN. This connector takes a variability-encoded source file as input and calls the `joern-parse` utility of JOERN on the file to construct a corresponding CPG. The resulting CPG is passed to JOERN, together with a short Scala script specifying the operations the JOERN shell should perform in non-interactive mode. In this regard, JOERN is instructed to import the CPG, run the queries of the local query database, and emit the resulting warnings as a JSON file. This two-stepped approach could also be realized using a single command by invoking JOERN's built-in code scanner `joern-scan` [24i]. However, providing a custom execution script allows us to extract more information from a warning. While `joern-scan` outputs the most important information on a warning following a fixed format, our script is executed by the JOERN shell and thus has direct access to the objects representing the findings. As a result, we can emit additional important information, such as the description of the matching query, to the JSON file.

Parsing the Output of Joern

Addressing (2) is equally straightforward. We introduced a new reader that is responsible for parsing the output of JOERN into corresponding vulnerability warning objects that can then be used for further processing. For this purpose, our implementation parses one of the JSON files created by JOERN, extracts the identified warnings, and creates corresponding instances capturing their information. Since the general data type used by SUGARLYZER to represent variability warnings only encapsulates generic information on a warning, we additionally derived a new subtype. This subtype allows us to create warning instances that capture JOERN-specific information, such as the score (i.e., the severity) of the query that produced a particular warning.

5.4 Further Adjustments

Using the implementation outlined in Sections 5.2 and 5.3, we were able to obtain first results confirming the fundamental viability of our approach. Simultaneously, this implementation revealed a number of limitations of the SUGARLYZER framework and its associated tools. These limitations would need to be addressed to provide an effective and efficient analysis approach. Accordingly, in the following, we now present a brief overview of some of the most important limitations and the adjustments implemented to address them.

Retaining Identifiers

In unpreprocessed C, a single identifier can be used in multiple declarations [Pat+22]. As an example, recall that a variable or struct field can be declared with different types depending on the configuration of the program to ensure compatibility with different system architectures (cf. Section 3.3.1.4). Accordingly, for variability encoding, SugarC maintains a symbol table with entries for all variants [Pat+22]. To differentiate between multiple declarations of the same identifier, the symbol table stores a renaming of the identifiers [Pat+22]. This renaming is carried into the product simulator, where each identifier is renamed according to the fixed pattern `__identifier_numeric` [Pat+22]. In this regard, `identifier` refers to the original identifier, whereas `numeric` represents a unique integer.

Listing 5.1 exemplifies the renaming of identifiers in the context of the `foo` function (cf. Listing 1.1). While renaming ensures that symbols do not unintentionally interact within the product simulator, it means that static analyses relying on specific identifiers (e.g., the names of called functions) will miss obvious problems. For instance, in the context of Listing 5.1, a Joern query incorporating a search for calls to the `sink` function will never report any matches considering that the function was renamed to `__sink_3`. This problem has already been identified when Patterson et al. [Pat+22] applied the popular NQ-SAST tool Clang Static Analyzer [24d] to the product simulator produced by SugarC. Accordingly, a later improvement to SugarC saw the addition of a new `keep-mem` option that enabled identifiers of certain memory-related functions, such as `malloc` or `free`, to be retained in function calls [Pat23]. However, SugarC does not provide any means to whitelist other identifiers. This is suboptimal, considering that the Joern query database contains many queries relying on specific function identifiers not contained in the fixed whitelist used for `keep-mem`.

```
// #include ...
// ...
__static_condition_renaming("__static_condition_default_7",
                            "(defined CONFIG_PROCESS_INPUT)");
__static_condition_renaming("__static_condition_default_8",
                            "!(defined CONFIG_PROCESS_INPUT)");
// ...
void  (__foo_4) ()                                //M:L9:L18
{                                                 // L9
    int __x_5 = __source_1();                     // L10
    if ( __x_5 < __MAX_0)                         // L11
    {                                             // L11
        int __y_6 = 0;                            // L12
        if (__static_condition_default_7()) {
            __y_6 = 2 * __x_5;                    // L14
        }
        if (__static_condition_default_8()) {
            __sink_3(__y_6);                      // L16
        }
        if (__static_condition_default_7()) {
            __sink_3(__y_6);                      // L16
        }
    }                                             // L17
}                                                 // L18
```

Listing 5.1: A simplified extract of the output produced by SugarC for the `foo` function (cf. Listing 1.1)

As a solution to the aforementioned problem, we introduced a new `renaming-whitelist` option to the SugarC version used by Vari-Joern.[7] This new option works analogously to `keep-mem` but allows an arbitrary function identifier to be excluded from renaming. We leverage this functionality in Sugarlyzer by declaring an array of whitelisted function identifiers inside the connector responsible for initiating analyses with Joern. This array was manually populated with the function identifiers referenced in queries of the Joern query database. The entries are then excluded from renaming using the new `renaming-whitelist` option of SugarC. While this approach represents a simple solution to the problem, it carries certain limitations that need to be considered. First, the array of whitelisted function identifiers is tailored to the specific version of the Joern query database used for our implementation.[8] For other versions of the database (or user-provided queries), it thus may not whitelist every referenced identifier. In addition, `renaming-whitelist` can only be used with identifiers of functions defined outside the analyzed file (e.g., external library functions). Since the option only retains the original identifier for function calls, corresponding function definitions found within the same file are still renamed. If used incorrectly, the option can thus lead to mismatches between function calls and their definitions, which would nullify any previous attempt at behavior preservation of the product simulator. Furthermore, for an analysis tool, it would make it impossible to connect the affected function calls to their definitions.

Approximating Line Mappings

For lifting by simulation to be viable in practice, warnings reported on a product simulator need to be mapped back to their originating location within the unpreprocessed code (cf. Section 3.3.2.1). To simplify this process, SugarC appends specially formatted comments to the end of each relevant line of code in the simulator. This is exemplified by Listing 5.1. Lines that originate within the unpreprocessed C source code receive a comment indicating the original line number. All remaining lines (e.g., the ones added by SugarC to simulate variability) do not have an associated line within the unpreprocessed C code and thus do not receive a mapping. This works well for simple examples like the code shown in Listing 5.1. However, for large and complex source files that include many headers, we noticed deviations from the intended behavior. For instance, a for loop encapsulating a single statement in the same line as its condition is transformed to a loop spanning multiple lines if the statement contains complex macro substitutions. Crucially, the lines inside the loop do not receive individual mappings. Instead, the mapping is added to the closing bracket of one of the scopes introduced by SugarC to surround the now expanded loop. A similar behavior can be observed for other complex constructs spanning just a single line of code, such as the combined declaration and initialization of an array.

If the analysis tool reports a warning for a line whose associated line mapping is situated further down in the simulator, by default, Sugarlyzer is not capable of identifying the mapping and assigns an error tag to the corresponding field of the warning. A naive solution to this problem is to discard all warnings issued by the analysis tool for lines within the simulator that do not contain an explicit mapping. Since this can lead to warnings for dangerous vulnerabilities being discarded, it is

[7]https://github.com/KIT-TVA/superc.

[8]We used version 4.0.48.

certainly not ideal. As a result, we implemented an improved solution that searches subsequent lines for the presence of a line mapping. In summary, this solution works as follows: If a warning is associated with a line that does not contain a mapping, we check whether the line is contained within a function. This avoids searching for line mappings for warnings raised at a global level. Warnings at this level relate to structures, such as global variable definitions, structs, or typedefs. While these structures also frequently face inconsistent line mappings, we observed that simply searching subsequent lines proves insufficient on the global level. We then iterate over the subsequent lines, keeping track of opening and closing scopes (i.e., curly braces). Upon encountering a line mapping in a scope not opened after the line associated with the warning, the encoded line range is returned. In the worst case, where even subsequent lines do not expose a mapping for the line of interest, this can lead to the selection of a mapping related to another structure contained in the same or a parenting scope (e.g., a subsequent statement). Thus, a flag inside the line range object is set, indicating that the line mapping might only represent an approximated range within the unpreprocessed code. Our solution is therefore not perfect; it tries to address the effects rather than solving the problem at its source. It does, however, represent a distinct improvement to the status quo in which warnings whose mapping could not be established were simply assigned an error tag for their original line range. Moreover, improving the transformations performed inside SugarC such that line mappings are applied more consistently represents a challenging task that is beyond the scope of this thesis.

Expanding Automation

A central step during variability encoding is parsing (cf. Section 3.3.1.3). In general, for parsing to succeed, macro substitutions and header inclusions must be resolved beforehand (cf. Section 3.2.1.3). To this end, SugarC provides dedicated arguments that allow users to define paths for resolving and including headers, as well as specific macro definitions to use. Using Sugarlyzer, this information can be specified on a per-file basis in a JSON file tied to the system to be analyzed. Adding a new system thus entails creating a new JSON file for the chosen system and adding the information by hand. This can be a tedious process, considering that large systems have many source files, each potentially with a distinct set of required includes and macro definitions. However, as a demo script in the original Sugarlyzer's resources directory[9] demonstrates, most of the information can be automatically derived from the output of running Make[10] on the system to be analyzed. Inspired by this script, we aimed to ease the integration of new subject systems into the framework. Accordingly, we introduced two new methods into the abstract base class of every subject system:

(1) `run_make` is responsible for configuring the system to be analyzed with a fixed selection of features, for which the associated build process will include most of the source files. For this purpose, most systems provide a predefined configuration selecting most, if not all, available features (e.g., `allyesconfig` for the Linux kernel [Aba+17]). The function is also responsible for building

[9]`resources/programs/axtls/axtlsFileBuilding.py`.
[10]https://www.gnu.org/software/make/.

this configuration with Make and writing the resulting output containing the concrete compile calls to a dedicated file.[11]

(2) `parse_make_output` is then responsible for identifying compile calls in the output produced by Make and associating the includes contained within with the respective source file.

Information that could not be extracted from running make (e.g., includes of files not compiled in the chosen configuration) can then, as before, be added to the JSON file associated with the system. Even though establishing and building a configuration of a system incurs additional performance overhead, based on our experience, this overhead is negligible compared to variability encoding and the actual analysis. In addition, in the case of certain systems, establishing a configuration creates additional directories and files (such as headers) required for parsing. Therefore, it may even be infeasible to omit the step.

Another key factor during variability encoding is to prevent the expansion (i.e., consideration) of every macro found within the unpreprocessed code. Otherwise, even execution paths tied to invalid products (i.e., products whose construction is prevented by the build system) will be incorporated in the product simulator [Pat23]. Although SugarC provides a `restrictConfigToPrefix` option that allows expansion to be limited to macros exhibiting a certain prefix, this functionality does not allow the constraints introduced by an SPL's feature model to be captured. Furthermore, it does not provide a way to handle feature-related macros of non-boolean type. Sugarlyzer addresses these issues by incorporating a custom header file into the subject system before analysis. This header takes the place of the header generated by Make when establishing a new configuration (usually `config.h`). By default, the header generated by make contains the macro definitions resulting from the particular feature selection and is included in all source files making use of feature-controlled conditional compilation. Sugarlyzer changes the role of the header and wraps all configuration-related macros of the system in newly created boolean preprocessor conditionals respecting the constraints of the associated feature model. Since these new conditionals rely on macros sharing a common prefix, SugarC's expansion can be restricted accordingly using its `restrictConfigToPrefix` option.

For the creation of the header, Sugarlyzer provides kgenerate [Pat23]. kgenerate is a tool that leverages Gazillo et al.'s [Gaz17] kmax tool suite to parse the analyzed system's Kconfig files, extracting information on configuration variables, their constraints, and default values [Pat23]. kgenerate uses this information, together with a user-provided format file, to generate a corresponding header. Additionally, it emits a JSON file mapping the newly introduced macros to their original counterpart and the represented value. Although kgenerate automates the creation of the custom header and its associated mapping, for the analysis of a new subject system, Sugarlyzer requires manual execution of the tool. The same applies to updates to a system's feature model. To reflect the changes in the feature model in the header and its mapping, kgenerate has to be invoked manually after each modification. Therefore, to ease the handling of feature model updates and integration of other systems, we aimed to further automate the overall process. For this purpose, we incorporated a call to kgenerate into Sugarlyzer. Using the information on

[11]Depending on the system, it may suffice to echo the recipes of the build process using Make's `--just-print` option instead of executing them.

the configuration header's location specified in the JSON file tied to the analyzed system, this enables us to automatically insert the header created by KGENERATE.

Extended Caching

By default, SUGARLYZER provides caching for the variability encoding process. This means that before calling SUGARC, SUGARLYZER checks whether there exists a variability-encoded source file in the cache directory that was created using the same unpreprocessed source file and options. If this is the case, variability encoding is skipped and the cached source file is loaded for further analysis by the chosen analysis tool. For large SPLs containing multiple hundreds to thousands of large source files, this avoids repeating variability encoding for all files, even if just a single file changed between analyses. While this can drastically cut down on the resources required for consecutive executions of the analysis approach, our first implementation indicated that the actual analysis by the chosen analysis tool can be just as resource demanding as variability encoding. Accordingly, we extended the caching of SUGARLYZER to the output created by the analysis tool. Analogously to the existing caching functionality, before invoking the analysis tool, we check whether the caching directory contains a report file for the considered variability encoded source file. If this is the case, the cached report is loaded and passed directly to the corresponding reader without having to invoke the analysis tool.

6. Evaluation

For the practical viability of the presented family-based vulnerability discovery approach, two aspects are of particular interest. First, the analysis approach should be effective, meaning that its results should expose vulnerabilities across the code of real-world SPLs. If this is not the case, the analysis adds no value to the overall development effort, effectively rendering it superfluous. Second, the approach should be efficient, meaning that its resource demand should be manageable in practice. In this regard, even a highly effective approach loses its viability once its execution becomes too costly. Considering these aspects, we formulate the two research questions ***RQ-1*** and ***RQ-2*** shown below:

RQ-1:	*How effective is our family-based analysis in finding potential occurrences of common vulnerability types across an SPL?*

RQ-2:	*How efficient is our family-based analysis with regard to execution time and storage demand?*

The goal of this chapter is to answer these research questions. In pursuit of this goal, we first describe our experimental setup (Section 6.1). We then present the results obtained using this setup (Section 6.2) and discuss their implications (Section 6.3). Lastly, we give an overview of the threats to the validity of our analysis and its evaluation (Section 6.4).

6.1 Experimental Setup

For the evaluation of the proposed analysis approach with regard to ***RQ-1*** and ***RQ-2***, three aspects need to be considered. First, a baseline that acts as a reference point for the effectiveness of the approach needs to be established (Section 6.1.1). Furthermore, subject systems on which the approach is going to be evaluated need to be selected (Section 6.1.2). Finally, a method for interpreting and comparing results on the subject systems is required to derive meaningful insights (Section 6.1.3). Our setup for each of these aspects is outlined below.

6.1.1 Baseline

In the context of vulnerability discovery, there are two common alternatives for establishing a baseline. First, a code base containing a known set of vulnerabilities can be employed. Alternatively, the results of another established analysis approach can be used. In the following, we briefly describe both alternatives and discuss our choice.

