University of Brasília

Exact Sciences Institute
Computer Science Department

# Evolution-Aware Static Analysis of Software Product Lines

Bruno Matissek Worm

Dissertation submitted in partial fullfilment of
the requirements to obtain a Master's Degree in Informatics

Advisor
Prof. Dr. Vander Ramos Alves

Brasília
2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Análise Estática Ciente de Evolução de Linhas de Produtos de Software

Bruno Matissek Worm

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Brasília
2024

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

MM433ee

Matissek Worm, Bruno
Evolution-Aware Static Analysis of Software Product Lines / Bruno Matissek Worm; orientador Vander Ramos Alves. -- Brasília, 2024.
49 p.

Dissertação(Mestrado em Informática) -- Universidade de Brasília, 2024.

1. Software Product Lines. 2. Product Line Analysis. 3. Software Evolution. 4. Functional Programming. 5. Memoization. I. Ramos Alves, Vander, orient. II. Título.

University of Brasília

Exact Sciences Institute
Computer Science Department

# Evolution-Aware Static Analysis of Software Product Lines

Bruno Matissek Worm

Dissertation submitted in partial fullfilment of
the requirements to obtain a Master's Degree in Informatics

Prof. Dr. Vander Ramos Alves (Advisor)
CIC/UnB

Prof. Dr. Ralf Lämmel
University of Koblenz

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordinator of Graduate Program in Informatics

Brasília, November 27, 2024

# Dedicatória

Aos agitados, aos inquietos e aos ansiosos.

# Agradecimentos

À Daniele, minha esposa, pelo companheirismo, apoio, e por ter sido meu pilar de sustentação durante esta jornada.

Aos Professores Vander e Leopoldo, meus orientadores, pela dedicação, seriedade e profissionalismo na condução dos trabalhos. Vocês foram fundamentais para a ótima introdução à vida acadêmica que tive.

Aos membros da banca, Prof. Ralf Lämmel e Prof. Rodrigo Bonifácio de Almeida, pela participação na avaliação do trabalho e sugestões de aprimoramento.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

# Resumo

A necessidade de lidar com a variabilidade durante a análise das Linhas de Produtos de Software (LPS) é intrínseca, pois o número de combinações de produtos válidos pode ser uma função exponencial em relação ao número de características. Além disso, à medida que uma LPS evolui, os resultados das análises anteriores poderiam ser usados para otimizar os cálculos. Entretanto, estas oportunidades de reúso são frequentemente descartadas pelas técnicas de análise de LPS presentes no atual estado da arte. Este trabalho propõe um método para embutir memoização em análises estáticas de Control-Flow Graph (CFG) implementadas em Haskell e reescritas para serem aplicadas em LPS. O método memoizado proposto foi usado para transformar seis análises estáticas de CFG levantadas para LPSs, e comparou-se o desempenho destas em relação às suas contrapartes sem memoização em um conjunto de dez versões da LPS BusyBox. Verificou-se que esta técnica de memoização foi eficiente em reusar os resultados das análises aplicadas em revisões anteriores, com reduções de tempo total computando análises de até duas ordens de magnitude em relação às análises sem memoização, tendo impacto limitado no uso de armazenamento dos resultados memoizados.

**Palavras-chave:** Linhas de Produtos de Software, Análise de Linhas de Produtos, Evolução de Software, Programação Funcional, Memoização

# Abstract

Handling variability