Code Bases with Known Vulnerabilities

A straightforward way to establish a baseline is to consider a code base where either all vulnerabilities or a selected subset thereof are known. In this regard, identifying earlier versions of real-world systems still containing unpatched vulnerabilities can be difficult and involve a considerable amount of manual work. To ease this process, there are prefabricated benchmarks containing manifestations of various vulnerabilities. While these benchmarks can be very extensive,[1] their main weakness is that the contained programs generally represent artificial examples instead of real-world programs. Another problem specific to the domain of SPLs is that there is no benchmark dedicated to VIVs. To the best of our knowledge, the only effort in this direction has been undertaken by Abal et al. [Aba+17; ABW14] and Mordahl et al. [Mor+19]. However, their benchmarks have both not been updated in the last five years and are limited in scope, comprising only 98 and 77 examples, respectively. Furthermore, both benchmarks do not explicitly focus on VIVs but contain examples of general variability bugs.

Results of Alternative Approaches

Another way through which a baseline can be established is to consider results of an alternative analysis approach. This has the advantage that the baseline can be established on any real-world system compatible with the alternative analysis approach, thereby avoiding the consideration of artificial examples. Additionally, it allows the resource demand for the construction of the baseline to be captured. Contrary to a code base containing a known set of vulnerabilities, the efficiency of an analysis can therefore also be judged. It is, however, important to remember that static analysis is generally limited to providing approximate results [CM04]. A baseline constructed from an alternative approach will therefore be subject to imprecision. Depending on the concrete extent of this imprecision, evaluating another analysis approach may be difficult.

Our Choice

For the purpose of this thesis, we decided to pursue the option of using the results of another analysis approach as the baseline. This choice allows us to avoid having to identify earlier versions of the considered subject systems still containing unpatched vulnerabilities. In the context of SPL research, it is common to use results of product-based approaches as baselines for the evaluation of a family-based approach [Ape+13b; Bra+12; von+18]. To this end, the simplest strategy would be to use the results of an exhaustive product-based strategy, analyzing each product

[1]The popular JULIET test suite [24k] comprises more than 64,000 test cases.

in isolation. However, since we aim to evaluate our approach on real-world systems exhibiting large configuration spaces, this would represent an infeasible task.[2] Instead, we focus on results produced by optimized product-based strategies. These strategies make use of sampling to reduce the number of products to analyze (cf. Section 2.1). Since the goal of this thesis is not to compare the capabilities of modern Q-SAST tools, it is vital that any baseline approach used for our evaluation leverages JOERN. In this regard, VARI-JOERN already provides us with a product-based analysis approach built around JOERN. Consequently, we chose this approach for the creation of our baseline. Specifically, we chose the uniform sampling strategy that samples products uniformly across an SPL's feature model.

6.1.2 Subject Systems

Using an alternative analysis approach as a baseline allows us to utilize real-world systems for the evaluation. However, there are a number of criteria a particular system must fulfill:

- **(C1)** **Variable**. Our focus is on the analysis of variable software (i.e., SPLs). Non-variable systems encompass just a single software product and can be analyzed using conventional Q-SAST tools. Their analysis does subsequently not face the same issues as the analysis of SPLs and is beyond the scope of this thesis.
- **(C2)** **Open-Source**. Analyzing an SPL in its entirety requires access to the source code of all products, as well as its variability model. Therefore, having access to only a select number of source files or a precompiled binary proves insufficient.
- **(C3)** **Mostly Implemented in C**. For the scope of this thesis, we focus on the analysis of SPLs implemented in C (cf. Section 1.2). Subject systems implemented in other programming languages do therefore not benefit from our approach.
- **(C4)** **Moderate to Large Size**. We avoid small SPLs, considering that their limited complexity prevents us from gaining conclusive insights into the capabilities of our approach. On the other end, we avoid very large SPLs, considering that our solution represents a first prototype that may still face scalability issues.
- **(C5)** **Practical Relevance**. Subject systems should exhibit a certain degree of practical relevance. Otherwise, results may not reflect real-world performance and thus become irrelevant. To this end, we consider systems that were used as subject systems in recent SPL studies conducted by other researchers.
- **(C6)** **Configuration Management via Kconfig**. SUGARLYZER makes use of KGENERATE to create a custom header limiting the macro expansion performed by SUGARC according to a system's feature model (Chapter 5). KGENERATE is realized using the KMAX tool suite [Gaz17] and therefore expects a KCONFIG-based system as input. Non-KCONFIG-based systems are therefore not supported by our approach.

Selected Subject Systems

Table 6.1 shows a characterization of the subject systems selected based on the aforementioned criteria. For all three systems, we selected the latest stable release available as of September 2024. C lines of code (column C-LoC) were measured using

[2]Recall that the number of products derivable from an SPL may be exponential in the number of features [Lie+13; Thü+14; von+18] (cf. Section 2.1).

cloc[3] version 1.98, considering only the count for C code lines. For approximating the number of features, we followed an approach similar to Abal et al. [Aba+17] and considered the number of unique configuration options codified in the systems' Kconfig files. We extracted this number by counting the corresponding entries in the output produced by the Kconfig parser kextract, which is part of Kmax (version 4.7.3) [Gaz17]. To gain insights into the scalability of our approach, we selected a small (axTLS), a moderate (Toybox), and a large (BusyBox) SPL as subject systems. In this regard, the size of the systems, measured by C lines of code and the number of features, nearly triples from axTLS to Toybox. A similar relationship can be observed between Toybox and BusyBox. Lastly, all systems have been used in at least three recent studies in the field of SPL research.

System	Type	Version	C-LoC	Features	Academic Use
axTLS [24a]	Client/server TLS library	2.1.5 (2019)	17,556	95	[Mor+19; Oh+19; Pat+22]
Toybox [24p]	Collection of Linux command line utilities	0.8.11 (2024)	58,127	355	[Mor+19; Oh+19; Pat+22]
BusyBox [24b]	Collection of UNIX utilities	1.36.1 (2023)	182,966	1,079	[Ios+17; Mor+19; Oh+19; Pat+22; Pet+23; von+18]

Table 6.1: Characteristics of the selected subject systems

6.1.3 Methodology

Using the aforementioned baseline and subject systems, the methodology applied to evaluate our family-based analysis approach is illustrated in Figure 6.1. In essence, it consists of two steps that are performed on a per-subject system basis.

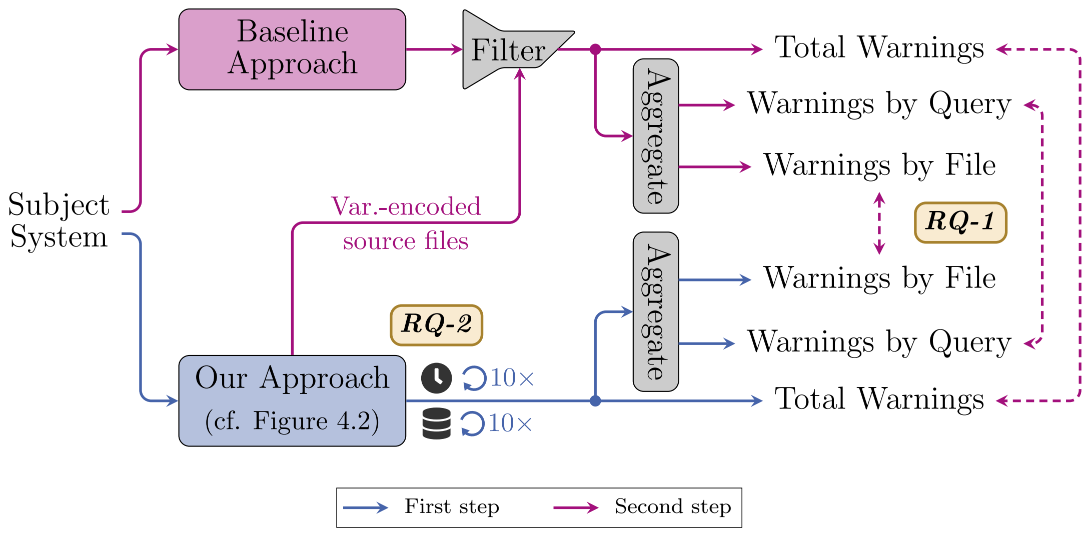


Figure 6.1: Overview of the methodology employed for the evaluation

[3]https://github.com/AlDanial/cloc.

Step 1: Data Collection

As a first step, we run our analysis approach on the subject system (cf. Table 6.1) and collect metrics relating to its effectiveness and efficiency. In this regard, we execute VARI-JOERN within its provided DOCKER container, which uses an image based on UBUNTU 24.04.1. The container is, in turn, executed on a virtualized server equipped with a 16-core AMD EPYC processor and 32GB of RAM, running UBUNTU 22.04 LTS. To prevent a potential bias in our findings, we clear the caches of all intermediary results, such as variability-encoded source files, before execution.

For judging the effectiveness of our approach (***RQ-1***), we consider the total number of reported vulnerability warnings. To this end, we collect the number of entries contained in the final JSON report file produced for the analyzed subject system. This number represents the total number of warnings reported across the whole SPL. To allow for more fine-grained insights, we additionally aggregate the warnings extracted from the report on a per-file and per-query basis.

For the evaluation of our approach's efficiency (***RQ-2***), we assess its performance with regard to total execution time and overall storage demand. We measure the execution time of our approach using GNU TIME.[4] For this purpose, we use the maximum number of concurrent workers for SUGARC and JOERN possible without exceeding the main memory available in our execution environment. As the total execution time is a measure that is typically subject to variability due to factors like the operating system's scheduling policy, we repeat the measurement across ten consecutive executions and report the resulting arithmetic mean. For the storage demand, we consider the maximum demand involving all files created as part of our analysis approach. To this end, we measure the increase in the DOCKER container's file system usage using the `df` utility of the GNU COREUTILS[5] utility collection. As logging files can artificially increase our approach's storage demand, we disable the creation of corresponding files. Furthermore, since temporary files created by the operating system or other processes can lead to additional storage demand, we once again repeat the measurement across ten consecutive executions and report the arithmetic mean.

Step 2: Comparison with the Baseline

In a second step, we then compare the collected metrics relating to effectiveness with results produced by VARI-JOERN's product-based strategy, which serves as our baseline (Section 6.1.1). Results for the baseline are generously provided to us by other researchers and were collected using an execution environment identical to ours. As with our approach, we aggregate individual warnings of the baseline on a per-query and per-file level. To allow for a fair evaluation of the results, we filter out warnings on files that could not be successfully variability encoded by SUGARC before this aggregation takes place. This then allows us to judge our approach's effectiveness in total, on a per query, and on a per-file level.

6.2 Results

In this section, we report the results of our evaluation. As we have described in Section 6.1.3, our evaluation consists of two steps. First, we execute our family-

[4]https://www.gnu.org/software/time/.
[5]https://www.gnu.org/software/coreutils/.

based analysis approach on the three selected subject systems and collect the chosen metrics. The results of this step are described in Section 6.2.1. The second step of the evaluation then involves relating the data gathered for our approach to the data obtained for the optimized product-based strategy serving as our baseline. The results of this step are described in Section 6.2.2.

6.2.1 Results for the Selected Subject Systems

For all three subject systems, Table 6.2 summarizes the metrics collected for a full analysis using our family-based approach. The $|F|$ column lists the total number of C source files contained in the system, whereas the $|F_{sim}|$ column indicates the number of source files for which SugarC was able to produce a product simulator. $|F_{sim}|$ therefore excludes all files for which variability encoding either outright failed or exceeded the ten minute timeout issued for calls to SugarC by Sugarlyzer. The $|W_{raw}|$ column shows the number of individual warnings reported by Joern across all product simulator files in F_{sim}. The number of warnings that remain after warning mapping are shown in the $|W_{mapped}|$ column. This number is smaller than $|W_{raw}|$ since Sugarlyzer's warning mapping component already performs advanced post-processing of warnings. For instance, warnings relating to the same issue of the unpreprocessed code are merged into a single warning with an extended presence condition. As warning mapping constitutes the last step of the analysis, $|W_{mapped}|$ also represents the number of alarms in the final report file. In this regard, the $|Q|$ column reflects the number of queries from the Joern query database exhibiting at least one match in the final report. Considering the resource demand of our approach, $\bar{t}_{exec}$ describes the total execution time, whereas $\bar{s}_{max}$ describes the maximum storage demand. For both measures, we report the arithmetic mean across ten executions (cf. Section 6.1.3). In this regard, we always used five concurrent workers for all three systems.

System	$\|F\|$	$\|F_{sim}\|$	$\|W_{raw}\|$	$\|W_{mapped}\|$	$\|Q\|$	$\bar{t}_{exec}$	$\bar{s}_{max}$
axTLS	45	36	181	119	9	400 s	105 MB
Toybox	291	278	784	494	10	4,265 s	1,530 MB
BusyBox	685	619	5334	670	9	8,513 s	3,494 MB

Table 6.2: Results of the proposed family-based analysis approach on axTLS, Toybox, and BusyBox ($|F|$ = #C source files, $|F_{sim}|$ = #variability-encoded C source files, $|W_{raw}|$ = #total warnings by Joern, $|W_{mapped}|$ = #warnings after warning mapping, $|Q|$ = #matched queries, $\bar{t}_{exec}$ = average execution time, $\bar{s}_{max}$ = average storage demand)

6.2.2 Results in Relation to the Baseline

Comparing the results of the two analysis strategies offered by Vari-Joern, for our family-based approach, we use the data presented in Section 6.2.1. For the optimized product-based approach, we use data for the uniform sampling strategy made available to us by other researchers. While we set out to compare the results on all three of our selected subject systems, limited support for Toybox in Vari-Joern's product-based analysis approach restricted the available baseline data to axTLS and BusyBox. In the following, we thus concentrate on a comparison of the results produced on these two subject systems. Detailed results for the analysis of Toybox using our approach can be found in Appendix A.2.

Baseline Results

Table 6.3 summarizes the baseline results made available to us. Since an optimized product-based analysis strategy considers only a subset of all products derivable from an SPL (cf. Section 2.1), the $|S|$ column indicates the sample size used. This measure directly reflects the number of individual products considered during the analysis. $|W|$ describes the total number of vulnerability warnings found in the final report and $|Q|$ the number of queries with at least one match. The $|W_{sim}|$ and $|Q_{sim}|$ columns repeat these measures, considering only warnings on C source files for which SugarC of the family-based approach was able to produce a product simulator. For the evaluation of our approach's effectiveness, we use $|W_{sim}|$ and $|Q_{sim}|$. Since $|W|$ and $|Q|$ consider warnings on source files for which variability encoding failed, the results would otherwise not allow for a fair comparison of the approaches' vulnerability discovery capabilities.

System	$\|S\|$	$\|W\|$	$\|Q\|$	$\|W_{sim}\|$	$\|Q_{sim}\|$
axTLS	1,000	95	8	95	8
BusyBox	1,000	1,356	11	971	11

Table 6.3: Results of Vari-Joern's product-based analysis approach using uniform random sampling that serves as our baseline ($|S|$ = sample size, $|W|$ = #warnings, $|Q|$ = #matched queries, $|W_{sim}|$ = #warnings on variability-encoded source files, $|Q_{sim}|$ = #matched queries on variability-encoded source files)

Vulnerability Warnings by Query

Using the aforementioned data on the two analysis approaches, we are able to compare their capabilities on axTLS and BusyBox. To this end, Figure 6.2 visualizes the approaches' ability to identify matches to the queries of the Joern query database within the subject systems. Specifically, Figure 6.2 illustrates the number of reported vulnerability warnings aggregated by query. Queries with at least one match in the results of either analysis approach are shown on the y-axis. To improve clarity, queries without any matches are omitted. Each query on the y-axis is depicted by its name and associated numerical score, as specified in the Joern query database. In this regard, the score is intended to provide a coarse estimate of the severity of the issue modeled by a query. The x-axis then shows the absolute number of warnings associated with a particular query for both analysis approaches and subject systems.

Overall, depending on the subject system and query, the number of vulnerability warnings varies. For the *free-field-no-reassign* and *constant-array-access-no-check* queries, our approach was not able to identify any matches. In contrast, the product-based approach on BusyBox identified one match for *free-field-no-reassign* and eight matches for *constant-array-access-no-check*. Conversely, on axTLS, the family-based approach identified one match for the *malloc-memcpy-int-overflow* query, for which the baseline did not discover any matches. For the remaining queries, both analysis approaches were able to identify manifestations. While the family-based approach identified more matches on axTLS, the product-based approach detected a greater number of matches on BusyBox.

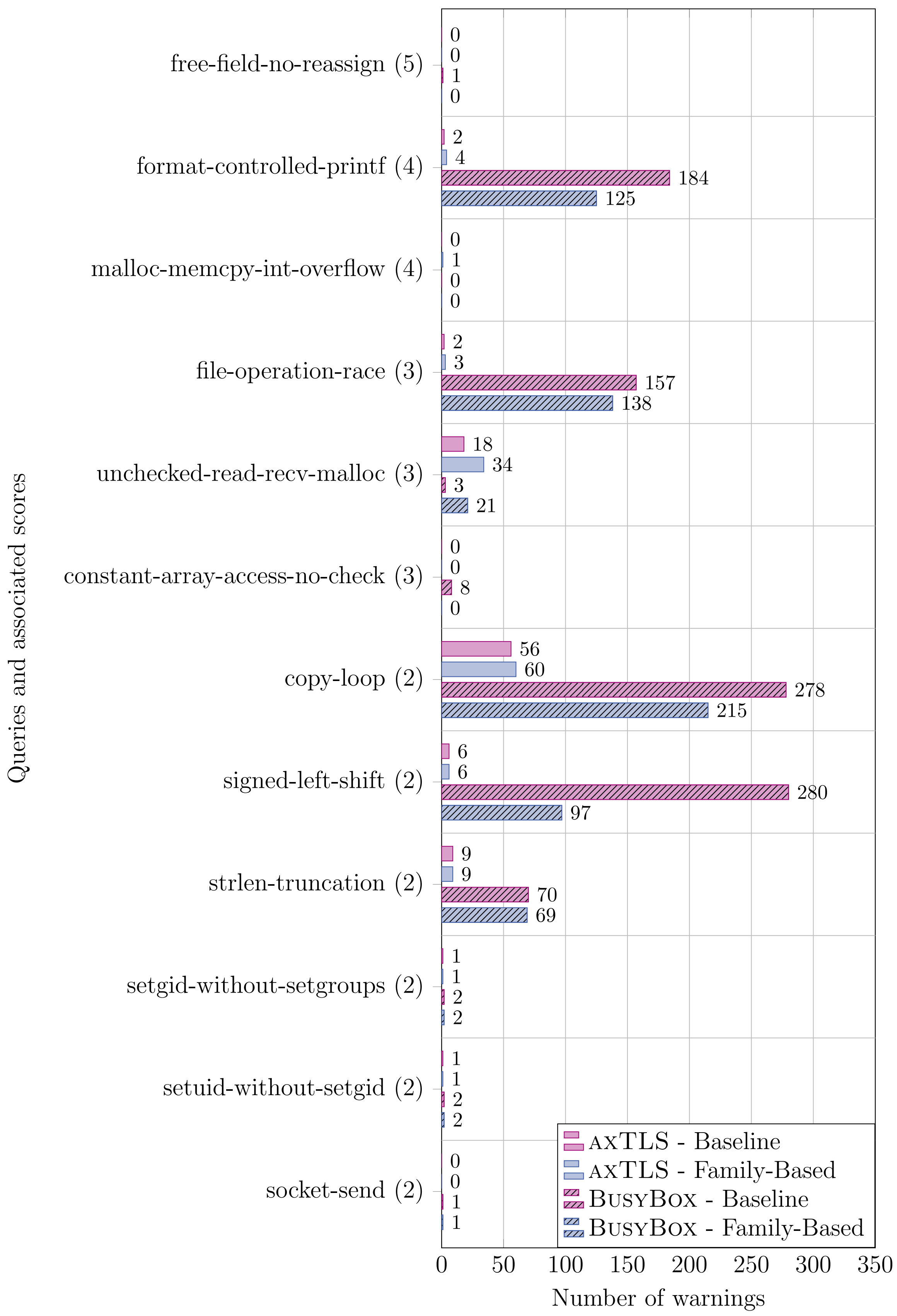


Figure 6.2: Warnings reported on axTLS and BusyBox aggregated by query

Vulnerability Warnings by Source File

In Figure 6.3 we show the distribution of vulnerability warnings across the individual source files of the subject systems. On AXTLS, the family-based approach statistically reports more warnings per source file as indicated by the average and median, as well as the lower and upper quartiles. While the two approaches share the same median on BUSYBOX, the averages, upper quartiles and maxima suggest that the product-based approach is capable of identifying more matches per source file on this subject system.

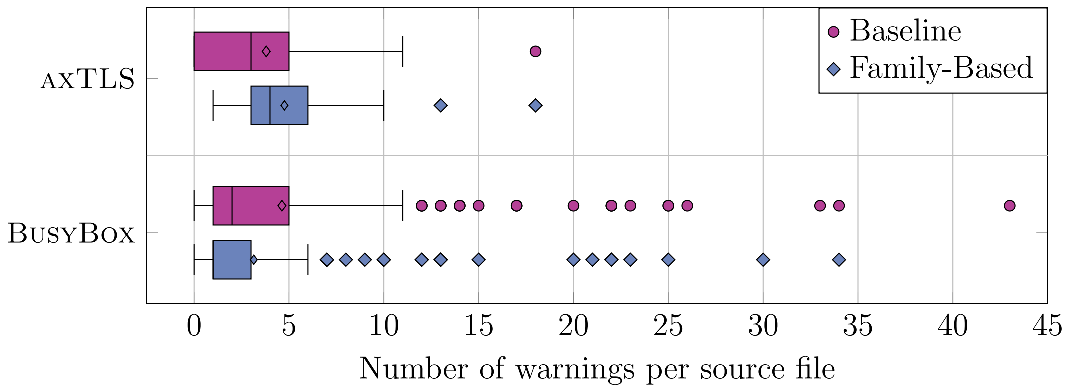
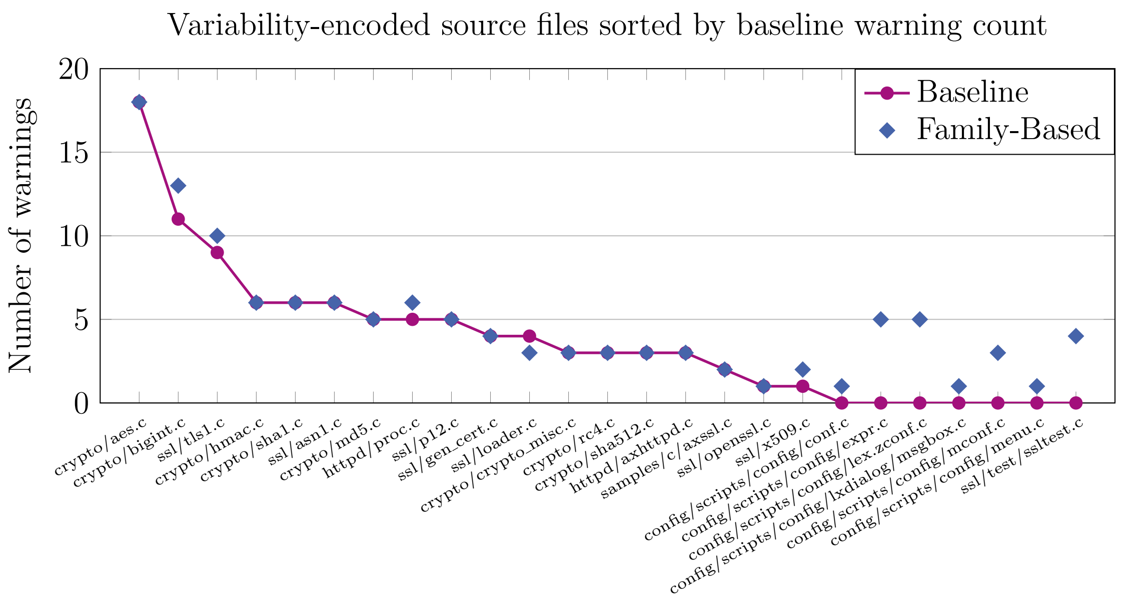


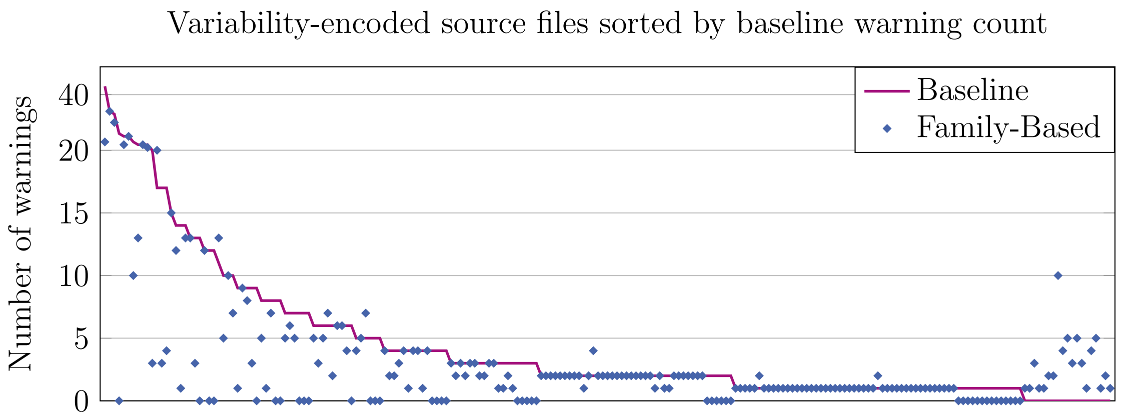
Figure 6.3: Distribution of vulnerability warnings across the source files of AXTLS and BUSYBOX

Providing more detail on the aforementioned distribution, Figure 6.4 illustrates a fine-grained comparison of the number of vulnerability warnings per source file. To this end, Figure 6.4a shows the results for AXTLS, while Figure 6.4b shows the results for BUSYBOX. For both figures, the x-axis depicts all source files on which either of the approaches reported at least one vulnerability warning. To establish a clear trend line for the warnings reported by the baseline approach, the source files on the x-axis are sorted in descending order by the number of warnings reported by the product-based approach. Since AXTLS contains only 25 C source files for which either approach reported warnings, the x-axis of Figure 6.4a is labeled with the relative paths of the corresponding files. For BUSYBOX, there are over 200 source files with at least one warning. Therefore, to improve clarity, we omit the names of individual source files in Figure 6.4b. Lastly, the y-axis of both figures shows the total number of all warnings reported on a particular source file by the two analysis approaches.

Considering the results for AXTLS (Figure 6.4a), except for `ssl/loader.c`, the family-based approach is always able to match the number of vulnerability warnings reported by the baseline. Notably, for 11 source files, the family-based approach is able to exceed the number of warnings reported by the baseline approach. In the case of BUSYBOX (Figure 6.4b), results were significantly more mixed. Out of a total of 213 files with at least one warning, 26 files saw the family-based approach exceed the numbers of warnings reported by the product-based approach. On 97 files, both strategies reported the same number of warnings. Crucially, on the remaining 90 files, the baseline reported a higher number of warnings.



(a) Results for AXTLS



(b) Results for BusyBox

Figure 6.4: Reported warnings on AXTLS and BusyBox aggregated by source file

6.3 Discussion

In this section, we discuss the results presented in Section 6.2. We start with a consideration of our vulnerability discovery approach's effectiveness, ultimately addressing ***RQ-1*** (Section 6.3.1). Finally, we consider our approach's efficiency to answer ***RQ-2*** (Section 6.3.2).

6.3.1 Effectiveness (RQ-1)

To assess the effectiveness of our family-based approach, we consider the number of raised vulnerability warnings. In general, as shown in Table 6.2, our approach is capable of identifying vulnerability warnings for all three subject systems. Comparing the total numbers between our approach and the product-based baseline (cf. Table 6.3), we observe that the family-based approach identifies more potential vulnerabilities on AXTLS (119 vs. 95). Conversely, on BusyBox, the family-based approach falls short and raises significantly fewer warnings than the baseline (670 vs.

971). Both observations are supported by the distribution of vulnerability warnings per file shown in Figure 6.3. Considering the matches to queries of the JOERN query database, we observe that our approach matches one additional query on AXTLS (9 vs. 8), while it matches two queries less on BUSYBOX (9 vs. 11). In the following, we go into more detail and discuss potential reasons for these findings.

6.3.1.1 Matched Queries on axTLS

On AXTLS, both analysis approaches are able to detect matches to eight queries of the JOERN query database. However, as mentioned above, the family-based approach also identifies a match to one additional query. As indicated by the distribution of warnings per query illustrated in Figure 6.2, this is the *malloc-memcpy-int-overflow* query. This query checks whether a copy operation into a newly heap-allocated buffer might not fill the entire buffer, leaving uninitialized data that could be exploited by an attacker. It therefore constitutes a rather serious vulnerability pattern, as evidenced by its score of 4. The particular manifestation of the pattern identified by the family-based approach is contained in a function wrapped in a preprocessor conditional. It is therefore not part of every product of the SPL, effectively constituting a potential VIV. While the uniform sample of the baseline manages to sample multiple products for which the function is included, to our surprise, JOERN was not able to detect the match on any of these products. While the concrete reasons for this behavior are still unknown to us, it appears to be caused by a bug in JOERN triggered by the full analysis of specific products of AXTLS. This would explain why the family-based approach, which calls JOERN not for entire products but for product simulators of individual source files, is able to reliably identify the match.

6.3.1.2 Vulnerability Warnings on axTLS

Overall, on AXTLS, the family-based approach issued a greater number of vulnerability warnings compared to the baseline. This is reflected in the number of warnings reported per query, where for all queries, the family-based approach matches or exceeds the number set by the baseline (cf. Figure 6.2). Additionally, as the distribution of warnings per source file illustrated in Figure 6.3 indicates, the family-based approach statistically issues more warnings per source file. These observation can best be explained by considering the number of warnings per source file as shown in Figure 6.4a.

Files not Considered by the Baseline

It is striking that the family-based approach reported vulnerability warnings for seven files, for which the product-based approach did not issue any warnings. This is due to a limitation of the product-based approach. While it aims at an analysis of all source files, it decides on the files to incorporate into a sampled product by analyzing the compile calls issued for the concrete configuration by MAKE. Given that these calls do not involve source files relating to test cases or the system's KCONFIG infrastructure, corresponding files are not added to individual products and thus evade analysis. As these files are still analyzed by the family-based approach, we see an increase in warnings relating to four queries: *unchecked-read-recv-malloc* (16 matches), *format-controlled-printf* (2 matches), *copy-loop* (1 match), and *strlen-truncation* (1 match).

Joern and its Queries

Beyond the seven source files related to tests and Kconfig, there are other files on which the family-based approach reports a greater number of vulnerability warnings (cf. Figure 6.4a). The additional warning on `ssl/x509.c` represents the match to the *malloc-memcpy-int-overflow* query described earlier during our discussion of the matched queries on axTLS. Recall that this manifestation is likely missed by the product-based approach due to a bug in Joern. The additional match on `httpd/proc.c` can, on the other hand, be attributed to the *file-operation-race* query. This query checks for the presence of two file operations on the same file path. As such a pattern can lead to a race condition that may be exploitable by an attacker, it constitutes a potential vulnerability. At first glance, the query does not appear to model a particular syntactical structure of a program that might be changed during variability encoding. As a result, it did not stand out during our examination of the queries in the Joern query database (cf. Section 4.4.2). However, in essence, it merely searches function bodies for two calls to file operations sharing the same argument for the file path. Thus, since no semantic relationship between the two calls is considered, the query is susceptible to changes introduced during variability encoding. In the case of `httpd/proc.c`, the code duplication introduced into the product simulator by SugarC duplicated a code region containing a call to a file operation. The duplicated call then resulted in a false match for the query.

Line Mapping Deficiencies

The family-based approach issued additional warnings relating to the *copy-loop* query on `crypto/bigint.c` and `ssl/tls1.c`. A closer inspection revealed that the two additional matches on `crypto/bigint.c` constituted duplicates. Due to inconsistencies in the comments emitted into a product simulator by SugarC (cf. Listing 5.1), these duplicates received slightly different line mappings and thus slipped through Sugarlyzer's deduplication. The remaining match on `ssl/tls1.c` was raised on source code resulting from the substitution of an externally defined macro. Since this code was not present in the unpreprocessed source code, the warning received an invalid line mapping. However, as the referenced code region actually contained an array access typical of a copy loop, the sanity check we have added to Sugarlyzer was not able to prune the warning.

Insufficient Variability Encoding

Due to the aforementioned reasons, the family-based approach reported a greater total number of vulnerability warnings than the product-based approach. However, it is worth noting that it fell below the baseline on a single source file. In this regard, on `ssl/loader.c`, the approach missed an obvious instance of the *strlen-truncation* query. A closer examination of the variability-encoded file produced by SugarC revealed that variability-encoding of the corresponding code region was unsuccessful. Instead of transforming the region into corresponding simulator code, SugarC inserted an artificial function call indicating a presumed type error in the enclosing if statement during parsing. As a result, the vulnerability pattern was not present in the product simulator and could thus not be detected by Joern.

6.3.1.3 Matched Queries on BusyBox

On BusyBox, both analysis approaches are able to detect matches to nine queries. Crucially, beyond these nine queries, the product-based approach identified one match to the *free-field-no-reassign* and eight matches to the *constant-array-access-no-check* query. For these queries, our family-based approach yielded no warnings (cf. Figure 6.2). On closer inspection, we found that both queries rely on the `name` property of certain CPG nodes. Further, we noticed that the parentheses added around all identifiers in a product simulator by SugarC caused this property to always be an empty string. As a result, corresponding parts of the queries resulted in string comparisons with an empty string, eliminating any potential matches early. We reported this behavior to the Joern developers, who confirmed it as a bug. This bug has since been fixed and merged into Joern's main code line.[6] Consequently, using newer releases of Joern for the analysis, our approach should be able to identify matches for the *free-field-no-reassign* and *constant-array-access-no-check* queries.

6.3.1.4 Vulnerability Warnings on BusyBox

On BusyBox, the family-based approach issued significantly less vulnerability warnings, totaling just above two thirds of the warnings identified by the product-based baseline (cf. Tables 6.2 and 6.3). As indicated by the distribution shown in Figure 6.3, a similar trend can be observed with regard to the number of warnings raised on individual source files. Furthermore, on six of the nine queries matched by both approaches, the product-based approach issued more warnings (cf. Figure 6.2). Due to the large number of source files of BusyBox and the large discrepancies exhibited by the two strategies, we are not able to analyze every instance where the two strategies deviate by hand. However, based on our experience and a manual examination of a select number of instances, we found that many of the deviations can be attributed to the same classes of problems identified for axTLS.

Files not Considered by the Baseline

The family-based approach issues a total of 40 warnings on files related to BusyBox's Kconfig infrastructure. As described during our discussion of the vulnerability warnings on axTLS, these files are not incorporated into concrete software products and thus not analyzed by the product-based approach. They therefore explain most of the deviations in the number of warnings exhibited by the rightmost files of Figure 6.4b. Another interesting insight is that warnings on Kconfig files are the main reason why the family-based approach issues a significantly larger number of warnings for the *unchecked-read-recv-malloc* query than the baseline. Of the 18 additional matches reported, 15 stem from Kconfig-related files. The remaining 25 warnings on Kconfig files contribute matches to four queries: *file-operation-race* (10 matches), *format-controlled-printf* (9 matches), *strlen-truncation* (5 matches), and *copy-loop* (1 match).

Joern

As mentioned during our discussion of the matched queries on BusyBox, the deficits relating to the *free-field-no-reassign* and *constant-array-access-no-check* queries (cf.

[6]https://github.com/joernio/joern/pull/4996.

Figure 6.2) are caused by a bug in Joern. This bug prevents the family-based approach from identifying any matches for the two queries. We found the same bug to also affect certain manifestations of the *unchecked-read-recv-malloc* query. However, in this case, it does not prevent but rather introduces artificial matches. The cause for this lies in the fact that the query demands a check of the return value of certain functions within the caller. While it allows this check to take place on a variable assigned the return value of the called function, the bug prevents this check to be correctly associated with the variable. As a result, even instances where an obvious check is present are flagged with a warning.

Line Mapping Deficiencies

We observed that many manifestations of the *signed-left-shift* query are attributable to the contents of global struct or enum definitions. For these definitions, SugarC often does not emit comments indicating the original line numbers into the product simulator. Since we prune warnings without a corresponding line mapping, this leads to a significant reduction in the number of vulnerability warnings. A similar problem can be observed for occurrences of the *copy-loop* query. Although loops and their enclosed statements typically receive a line mapping within the product simulator, we noticed that this mapping can become very imprecise. Depending on the complexity of the loop, the mapping may even refer to a completely different code region. If this region does not contain a structure typical of the matched query, an additional sanity check added to Sugarlyzer fails and the corresponding warning is removed from the report.

Insufficient Variability Encoding

A particularly severe issue affecting all queries is the fact that on BusyBox, SugarC often does not correctly variability encode crucial parts of individual source files. Similar to the missed *strlen-truncation* manifestation on axTLS' `ssl/loader.c` mentioned earlier, this leads to whole code regions not being correctly represented in the product simulator. As a result, potential vulnerabilities in these code regions cannot be detected by Joern and are thus not reported by the approach. We believe that this is the main reason for the large discrepancies between the results of the family-based approach and the baseline. It also explains the large spread in warnings per file shown by Figure 6.4b. While the family-based approach frequently matches the number of warnings of the baseline for files with two and fewer warnings, the files with more warnings show far more unreliable results.

6.3.1.5 Conclusion

Taking into account the aspects discussed above, we draw the following conclusion with regard to ***RQ-1***:

RQ-1: *How effective is our family-based analysis in finding potential occurrences of common vulnerability types across an SPL?*

Our family-based analysis demonstrates promising effectiveness in identifying potential vulnerabilities across an SPL. On axTLS*, it matches or exceeds the baseline on all but a single source file, while also detecting a match for an additional query. Although limitations of the utilized Q-SAST and variability encoding tooling impair effectiveness on* BusyBox*, our approach is still able to match or exceed the baseline on the majority of source files.*

6.3.2 Efficiency (RQ-2)

6.3.2.1 Execution Time

Across all three subject systems, the total execution time of the family-base approach remained reasonable, consistently staying below three hours (cf. Table 6.2). On the smallest subject system, axTLS, we repeatedly observed execution times below the 10-minute mark. In the case of smaller subject systems, the approach could therefore be applied frequently during the system's development and maintenance lifecycle. For instance, depending on the frequency with which changes are committed to a system's repository, the analysis could be run after every commit. For larger systems like Toybox and BusyBox, where the total execution time always exceeded one hour, performing a full analysis in such a fashion can quickly become infeasible. Instead, to save on resources, a full analysis of a system could be limited to a mandatory check executed before changes to the code base are incorporated into the main code line. If this check raises any vulnerability warnings that need to be addressed, the caching functionality of our approach would enable quick feedback for the modified source files.

6.3.2.2 Storage Demand

The total storage demand of our analysis ranged from an additional 105 MB on axTLS to an additional 3,494 MB on BusyBox (cf. Table 6.2). While it therefore vastly exceeded the sizes of the individual subject systems,[7] it always remained on a reasonable level. As expected, the variability-encoded source files used for the analysis exhibited a significantly larger size than their unpreprocessed counterparts across all subject systems. In this regard, most variability-encoded source files remained below 10 MB, with only five files from BusyBox exhibiting a larger size. Notably, the product simulator created by SugarC for BusyBox's `archival/libarchive/init_handle.c`, a source file containing a mere 17 line of C code, exceeded a staggering 470 MB in size. Considering the output produced by Joern, we observed that all CPG binaries remained below 6 MB, while all JSON report files consistently remained below 100 KB.

6.3.2.3 Conclusion

Taking into account the performance of our family-based approach with regard to execution time and storage demand, we draw the following conclusion for ***RQ-2***:

RQ-2:	*How efficient is our family-based analysis with regard to execution time and storage demand?*
Our family-based analysis achieves a high level of efficiency, making it fit for practical use. Across all considered subject systems, it maintained a very reasonable execution time, always staying below 3 hours for a complete analysis. Furthermore, it exhibits a practical storage demand, requiring less than 3.5 GB on the largest subject system.	

[7]All subject systems do not exceed a total size of 20 MB.

6.4 Threats to Validity

In this section, we discuss aspects threatening the validity of our approach and the associated evaluation. In general, threats can relate to four major classes of validity: conclusion, internal, construct, and external validity [Woh+24]. In the following, we focus on threats relating to internal (Section 6.4.1), construct (Section 6.4.2), and external validity (Section 6.4.3). We omit a separate discussion of the threats to the conclusion validity as this validity class is predominantly concerned with statistical relationships [Woh+24]. However, for our evaluation, we did not conduct any statistical tests establishing such relationships. Furthermore, in applied research, such as ours, conclusion validity usually holds the least significance among the different validity types [Woh+24].

6.4.1 Internal Validity

In essence, internal validity is concerned with two aspects: "the validity of the given environment and the reliability of the results" [Woh+24]. As Vari-Joern is meant to be executed in a corresponding Docker container, there are no environmental changes that may threaten internal validity. However, there are a number of factors that may threaten the reliability of our results. These threats are described below:

Variability Encoding

As indicated by our results (cf. Table 6.2), not all source files found within the considered subject systems could be variability encoded. Consequently, as stated in Section 6.2.2, this led us to consider only warnings on files for which a product simulator could be created. While this ensured a fair assessment of our approach's vulnerability discovery capabilities on the remaining source files, it led to a reduction in the overall warnings considered for the baseline. As a result, our results relating to the approaches' effectiveness may be biased towards the family-based approach. However, we argue that the extent of this bias should be limited. Across all three selected subject systems, SugarC was always capable of variability encoding at least 80% of all source files. The number of files excluded from consideration is therefore limited. In addition, while excluding certain source files from the report of the product-based approach decreased the total number of warnings on BusyBox, the number of warnings on axTLS remained unchanged (cf. Table 6.3). Moreover, the number of matched queries remained unaffected for both BusyBox and axTLS. Lastly, from a practical viewpoint, many of the source files for which variability encoding failed are not part of the actual source code of the subject systems. Rather, they represent auxiliary files associated with aspects like the systems' configuration or build process. Thus, the vulnerability warnings removed from consideration arguably only have a limited relevance for a system's overall security.

Line Mappings

In the context of the Sugarlyzer framework, the correct operation of the warning mapping component heavily relies on the source line mappings incorporated into a product simulator by SugarC. As we have outlined in Section 5.4, these line mappings are not always reliable. While we tried to mitigate this problem by searching for a plausible line mapping in the neighborhood of a warning, this strategy is not perfect. Thus, the analysis may still yield warnings without a mapping to a location in the SPL's code. In this regard, we prune all warnings without a line mapping. Similarly, we employ a sanity check that filters out warnings referring to code regions within the unpreprocessed source code that do not contain structures typical of the matched query. Evidently, these steps introduce a risk of discarding valid vulnerability warnings. However, they also improve the overall meaningfulness of the report created by our approach. Retaining warnings with questionable line mappings would otherwise artificially inflate the report with warnings that are either invalid or originate in another source file, effectively constituting duplicates. We thus argue that the benefit of pruning warnings with an invalid line mapping outweighs the potential loss of valid vulnerability warnings.

Joern

A central piece of our analysis approach, as well as the product-based baseline used for the evaluation, is the Q-SAST tool JOERN. While JOERN represents an established tool that has seen more than ten years of continuous development [Yam+14], it evidently still contains a number of serious bugs. As described in Section 6.3.1, we identified a bug that caused certain queries to produce none or false warnings on a product simulator. Additionally, on certain products of AXTLS, another bug caused the baseline approach to miss an obvious match to a simple query. It thus remains questionable whether the vulnerability warnings collected for the two considered analysis approaches represent the true set of vulnerability patterns identifiable in the corresponding CPGs. To this end, a bug triggered by only one of the two approaches could have unknowingly introduced a strong bias into our results. Mitigating this problem would have required adding support for another Q-SAST tool to our family-based analysis and expanding the associated evaluation. However, this exceeded the scope of this thesis.

6.4.2 Construct Validity

Besides the threats to the internal validity outlined above, there are also factors that threaten the fundamental connection between our collected metrics and the overarching goals of effectiveness and efficiency. These factors represent threats to the construct validity [Woh+24] and are detailed below:

Consideration of Aggregated Alarms

For assessing our approach's effectiveness, we considered the number of vulnerability warnings. To this end, we not only analyzed the total numbers of warnings but also the numbers of warnings aggregated by query and by source file. Using these aggregated numbers, for AXTLS, we analyzed individual cases where we observed deviations between our family-based and the baseline strategy. For BUSYBOX, we limited the analysis to identifying general patterns causing deviations due to the large number of source files and warnings. Given that we thus not analyzed every vulnerability warning in isolation, there is a risk that individual warnings for particular source files or queries might not relate to the same problem between our and the baseline strategy. For instance, even though both strategies might report similar numbers for a particular source file or query, internally, the warnings might relate to completely different issues. While the granularity of the comparison could be refined to account for this problem, this would have increased the extent of the evaluation to a level beyond the scope of this thesis.

Baseline Approach

As our baseline, we employed results produced by VARI-JOERN's product-based approach using the uniform sampling strategy. We chose a product-based strategy using sampling as it represents a popular method for the analysis of SPLs in practice [Lie+13]. Furthermore, the chosen uniform sampling strategy is simple and can freely be scaled to arbitrary sample sizes, making it an ideal baseline. However, in practice, there exist far more complex sampling strategies that have the potential to outperform a uniform strategy in both effectiveness and efficiency. Therefore, while our approach demonstrated promising results compared to the uniform sampling strategy, this might not reflect its capabilities in comparison to state-of-the-art sampling strategies. To provide a clear understanding of our approach's true effectiveness and efficiency, a comparison with other sampling algorithms would thus be necessary. However, this would have exceeded the scope of this thesis.

6.4.3 External Validity

Lastly, external validity is concerned with the generalizability of the findings to other settings [Woh+24]. In this regard, we have identified the following threats:

Limited Number of Subject Systems

An obvious threat to the external validity of our approach lies in the fact that our evaluation focused on a small set of only three subject systems. Furthermore, with both TOYBOX and BUSYBOX representing utility collections, the diversity exhibited by our selection is limited. As a result, it remains questionable whether our findings generalize to other systems. We tried to address this issue by selecting representative systems used frequently in the field of SPL research. Furthermore, to avoid overfitting towards a specific system size, we selected three systems of different sizes, ranging from a moderate size of around 17,000 lines of code to over 180,000 (cf. Table 6.1).

Limited Number of Utilized Queries

For our evaluation, we relied on the default query set for C contained in the JOERN query database. While this set models a number of common vulnerability patterns, its overall size of 16 queries is relatively small. In comparison, for the popular Q-SAST tool CODEQL, there are more than 500 queries tailored to C/C++ [Li+24]. As a result, generalizing the effectiveness of our approach with regard to its vulnerability discovery capabilities to other vulnerability patterns might be difficult. This problem is further exacerbated by the fact that the complexity of most queries in the JOERN query database is limited. Therefore, our results may also be subject to simplicity bias [Aba+17], potentially deviating for more complex vulnerability patterns. Addressing these problems would require two steps. First, a larger number of subject systems would need to be considered. Given that the available data of the product-based approach serving as our baseline only considered AXTLS and BUSYBOX, for the scope of this thesis, this was not possible. Second, a larger set of queries would need to be considered. While we could have expanded to the whole 27 queries of the JOERN query database dedicated to C, the baseline data was collected using the default set. Therefore, for the additional queries, we would have missed baseline data, making an assessment of our approach's effectiveness on these queries difficult.

7. Related Work

In Chapter 4, we have described the core contribution of this thesis: a family-based analysis approach aimed at the identification of VIVs in real-world SPLs. The structure of this approach is shown again in Figure 7.1, highlighting three aspects covered by existing solutions or prior research. In this chapter, we aim to explore the related work on these aspects in more detail. From a technical perspective, our approach builds on advancements in the fields of variability encoding and Q-SAST (cf. ① and ② in Figure 7.1). In Sections 7.1 and 7.2, we therefore give an overview of prominent solutions in these two fields. Furthermore, while our approach is the first to leverage the benefits of Q-SAST for the analysis of SPLs, previous research has explored the broader goal of applying off-the-shelf SAST tools to SPLs (indicated by ③ in Figure 7.1). Therefore, we additionally provide a summary of key publications that relate to this goal (Section 7.3).

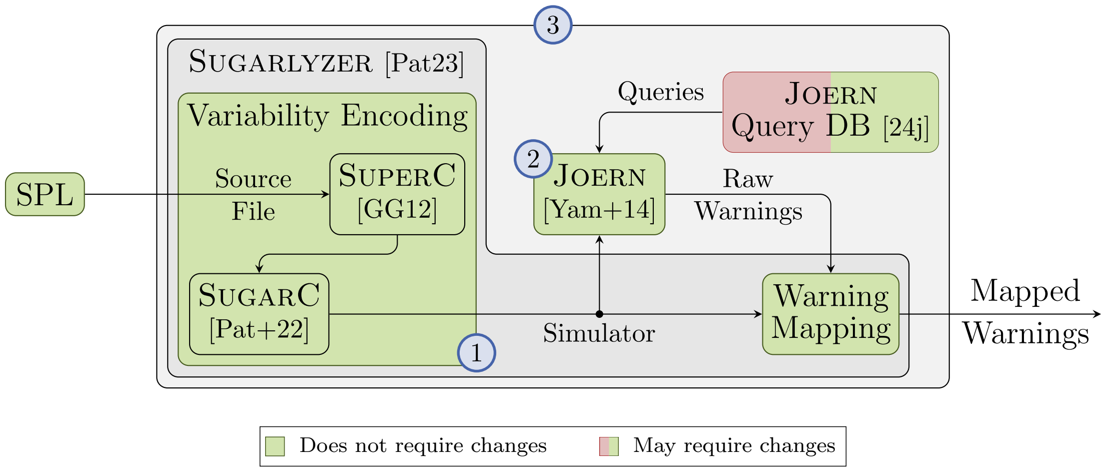


Figure 7.1: Aspects of our approach covered by previous research or existing solutions

7.1 Variability Encoding

As we detailed in Section 3.3, transforming unpreprocessed source code into a corresponding product simulator usually encompasses two tasks (cf. Figure 7.1).

First, to recognize its syntactical structure, the source code has to be parsed. We give an overview of publications relating to this step in Section 7.1.1. Either during or after parsing, behavior-preserving transformations can then be applied to rewrite compile-time variability into run-time variability, ultimately resulting in a product simulator. Publications relating to this step are described in Section 7.1.2.

7.1.1 Variability-Aware Parsing

Parsing unpreprocessed C represents a challenging task [Ken+10], even believed impossible by some without the use of heuristics [Käs+11; Pad09; SL98]. This has sparked interest in the research community, leading to the development of two parsers that form the state of the art for parsing unpreprocessed C of real-world SPLs.

TypeChef

First, Kenner et al. [Ken+10] introduced TypeChef. As the name of their tool suggests, TypeChef not only aims at parsing unpreprocessed C but also performs type checking to detect type errors within an SPL. For parsing, TypeChef first employs a configuration-preserving preprocessor for resolving file inclusions and macro substitutions [KGO11]. It then uses a parser generated from a custom C grammar extended with dedicated productions for preprocessor conditionals to build a variability-aware AST [Ken+10]. This AST expresses variability by annotating edges with their corresponding presence condition [GS20] and can be used to type check all products of an SPL at once [Ken+10]. While TypeChef was continuously improved from its initial release (e.g., to add support for most cases of undisciplined preprocessor annotations [Käs+11]), development on the TypeChef project[1] has stopped in late 2021.

SuperC

To address limitations exhibited by TypeChef, Gazzillo et al. [GG12] introduced SuperC. Similar to TypeChef, their tool follows a two-phase approach. First, SuperC resolves file inclusions and macro substitutions through the use of a configuration-preserving preprocessor [GG12]. It then employs a fork-merge parser to build a variability-aware AST [GG12]. This AST expresses variability through dedicated choice nodes [GG12; GS20]. Although SuperC was introduced more than a decade ago, its associated project[2] still receives occasional updates. It is also employed by the Sugarlyzer framework [Pat23] used as the foundation of our approach. SuperC therefore represents the parsing solution utilized by our approach (cf. Figure 7.1).

7.1.2 Behavior-Preserving Transformation

While it is always possible to transform variable source code into a corresponding product simulator, minimizing code duplication and maintaining behavior preservation during this process represent challenging tasks [von16]. This is exacerbated for source code written in C, where a significant portion of preprocessor annotations can be undisciplined [LKA11]. To this end, a number of studies proposed methods for transforming parsed unpreprocessed C code into a product simulator.

[1]https://github.com/ckaestne/TypeChef.
[2]https://github.com/appleseedlab/superc/.

Hercules

Building on the initial considerations of von Rhein et al. [von+16], von Rhein [von16] and Garbe [Gar17] proposed HERCULES. Their tool takes a variability-aware AST produced by TYPECHEF [Ken+10] (cf. Section 7.1.1) as input and aims at enabling an efficient performance measurement of the represented SPLs [Gar17]. In pursuit of this goal, the tool applies tree transformations to the variability-aware AST, turning it into a corresponding product simulator [Gar17]. To enable performance measurements for individual configurations, HERCULES additionally injects the product simulator with artificial performance measuring functions [Gar17]. Building on TYPECHEF, HERCULES is only able to variability-encode source code that previously passed TYPECHEF's type checking [Gar17; Pat+22]. Furthermore, the transformations applied are only informally described and do not cover all variability patterns found inside real-world C [Pat23].

C Reconfigurator

Following a more formal approach, Iosif-Lazar et al. [Ios+17] presented a number of general program transformations, whose correctness was demonstrated for the small imperative language IMP. Applying these transformations to C, Iosif-Lazar et al. introduced the tool C RECONFIGURATOR [Ios+17]. Their tool uses SUPERC [GG12] (cf. Section 7.1.1) for the creation of a variability-aware AST [Ios+17]. On such an AST, the tool then applies the presented transformations as tree transformations realized in Xtend[3] [Ios+17]. The resulting ordinary AST is subsequently translated into C, representing the corresponding product simulator [Ios+17]. Since C RECONFIGURATOR's transformations are based on IMP, which only represents a subset of C, it does not support many constructs found in real-world C programs [Ios+17; Pat+22]. Moreover, C RECONFIGURATOR cannot handle type errors found within a variability-aware AST and relies on SUPERC for detecting syntax errors [Pat+22].

SugarC

Addressing some of the limitations exhibited by HERCULES [Gar17; von16] and C RECONFIGURATOR [Ios+17], Patterson et al. [Pat+22] proposed SUGARC. It not only represents the latest solution in the field of behavior-preserving variability transformations for C but is also the tooling employed by our approach. To this end, SUGARC serves as a central component of the SUGARLYZER framework [Pat23], which forms the foundation of our approach (cf. Figure 7.1). Instead of operating on a fully-built variability-aware AST, SUGARC is realized as semantic actions that transform source code constructs directly after they have been parsed [Pat+22]. Evidently, this approach requires tight coupling between the utilized parser and SUGARC. To this end, SUGARC has been integrated into the variability-aware parser SUPERC (cf. Section 7.1.1). SUGARC is capable of handling syntax and type errors and preserves them in the product simulator as run-time errors [Pat+22]. Additionally, special comments are incorporated into the simulator to enable lines to be traced back to their location in the unpreprocessed source code [Pat+22]. Although the applied transformations are not complete and their correctness has not been formally proven, SUGARC claims to support many complex cases found inside real-world C [Pat+22]. Additionally, it has been shown to outperform HERCULES and C RECONFIGURATOR [Pat+22], establishing it as the current state of the art.

[3]https://eclipse.dev/Xtext/xtend/.

7.2 Query-Based Static Application Security Testing

With our approach, we aim to leverage the benefits of Q-SAST for the analysis of SPLs. To this end, we employ the analysis tool Joern (cf. Figure 7.1), originally introduced by Yamaguchi et al. [Yam+14]. We chose Joern as it leverages the expressive source code representation of CPGs, which allows even complicated taint-style vulnerabilities to be modeled. Furthermore, with its implementation being open source, it allowed us to consider both lifting by extension and lifting by simulation for our approach. Even though Q-SAST tools provide numerous distinct benefits compared to NQ-SAST tools [Li+24], they remain less common. Besides Joern, there are three Q-SAST tools that enjoy widespread use for the analysis of C in practice [Li+24].

SonarQube

A commercial Q-SAST tool that enjoys widespread use in practice is SonarQube.[4] Contrary to Joern, not all parts of SonarQube's implementation are open source [Li+24]. Additionally, compared to the small database of existing queries for Joern, it boats a large collection of more than 850 queries (referred to as *rules*) [Li+24]. These queries are designed to identify various issues in C/C++ code, ranging from bugs and code smells to security vulnerabilities [Li+24]. As characteristic of a Q-SAST tool, users can incorporate their own queries by deriving custom rules from existing templates [Li+24; Shi+22].

CodeQL

Another widely used Q-SAST tool is CodeQL.[5] Similar to SonarQube, the implementation of CodeQL is not entirely open-source [Li+24]. For its analysis, the tool is not limited to syntactical considerations. Besides using a program's AST, CodeQL also considers the associated CFG and the data flow graph [Li+24]. For C/C++ alone, there are more than 550 existing queries leveraging access to these data structures [Li+24], vastly exceeding the 27 queries available for C inside the Joern query database. These queries are written in a dedicated query language based on Datalog [Li+24].

Semgrep

Contrary to SonarQube and CodeQL, Semgrep[6] represents an open-source Q-SAST tool [Li+24]. It offers more than 80 existing queries for C/C++, specified in a syntax resembling regular expressions [Li+24]. While these queries are targeted towards different types of issues, their expressiveness is generally limited. From a technical standpoint, Semgrep converts queries into AST patterns that are subsequently searched within the analyzed program's AST [Li+24]. Compared to other Q-SAST tools like Joern, which leverage additional source code representations besides a program's AST, the tool therefore has limited capabilities.

[4]http://sonarqube.org/.
[5]https://codeql.github.com/.
[6]https://semgrep.dev/.

7.3 Analysis of SPLs with SAST Tools

The analysis of SPLs represents a challenging task that has been the subject of extensive research [Mei+14; Thü+14]. While engineering novel analysis approaches constitutes a valid option, this task demands considerable engineering effort [Ios+17; Pat23]. Therefore, many studies instead aim to adapt and reuse methods known from the analysis of non-variable software. For our approach, we followed this idea and aimed to employ an off-the-shelf Q-SAST tool for the analysis of SPLs for vulnerabilities. To the best of our knowledge, there is no existing solution that relates to our approach directly and applies a conventional Q-SAST tool to SPLs. However, considering the broader field of SAST tools, a small set of studies aimed to apply NQ-SAST tools to SPLs. These are the studies that are most closely related to ours.

Mordahl et al. [Mor+19] allowed Q-SAST tools to be applied to SPLs by following an optimized product-based approach [Thü+14]. To this end, they sampled products from the three SPLs of AXTLS, TOYBOX, and BUSYBOX using Oh et al.'s [Oh+19] uniform random sampling tool SMARCH. The sampled products were then analyzed using the four popular SAST tools CBMC, CLANG STATIC ANALYZER, CPPCHECK, and INFER. Lastly, post-processing was applied to the raised warnings to perform deduplication and to deduce the causing features. While they followed a goal similar to ours, their approach adopted a product-based strategy. This strategy is easy and therefore popular in practice [Lie+13; Thü+14]. However, it incurs redundant computations for all parts shared between the sampled products (e.g., the common core) [Sah+16; Thü+14]. Our approach avoids this problem by pursuing a family-based strategy, analyzing an SPL directly instead of individual products [Ape+13b; DBW19; Ios+17].

Schubert et al. [Sch+22] presented a family-based approach that leveraged variability encoding for applying the data-flow analysis framework PHASAR [SHB19] to SPLs. Their implementation, VARALYZER, first variability-encoded unpreprocessed C with a precursor to SUGARC [Pat+22] (cf. Section 7.1.2). It then applied PHASAR [SHB19] to the resulting product simulator for solving the data-flow problem specified by the user. Since PHASAR [SHB19] can solve data-flow analyses specified in the IFDS [RHS95] and IDE [SRH96] frameworks, this allowed Schubert et al. [Sch+22] to apply conventional data-flow analyses to SPLs. Overall, their approach follows the same idea of applying a variability-oblivious analysis tool to a product simulator as ours. However, in contrast to our approach, it focuses on PHASAR [SHB19] and thus data-flow analyses. While certain vulnerabilities may be detected by considering a program's data flow in isolation, complex vulnerabilities can require a consideration of additional aspects, such as a program's structure of data flow. To this end, besides a program's data flow, our approach, may also consider a program's syntactical structure as well as its control flow by leveraging CPGs.

Lastly, the study that comes closest to ours is that of Patterson [Pat23]. In his study, Patterson proposed the SUGARLYZER framework with the aim of enabling scalable and precise variability bug detection for SPLs through the use of conventional SAST tools. In essence, SUGARLYZER uses SUPERC [GG12] (cf. Section 7.1.1) and SUGARC [Pat+22] (cf. Section 7.1.2) for variability encoding and allows any SAST tool to be applied to the resulting product simulator through appropriate interfaces. Similar to the solution of Mordahl et al. [Mor+19], post-processing is applied to the raised warnings. While our approach is built around the SUGARLYZER framework (cf. Figure 7.1), we focus on the identification of VIVs rather than general variability bugs. Since Q-SAST tools provide expressive means to model VIVs, we subsequently added support for the off-the-shelf Q-SAST tool JOERN [Yam+14] to the framework. Furthermore, we incorporated SUGARLYZER into the existing analysis platform VARI-JOERN instead of treating it as its own standalone solution.

8. Conclusion and Outlook

Since the exploitation of even a single vulnerability can lead to disastrous consequences, the identification and subsequent removal of corresponding patterns in a software system represents a crucial task. For SPLs, which give rise to a plethora of different software products, this task substantially increases in complexity. This complexity arises from the presence of VIVs, vulnerabilities that are only present in specific software products of an SPL. The inherent challenge of using conventional (i.e., variability-oblivious) analysis tooling for the discovery of problematic patterns within an SPL has been the subject of earlier research [Mor+19; Pat23; Sch+22]. However, there has been no approach that follows a family-based strategy while also leveraging the unique advantages of Q-SAST.

Responding to this gap, in this thesis, we proposed a family-based analysis approach for SPLs leveraging the benefits of Q-SAST for the identification of VIVs. In this regard, we first examined two strategies through which an off-the-shelf Q-SAST tool can be lifted to the domain of SPLs and thus used for a family-based analysis. The strategy of *lifting by extension* aims at extending the internals of a tool to accommodate variability, thereby enabling the tool to operate on an SPL directly. *Lifting by simulation*, on the other hand, aims at transforming the variability of an SPL into a form on which an analysis tool can operate without the need for modification. Comparing the two aforementioned strategies, we found lifting by extension to promise greater precision and performance, while lifting by simulation promises benefits with regard to maintainability, expansion, and implementation effort.

Based on the comparison of the two lifting strategies, we decided to implement lifting by simulation. To limit the required implementation effort, the design of our approach revolved around reusing existing solutions whenever possible. Accordingly, we based our approach on the SUGARLYZER framework of Patterson [Pat23]. While this framework provided us with implementations for all components required by lifting by simulation, it did not support our chosen Q-SAST tool, JOERN, out of the box. Accordingly, our implementation focused on adding support for JOERN [24h] to SUGARLYZER. Moreover, we addressed a number of limitations of the framework exposed by this process and integrated the resulting implementation into the existing analysis platform Vari-Joern.

For the evaluation of the proposed approach, we considered the effectiveness and efficiency of its implementation on the three subject systems of axTLS [24a], Toybox [24p], and BusyBox [24b]. As a baseline for this evaluation, we used results for axTLS and BusyBox produced by the product-based approach available in Vari-Joern. Overall, our findings confirm the effectiveness of the approach in identifying potential vulnerabilities. For axTLS, it matched or exceeded the baseline on all but a single source file. Furthermore, while limitations of the tools employed by our approach severely impaired effectiveness on BusyBox, our approach was still able to match or exceed the baseline on the majority of source files. Considering the efficiency of the approach, our findings confirm a very reasonable resource demand. Even for the largest subject system of BusyBox, execution time consistently stayed below 3 hours for a full analysis. Furthermore, the maximum storage demand of our approach remained modest, not surpassing 3.5 GB. Taking into account the results of our evaluation, we argue that the proposed approach not only demonstrates promising effectiveness but also efficiency fit for practical use. Accordingly, we formulate the following conclusion for the ***MRQ*** (cf. Section 1.1):

MRQ: *How can a Q-SAST tool be lifted to the domain of SPLs, i.e., be employed for a family-based analysis that is both effective and efficient?*

Conclusion*. A Q-SAST tool can be lifted to the domain of SPLs by extending its internals (lifting by extension) or by rewriting the variability of the analyzed SPL (lifting by simulation). While lifting by extension promises greater precision and performance, lifting by simulation offers distinct benefits with regard to maintainability, extensibility, and implementation effort. Although lifting by simulation is therefore at a disadvantage in terms of precision and performance, the proposed analysis approach demonstrates that it can still achieve promising effectiveness and efficiency fit for practical use. As a result, lifting by simulation represents a viable solution for lifting an off-the-shelf Q-SAST tool to the domain of SPLs.*

Future Work

This thesis laid the foundation for lifting off-the-shelf Q-SAST tools to the domain of SPLs by highlighting two common lifting strategies and demonstrating the practical viability of the strategy of lifting by simulation. While this yielded valuable insights into the vulnerability discovery possibilities for SPLs, there are still a number of areas that could benefit from improvements and continued research. These areas are outlined below:

- **Enabling Custom Queries**: While the analysis approach proposed in this thesis aimed to leverage the advantages of Q-SAST, to provide a transparent evaluation of its effectiveness and efficiency, we used Joern in NQ-SAST mode (i.e., with a fixed set of queries). To benefit from all the advantages of Q-SAST, it is however necessary to allow users to control the analysis by providing their own queries. Only then can queries modeling newly discovered vulnerabilities be incorporated or existing ones adjusted to reflect the specifics of a particular system. Extending the proposed approach (and hence the

Sugarlyzer framework on which it is built) with this functionality would therefore constitute a valuable improvement.

- **Improving Variability Encoding Solutions**: The evaluation of the proposed analysis approach revealed that the variability encoding tooling used by the Sugarlyzer framework (i.e., SuperC and SugarC) still exhibits major limitations. On certain source files and code regions, the tooling outright fails to produce a corresponding product simulator. Additionally, the line mapping information inserted into a product simulator can be imprecise or even absent. As the proposed approach hinges on the presence of a product simulator containing precise line mapping information, addressing these limitations would significantly improve its overall effectiveness.
- **Extending the Evaluation**: For the evaluation of the proposed analysis approach, we focused on three subject systems. As a baseline, we used results produced by Vari-Joern's product-based analysis strategy using the uniform random sampling strategy (cf. Section 6.1). Furthermore, the approach currently relies on the 16 queries for C with the default tag of the Joern query database (cf. Section 4.4.1). While this setup allowed us to gain first insights into the effectiveness and efficiency of the proposed approach, its limited extent makes a generalization of the findings difficult. Extending the evaluation to include additional systems and queries, as well as using multiple advanced sampling strategies as baselines, could provide valuable insights into the generalizability of our findings.
- **Selective Variability-Awareness**: As indicated by our evaluation (cf. Section 6.3.2), the analysis approach proposed in this thesis exhibits high efficiency for real-world SPLs of small to large size. However, especially in view of the analysis of very large SPLs, this aspect could be further improved. In this regard, selective variability awareness, as explored by Dimovski et al. [DBW19], would be an interesting area for future research. Depending on the degree of variability exhibited by a source file or a code region, variability encoding could be omitted and the corresponding source code analyzed as if it belonged to a non-configurable system. This would reduce the overhead for sections of an SPL that do not benefit from a variability-aware analysis (e.g., the common core).
- **Implementing Lifting by Extension**: A key finding of this thesis is that lifting by simulation represents a viable strategy for lifting an off-the-shelf Q-SAST tool to the domain of SPLs. However, from a theoretical perspective, this strategy is at a disadvantage in terms of precision and performance when compared to lifting by extension (cf. Section 3.4). Applying the strategy of lifting by extension therefore represents an intriguing avenue for future research. A corresponding implementation using Joern could then be compared with the analysis approach proposed in this thesis. The extent to which lifting by extension enables better precision and performance for the analysis of SPLs could then be judged.

Bibliography

[24a] *axTLS Embedded SSL*. Website. May 2024. URL: https://axtls.sourceforge.net/ (visited on 05/09/2024).

[24b] *BusyBox*. Website. May 2024. URL: https://busybox.net/ (visited on 05/09/2024).

[24c] *CBMC: Bounded Model Checking for Software*. Website. May 2024. URL: https://www.cprover.org/cbmc/ (visited on 05/09/2024).

[24d] *Clang Static Analyzer*. Website. May 2024. URL: https://clang-analyzer.llvm.org/ (visited on 05/09/2024).

[24e] *CodeQL*. Website. May 2024. URL: https://codeql.github.com/ (visited on 05/09/2024).

[24f] *GitHub - Joernio/Joern: Open-source Code Analysis Platform for C / C++ / Java / Binary / Javascript / Python / Kotlin Based on Code Property Graphs*. Source Code Repository. 2024. URL: https://github.com/joernio/joern (visited on 09/13/2024).

[24g] *Infer Static Analyzer*. Website. May 2024. URL: https://fbinfer.com/ (visited on 05/09/2024).

[24h] *Joern - The Bug Hunter's Workbench*. Website. June 2024. URL: https://joern.io/ (visited on 06/02/2024).

[24i] *Joern Documentation*. Documentation. 2024. URL: https://docs.joern.io/ (visited on 07/16/2024).

[24j] *Joern Query Database*. Website. 2024. URL: https://queries.joern.io/ (visited on 09/12/2024).

[24k] *NIST Software Assurance Reference Dataset*. Website. Nov. 2024. URL: https://samate.nist.gov/SARD/test-suites/ (visited on 11/26/2024).

[24l] *PhASAR*. Website. June 2024. URL: https://phasar.org/ (visited on 06/02/2024).

[24m] *SonarQube*. Website. Nov. 2024. URL: https://www.sonarsource.com/products/sonarqube/ (visited on 11/22/2024).

[24n] *The Heartbleed Bug*. July 2024. URL: https://heartbleed.com/ (visited on 07/01/2024).

[24o] *The Linux Kernel Archives*. Website. May 2024. URL: https://www.kernel.org/ (visited on 05/09/2024).

[24p] *Toybox*. Website. May 2024. URL: https://www.landley.net/toybox/ (visited on 05/09/2024).

[Aba+17] Iago Abal et al. “Variability Bugs in Highly Configurable Systems: A Qualitative Analysis”. In: *ACM Transactions on Software Engineering and Methodology* 26.3 (July 2017), pp. 1–34. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3149119. URL: https://dl.acm.org/doi/10.1145/3149119 (visited on 05/15/2024).

[ABW14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. “42 Variability Bugs in the Linux Kernel: A Qualitative Analysis”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Vasteras Sweden: ACM, Sept. 2014, pp. 421–432. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642990. URL: https://dl.acm.org/doi/10.1145/2642937.2642990 (visited on 05/20/2024).

[ACB16] Marcelo Arroyo, Francisco Chiotta, and Francisco Bavera. “An User Configurable Clang Static Analyzer Taint Checker”. In: *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. Valparaíso, Chile: IEEE, Oct. 2016, pp. 1–12. ISBN: 978-1-5090-3339-3. DOI: 10.1109/SCCC.2016.7835996. URL: http://ieeexplore.ieee.org/document/7835996/ (visited on 05/08/2024).

[Aho+07] Alfred V. Aho et al. *Compilers: Principles, Techniques, & Tools*. 2nd Edition. Boston: Pearson/Addison Wesley, 2007. ISBN: 978-0-321-48681-3.

[AK09] Sven Apel and Christian Kästner. “An Overview of Feature-Oriented Software Development.” In: *The Journal of Object Technology* 8.5 (2009), p. 49. ISSN: 1660-1769. DOI: 10.5381/jot.2009.8.5.c5. URL: http://www.jot.fm/contents/issue_2009_07/column5.html (visited on 04/22/2024).

[Ang18] Renzo Angles. “The Property Graph Database Model”. In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*. Ed. by Dan Olteanu and Barbara Poblete. Vol. 2100. CEUR Workshop Proceedings. Cali, Colombia: CEUR-WS.org, May 2018. URL: https://ceur-ws.org/Vol-2100/paper26.pdf.

[Ape+11] Sven Apel et al. “Detection of Feature Interactions Using Feature-Aware Verification”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence, KS, USA: IEEE, Nov. 2011, pp. 372–375. ISBN: 978-1-4577-1639-3. DOI: 10.1109/ASE.2011.6100075. URL: http://ieeexplore.ieee.org/document/6100075/ (visited on 05/05/2024).

[Ape+13a] Sven Apel et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7. URL: https://link.springer.com/10.1007/978-3-642-37521-7 (visited on 04/17/2024).

[Ape+13b] Sven Apel et al. “Strategies for Product-Line Verification: Case Studies and Experiments”. In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 482–491. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606594. URL: http://ieeexplore.ieee.org/document/6606594/ (visited on 04/25/2024).

[BD23] Van-Cong Bui and Xuan-Cho Do. “Detecting Software Vulnerabilities Based on Source Code Analysis Using GCN Transformer”. In: *2023 RIVF International Conference on Computing and Communication Technologies (RIVF)*. Hanoi, Vietnam: IEEE, Dec. 2023, pp. 112–117. ISBN: 979-8-3503-1584-4. DOI: 10.1109/RIVF60135.2023.10471834. URL: https://ieeexplore.ieee.org/document/10471834/ (visited on 04/29/2024).

[Bod+13] Eric Bodden et al. “SPL $^{\text{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Seattle Washington USA: ACM, June 2013, pp. 355–364. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2491976. URL: https://dl.acm.org/doi/10.1145/2491956.2491976 (visited on 04/19/2024).

[Bra+12] Claus Brabrand et al. “Intraprocedural Dataflow Analysis for Software Product Lines”. In: *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*. AOSD ’12. New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 13–24. ISBN: 978-1-4503-1092-5. DOI: 10.1145/2162049.2162052. URL: https://dl.acm.org/doi/10.1145/2162049.2162052 (visited on 04/15/2024).

[Bra+13] Claus Brabrand et al. “Intraprocedural Dataflow Analysis for Software Product Lines”. In: *Transactions on Aspect-Oriented Software Development X*. Ed. by David Hutchison et al. Vol. 7800. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 73–108. ISBN: 978-3-642-36963-6 978-3-642-36964-3. DOI: 10.1007/978-3-642-36964-3_3. URL: http://link.springer.com/10.1007/978-3-642-36964-3_3 (visited on 04/22/2024).

[Bus+96] Frank Buschmann et al. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Vol. 1. Chichester, England: Wiley, Aug. 1996. ISBN: 978-0-471-95869-7.

[Cao+24] Sicong Cao et al. “EXVUL: Towards Effective and Explainable Vulnerability Detection for IoT Devices”. In: *IEEE Internet of Things Journal* (2024), pp. 1–14. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2024.3381641. URL: https://ieeexplore.ieee.org/document/10479158/ (visited on 04/16/2024).

[Cas+21] Thiago Castro et al. “A Formal Framework of Software Product Line Analyses”. In: *ACM Transactions on Software Engineering and Methodology* 30.3 (July 2021), pp. 1–37. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3442389. URL: https://dl.acm.org/doi/10.1145/3442389 (visited on 04/22/2024).

[Cla+10] Andreas Classen et al. “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*. Cape Town South Africa: ACM, May 2010, pp. 335–344. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806850. URL: https://dl.acm.org/doi/10.1145/1806799.1806850 (visited on 05/24/2024).

[CM04] B. Chess and G. McGraw. “Static Analysis for Security”. In: *IEEE Security and Privacy Magazine* 2.6 (Nov. 2004), pp. 76–79. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.111. URL: http://ieeexplore.ieee.org/document/1366126/ (visited on 04/23/2024).

[DBW19] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. “Finding Suitable Variability Abstractions for Lifted Analysis”. In: *Formal Aspects of Computing* 31.2 (Apr. 2019), pp. 231–259. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/s00165-019-00479-y. URL: https://dl.acm.org/doi/10.1007/s00165-019-00479-y (visited on 04/22/2024).

[Du+20] Xiang Du et al. “Vulnerability Analysis through Interface-based Checker Design”. In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Macau, China: IEEE, Dec. 2020, pp. 46–52. ISBN: 978-1-72818-915-4. DOI: 10.1109/QRS-C51114.2020.00019. URL: https://ieeexplore.ieee.org/document/9282714/ (visited on 04/16/2024).

[EL02] D. Evans and D. Larochelle. “Improving Security Using Extensible Lightweight Static Analysis”. In: *IEEE Software* 19.1 (Aug. 2002), pp. 42–51. ISSN: 07407459. DOI: 10.1109/52.976940. URL: http://ieeexplore.ieee.org/document/976940/ (visited on 04/28/2024).

[Fel+16] Michael Felderer et al. “Chapter One - Security Testing: A Survey”. In: *Advances in Computers*. Vol. 101. Elsevier, 2016, pp. 1–51. ISBN: 978-0-12-805158-0. DOI: 10.1016/bs.adcom.2015.11.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0065245815000649 (visited on 05/08/2024).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/24039.24041. URL: https://dl.acm.org/doi/10.1145/24039.24041 (visited on 06/05/2024).

[Gar17] Florian Garbe. “Performance Measurement of C Software Product Lines”. MA thesis. Passau: University of Passau, 2017. URL: https://www.se.cs.uni-saarland.de/theses/FlorianGarbeMA.pdf.

[Gaz17] Paul Gazzillo. “Kmax: Finding All Configurations of Kbuild Makefiles Statically”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 279–290. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106283. URL: https://dl.acm.org/doi/10.1145/3106237.3106283 (visited on 05/11/2024).

[GG12] Paul Gazzillo and Robert Grimm. “SuperC: Parsing All of C by Taming the Preprocessor”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing China: ACM, June 2012, pp. 323–334. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254103. URL: https://dl.acm.org/doi/10.1145/2254064.2254103 (visited on 04/19/2024).

[GJ03] A. Garrido and R. Johnson. "Refactoring C with Conditional Compilation". In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* Montreal, Que., Canada: IEEE Comput. Soc, 2003, pp. 323–326. ISBN: 978-0-7695-2035-3. DOI: 10.1109/ASE.2003.1240330. URL: http://ieeexplore.ieee.org/document/1240330/ (visited on 07/29/2024).

[GS19] Lea Gerling and Klaus Schmid. "Variability-Aware Semantic Slicing Using Code Property Graphs". In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. Paris France: ACM, Sept. 2019, pp. 65–71. ISBN: 978-1-4503-7138-4. DOI: 10.1145/3336294.3336312. URL: https://dl.acm.org/doi/10.1145/3336294.3336312 (visited on 04/15/2024).

[GS20] Lea Gerling and Klaus Schmid. "Syntax-Preserving Slicing of C-based Software Product Lines: An Experience Report". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems.* Magdeburg Germany: ACM, Feb. 2020, pp. 1–5. ISBN: 978-1-4503-7501-6. DOI: 10.1145/3377024.3377029. URL: https://dl.acm.org/doi/10.1145/3377024.3377029 (visited on 04/16/2024).

[GW19] Paul Gazzillo and Shiyi Wei. "Conditional Compilation Is Dead, Long Live Conditional Compilation!" In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Montreal, QC, Canada: IEEE, May 2019, pp. 105–108. ISBN: 978-1-72811-758-4. DOI: 10.1109/ICSE-NIER.2019.00035. URL: https://ieeexplore.ieee.org/document/8805666/ (visited on 06/10/2024).

[Hao+21] Zhang Haojie et al. "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)*. Chengdu, China: IEEE, Dec. 2021, pp. 250–255. ISBN: 978-1-66541-364-0. DOI: 10.1109/ICCWAMTIP53232.2021.9674145. URL: https://ieeexplore.ieee.org/document/9674145/ (visited on 04/29/2024).

[Ios+17] Alexandru Florin Iosif-Lazar et al. "Effective Analysis of C Programs by Rewriting Variability". In: *The Art, Science, and Engineering of Programming* 1.1 (Jan. 2017), p. 1. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2017/1/1. URL: http://programming-journal.org/2017/1/1 (visited on 04/19/2024).

[ISO18] ISO/IEC. *Information Technology - Security Techniques - Information Security Management Systems - Overview and Vocabulary.* Geneva, Switzerland, Feb. 2018.

[JDL19] Liyuan Jia, Wei Dong, and Bailin Lu. "Bug Finder Evaluation Guided Program Analysis Improvement". In: *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*. Dalian, China: IEEE, Oct. 2019, pp. 122–125. ISBN: 978-1-72813-299-0. DOI: 10.1109/ICCSNT47585.2019.8962414. URL: https://ieeexplore.ieee.org/document/8962414/ (visited on 04/17/2024).

[KA08] Christian Kastner and Sven Apel. “Type-Checking Software Product Lines - A Formal Approach”. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. L’Aquila, Italy: IEEE, Sept. 2008, pp. 258–267. ISBN: 978-1-4244-2187-9. DOI: 10.1109/ASE.2008.36. URL: http://ieeexplore.ieee.org/document/4639329/ (visited on 04/18/2024).

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. “Granularity in Software Product Lines”. In: *Proceedings of the 13th International Conference on Software Engineering - ICSE ’08*. Leipzig, Germany: ACM Press, 2008, p. 311. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368131. URL: http://portal.acm.org/citation.cfm?doid=1368088.1368131 (visited on 04/25/2024).

[Käs+11] Christian Kästner et al. “Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation”. In: *ACM SIGPLAN Notices* 46.10 (Oct. 2011), pp. 805–824. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2076021.2048128. URL: https://dl.acm.org/doi/10.1145/2076021.2048128 (visited on 04/22/2024).

[Ken+10] Andy Kenner et al. “TypeChef: Toward Type Checking #ifdef Variability in C”. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. Eindhoven The Netherlands: ACM, Oct. 2010, pp. 25–32. ISBN: 978-1-4503-0208-1. DOI: 10.1145/1868688.1868693. URL: https://dl.acm.org/doi/10.1145/1868688.1868693 (visited on 04/16/2024).

[KGO11] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. “Partial Preprocessing C Code for Variability Analysis”. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. Namur Belgium: ACM, Jan. 2011, pp. 127–136. ISBN: 978-1-4503-0570-9. DOI: 10.1145/1944892.1944908. URL: https://dl.acm.org/doi/10.1145/1944892.1944908 (visited on 05/03/2024).

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Englewood Cliffs, N.J: Prentice Hall, 1988. ISBN: 978-0-13-110370-2 978-0-13-110362-7.

[Kra19] Adam Krafczyk. “Variability-Aware Analysis of C Source Code”. MA thesis. Hildesheim: University of Hildesheim, Nov. 2019. URL: https://sse.uni-hildesheim.de/media/fb4/informatik/AG_SSE/Adam_Krafczyk.pdf.

[Kui+22] Elias Kuiter et al. “Tseitin or Not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester MI USA: ACM, Oct. 2022, pp. 1–13. ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3556938. URL: https://dl.acm.org/doi/10.1145/3551349.3556938 (visited on 05/11/2024).

[Leh80] M.M. Lehman. “Programs, Life Cycles, and Laws of Software Evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805. URL: http://ieeexplore.ieee.org/document/1456074/ (visited on 11/22/2024).

[Li+24] Zongjie Li et al. “Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools”. In: *IEEE Transactions on Dependable and Secure Computing* (2024), pp. 1–18. ISSN: 1545-5971, 1941-0018, 2160-9209. DOI: 10.1109/TDSC.2024.3354789. URL: https://ieeexplore.ieee.org/document/10400834/ (visited on 05/08/2024).

[Lie+10] Jörg Liebig et al. “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*. Cape Town South Africa: ACM, May 2010, pp. 105–114. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806819. URL: https://dl.acm.org/doi/10.1145/1806799.1806819 (visited on 04/18/2024).

[Lie+13] Jörg Liebig et al. “Scalable Analysis of Variable Software”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg Russia: ACM, Aug. 2013, pp. 81–91. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491437. URL: https://dl.acm.org/doi/10.1145/2491411.2491437 (visited on 04/28/2024).

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. “Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code”. In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. Porto de Galinhas Brazil: ACM, Mar. 2011, pp. 191–202. ISBN: 978-1-4503-0605-8. DOI: 10.1145/1960275.1960299. URL: https://dl.acm.org/doi/10.1145/1960275.1960299 (visited on 04/17/2024).

[LL05] V Benjamin Livshits and Monica S Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis”. In: *USENIX Association* 14 (July 2005), p. 18. URL: https://dl.acm.org/doi/10.5555/1251398.1251416.

[Mar+13] Dusica Marijan et al. “Practical Pairwise Testing for Software Product Lines”. In: *Proceedings of the 17th International Software Product Line Conference*. Tokyo Japan: ACM, Aug. 2013, pp. 227–235. ISBN: 978-1-4503-1968-3. DOI: 10.1145/2491627.2491646. URL: https://dl.acm.org/doi/10.1145/2491627.2491646 (visited on 11/02/2024).

[Mei+14] Jens Meinicke et al. “An Overview on Analysis Tools for Software Product Lines”. In: *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*. Florence Italy: ACM, Sept. 2014, pp. 94–101. ISBN: 978-1-4503-2739-8. DOI: 10.1145/2647908.2655972. URL: https://dl.acm.org/doi/10.1145/2647908.2655972 (visited on 04/25/2024).

[Mor+19] Austin Mordahl et al. “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*. Tallinn Estonia: ACM, Aug. 2019, pp. 50–61. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338967. URL: https://dl.acm.org/doi/10.1145/3338906.3338967 (visited on 04/15/2024).

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* San Francisco, Calif: Morgan Kaufmann Publishers, 1997. ISBN: 978-1-55860-320-2.

[Oh+19] Jeho Oh et al. *Uniform Sampling from Kconfig Feature Models.* Technical Report 19. Austin, Texas: The University of Texas at Austin, Department of Computer Science, Sept. 2019, p. 12.

[Oh+21] Jeho Oh et al. “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.* Athens Greece: ACM, Aug. 2021, pp. 893–905. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468578. URL: https://dl.acm.org/doi/10.1145/3468264.3468578 (visited on 05/11/2024).

[Pad09] Yoann Padioleau. “Parsing C/C++ Code without Pre-processing”. In: *Compiler Construction.* Ed. by Oege De Moor and Michael I. Schwartzbach. Vol. 5501. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 109–125. ISBN: 978-3-642-00722-4. DOI: 10.1007/978-3-642-00722-4_9. URL: http://link.springer.com/10.1007/978-3-642-00722-4_9 (visited on 11/09/2024).

[Pat+22] Zachary Patterson et al. “SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C”. In: *Proceedings of the 44th International Conference on Software Engineering.* Pittsburgh Pennsylvania: ACM, May 2022, pp. 2056–2067. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3512763. URL: https://dl.acm.org/doi/10.1145/3510003.3512763 (visited on 05/15/2024).

[Pat23] Zachary J. Patterson. “Toward Applying Variability-Oblivious Static Analyses to Software Product Lines”. PhD thesis. Dallas: The University of Texas at Dallas, Dec. 2023. URL: https://utd-ir.tdl.org/items/1623bed4-684c-44e7-94c2-f20cfeb7c976.

[Pet+19] Tobias Pett et al. “Product Sampling for Product Lines: The Scalability Challenge”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A.* Paris France: ACM, Sept. 2019, pp. 78–83. ISBN: 978-1-4503-7138-4. DOI: 10.1145/3336294.3336322. URL: https://dl.acm.org/doi/10.1145/3336294.3336322 (visited on 08/05/2024).

[Pet+23] Tobias Pett et al. “Continuous T-Wise Coverage”. In: *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A.* Tokyo Japan: ACM, Aug. 2023, pp. 87–98. ISBN: 979-8-4007-0091-0. DOI: 10.1145/3579027.3608980. URL: https://dl.acm.org/doi/10.1145/3579027.3608980 (visited on 05/03/2024).

[PS08] Hendrik Post and Carsten Sinz. “Configuration Lifting: Verification Meets Software Configuration”. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* L’Aquila, Italy: IEEE, Sept. 2008, pp. 347–350. ISBN: 978-1-4244-2187-9. DOI: 10.1109/ASE.2008.45. URL: http://ieeexplore.ieee.org/document/4639338/ (visited on 05/05/2024).

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’95*. San Francisco, California, United States: ACM Press, 1995, pp. 49–61. ISBN: 978-0-89791-692-9. DOI: 10.1145/199448.199462. URL: http://portal.acm.org/citation.cfm?doid=199448.199462 (visited on 05/08/2024).

[Ric53] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 0002-9947, 1088-6850. DOI: 10.1090/S0002-9947-1953-0053041-6. URL: https://www.ams.org/tran/1953-074-02/S0002-9947-1953-0053041-6/ (visited on 06/24/2024).

[RN10] Marko A. Rodriguez and Peter Neubauer. “The Graph Traversal Pattern”. In: (2010). DOI: 10.48550/ARXIV.1004.1001. URL: https://arxiv.org/abs/1004.1001 (visited on 06/16/2024).

[Sah+16] Mohd Zanes Sahid et al. “Combinatorial Interaction Testing of Software Product Lines: A Mapping Study”. In: *Journal of Computer Science* 12.8 (Aug. 2016), pp. 379–398. ISSN: 1549-3636. DOI: 10.3844/jcssp.2016.379.398. URL: http://thescipub.com/abstract/10.3844/jcssp.2016.379.398 (visited on 11/02/2024).

[Sch+22] Philipp Dominik Schubert et al. “Static Data-Flow Analysis for Software Product Lines in C: Revoking the Preprocessor’s Special Role”. In: *Automated Software Engineering* 29.1 (May 2022), p. 35. ISSN: 0928-8910, 1573-7535. DOI: 10.1007/s10515-022-00333-1. URL: https://link.springer.com/10.1007/s10515-022-00333-1 (visited on 04/18/2024).

[Sha+01] Umesh Shankar et al. “Detecting Format String Vulnerabilities with Type Qualifiers”. In: *USENIX Association* 10 (Aug. 2001), p. 16.

[SHB19] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. “PhASAR: An Inter-procedural Static Analysis Framework for C/C++”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Vol. 11428. Cham: Springer International Publishing, 2019, pp. 393–410. ISBN: 978-3-030-17465-1. DOI: 10.1007/978-3-030-17465-1_22. URL: http://link.springer.com/10.1007/978-3-030-17465-1_22 (visited on 05/18/2024).

[Shi+22] Haoxiang Shi et al. “A Software Defect Location Method Based on Static Analysis Results”. In: *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. Wulumuqi, China: IEEE, Aug. 2022, pp. 876–886. ISBN: 978-1-66548-877-8. DOI: 10.1109/DSA56465.2022.00124. URL: https://ieeexplore.ieee.org/document/9914475/ (visited on 04/29/2024).

[SL98] S.S. Some and T.C. Lethbridge. “Parsing Minimization When Extracting Information from Code in the Presence of Conditional Compilation”. In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC’98 (Cat. No.98TB100242)*. Ischia, Italy: IEEE Comput. Soc, 1998, pp. 118–125. ISBN: 978-0-8186-8560-6. DOI: 10.1109/WPC.1998.

693328. URL: http://ieeexplore.ieee.org/document/693328/ (visited on 11/09/2024).

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. “Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation”. In: *Theoretical Computer Science* 167.1-2 (1996), pp. 131–170. ISSN: 03043975. DOI: 10.1016/0304-3975(96)00072-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0304397596000722 (visited on 11/04/2024).

[SRS13] Sandro Schulze, Oliver Richers, and Ina Schaefer. “Refactoring Delta-Oriented Software Product Lines”. In: *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development.* Fukuoka Japan: ACM, Mar. 2013, pp. 73–84. ISBN: 978-1-4503-1766-5. DOI: 10.1145/2451436.2451446. URL: https://dl.acm.org/doi/10.1145/2451436.2451446 (visited on 04/22/2024).

[Tar+12] Reinhard Tartler et al. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *ACM SIGOPS Operating Systems Review* 45.3 (Jan. 2012), pp. 10–14. ISSN: 0163-5980. DOI: 10.1145/2094091.2094095. URL: https://dl.acm.org/doi/10.1145/2094091.2094095 (visited on 04/28/2024).

[Thü+12] Thomas Thüm et al. “Family-Based Deductive Verification of Software Product Lines”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering.* Dresden Germany: ACM, Sept. 2012, pp. 11–20. ISBN: 978-1-4503-1129-8. DOI: 10.1145/2371401.2371404. URL: https://dl.acm.org/doi/10.1145/2371401.2371404 (visited on 06/10/2024).

[Thü+14] Thomas Thüm et al. “A Classification and Survey of Analysis Strategies for Software Product Lines”. In: *ACM Computing Surveys* 47.1 (July 2014), pp. 1–45. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2580950. URL: https://dl.acm.org/doi/10.1145/2580950 (visited on 04/16/2024).

[Tol+24] Rafael F. Toledo et al. “(Neo4j)ˆ Browser: Visualizing Variable-Aware Analysis Results”. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings.* Lisbon Portugal: ACM, Apr. 2024, pp. 69–73. ISBN: 979-8-4007-0502-1. DOI: 10.1145/3639478.3640046. URL: https://dl.acm.org/doi/10.1145/3639478.3640046 (visited on 06/10/2024).

[von+16] Alexander von Rhein et al. “Variability Encoding: From Compile-Time to Load-Time Variability”. In: *Journal of Logical and Algebraic Methods in Programming* 85.1 (Jan. 2016), pp. 125–145. ISSN: 23522208. DOI: 10.1016/j.jlamp.2015.06.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S2352220815000577 (visited on 04/25/2024).

[von+18] Alexander von Rhein et al. “Variability-Aware Static Analysis at Scale: An Empirical Study”. In: *ACM Transactions on Software Engineering and Methodology* 27.4 (Oct. 2018), pp. 1–33. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3280986. URL: https://dl.acm.org/doi/10.1145/3280986 (visited on 04/15/2024).

[von16] Alexander von Rhein. “Analysis Strategies for Configurable Systems”. PhD thesis. Passau: University of Passau, June 2016. URL: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/368 (visited on 09/02/2024).

[Wag+00] David Wagner et al. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities”. In: *Proceedings of Network and Distributed Systems Security (NDSS 2000)* (2000), p. 15.

[Wal+14] Eric Walkingshaw et al. “Variational Data Structures: Exploring Trade-offs in Computing with Variability”. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software.* Portland Oregon USA: ACM, Oct. 2014, pp. 213–226. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661143. URL: https://dl.acm.org/doi/10.1145/2661136.2661143 (visited on 06/02/2024).

[Wei81] Mark Weiser. “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering.* ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0-89791-146-6.

[Woh+24] Claes Wohlin et al. *Experimentation in Software Engineering.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2024. ISBN: 978-3-662-69305-6 978-3-662-69306-3. DOI: 10.1007/978-3-662-69306-3. URL: https://link.springer.com/10.1007/978-3-662-69306-3 (visited on 10/28/2024).

[Yam+14] Fabian Yamaguchi et al. “Modeling and Discovering Vulnerabilities with Code Property Graphs”. In: *2014 IEEE Symposium on Security and Privacy.* San Jose, CA: IEEE, May 2014, pp. 590–604. ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.44. URL: http://ieeexplore.ieee.org/document/6956589/ (visited on 04/15/2024).

[Yam+15] Fabian Yamaguchi et al. “Automatic Inference of Search Patterns for Taint-Style Vulnerabilities”. In: *2015 IEEE Symposium on Security and Privacy.* San Jose, CA: IEEE, May 2015, pp. 797–812. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.54. URL: https://ieeexplore.ieee.org/document/7163061/ (visited on 04/29/2024).

[Yam15] Fabian Yamaguchi. “Pattern-Based Vulnerability Discovery”. PhD thesis. Georg-August-University Göttingen, 2015. DOI: 10.53846/goediss-5356. URL: https://ediss.uni-goettingen.de/handle/11858/00-1735-0000-0023-9682-0 (visited on 04/17/2024).

[Zho+21] Li Zhou et al. “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).* Shenzhen, China: IEEE, Oct. 2021, pp. 381–388. ISBN: 978-1-66541-815-7. DOI: 10.1109/DSC53577.2021.00060. URL: https://ieeexplore.ieee.org/document/9750460/ (visited on 04/29/2024).

A. Appendix

A.1 Compact Variability Encoding for the Example in Listing 3.3

A straightforward solution for handling the variability in the struct in Listing 3.3a is to duplicate the declaration of `y` with the corresponding data types and to apply unique identifiers to the resulting variants. Calls to `y` can then be resolved to the correct variant of the field via an if statement checking for the presence of the `64BIT` feature. However, this leads to incorrect behavior of the call to `sizeof` found in line 13 of Listing 3.3a. Instead of reporting the size of the struct as 16 bytes, the size is reported as 24 bytes.[1] This is because both the `int64_t` and `int32_t` variants of `y` reside in the struct regardless of the selection of the `64BIT` feature. As a result, the size of the struct is artificially increased, which could lead to different program behavior and hence additional imprecisions in the findings reported by an analysis tool. To avoid this undesired effect, code duplication has to be applied more coarse-grained, duplicating not only the field but the struct in its entirety.[2] Each instantiation of the struct then has to be duplicated as well, creating an instance for every struct variant. Accesses to fields or the struct itself (e.g., via `sizeof`) must, in turn, be resolved to the correct instance via if statements or ternary operators. This is illustrated in Listing 3.3b, which shows an improved variability encoded version of the code of Listing 3.3a. By using two variants of the struct (lines 4-7 and 9-12) whose internal structure resembles the structure found within the corresponding product, the size of the struct is not artificially increased. However, this necessitates the creation of an instance for each of the struct variants (lines 15-16). Additionally, corresponding calls have to be delegated to the correct instance based on the selection of the `64BIT`

[1]Assuming that the size of a pointer is 8 bytes and the size of a struct must be a multiple of the alignment of its largest member. In conjunction, these assumptions explain why the size of the struct in the original code is always reported as 16 bytes, regardless of the selection of the `64BIT` feature.

[2]An alternative solution is to transform calls to problematic system functions, such as `sizeof` directly [von+16]. For instance, the naive solution could be corrected by transforming the call to `sizeof` found in line 13 to `sizeof(char*) + (64BIT ? sizeof(int64_t) : sizeof(int32_t))`. However, this solution results in highly case-specific and complex transformations.

feature (lines 18-21). While this improved solution preserves the behavior of the original SPL source code (cf. Listing 3.3a), it reintroduces redundancies of sections shared between products and can therefore noticeably increase the length of the variability-encoded code.

A.2 Detailed Results for Toybox

Figures A.1 to A.3 show detailed results of our proposed analysis approach on Toybox [24p] (version 0.8.11). As described in Section 6.2.2 on page 70, limited support for Toybox in Vari-Joern's product-based analysis approach meant that no baseline was available for this subject system.

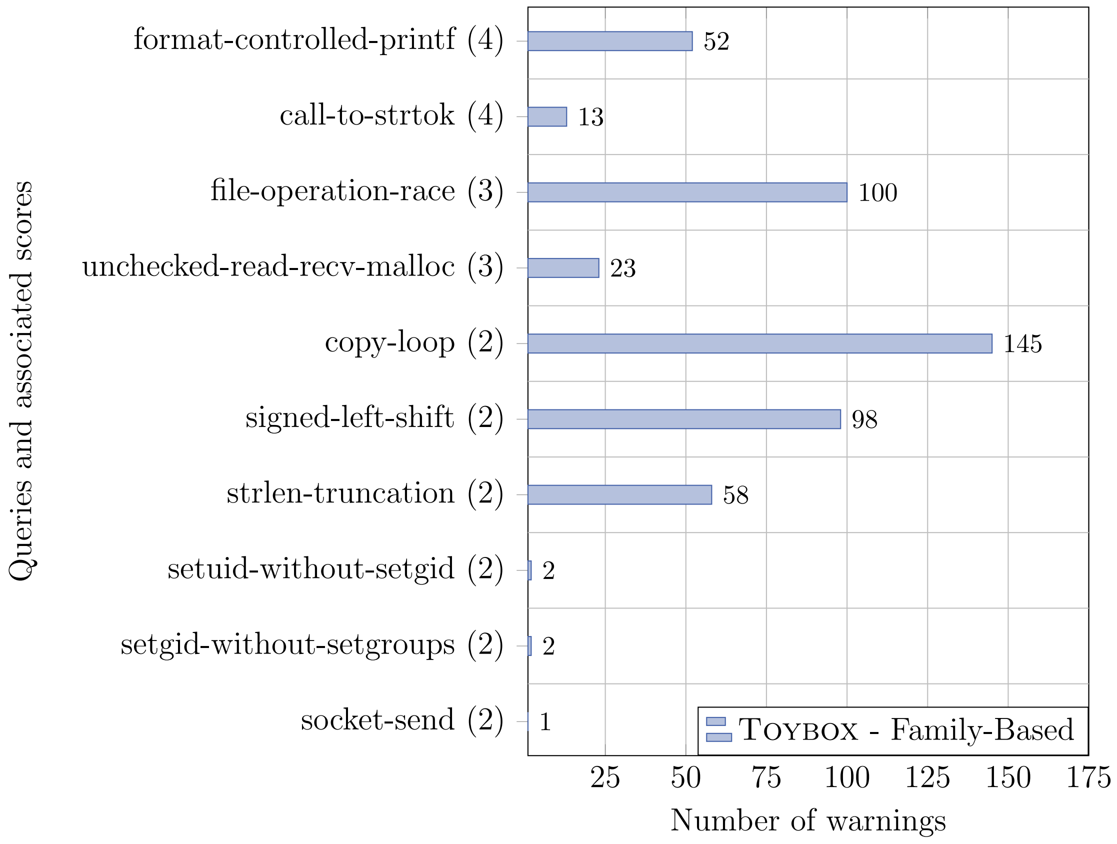


Figure A.1: Warnings reported by our family-based analysis on Toybox aggregated by query

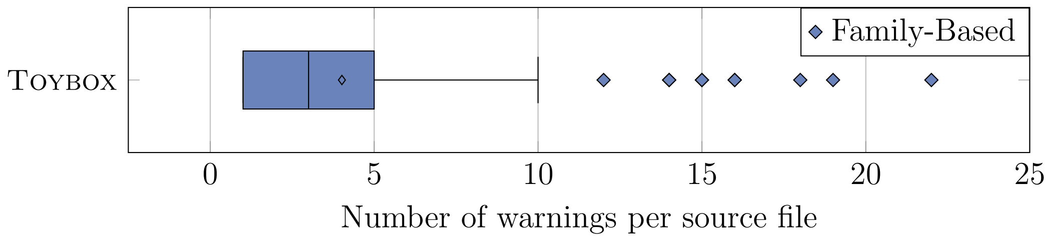


Figure A.2: Distribution of the number of warnings reported by our family-based analysis on Toybox's source files

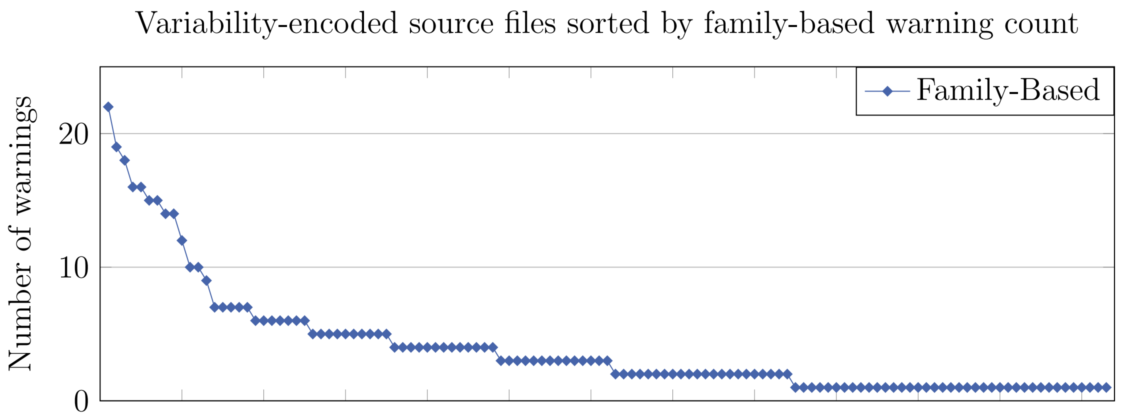


Figure A.3: Warnings reported by our family-based analysis on TOYBOX aggregated by source file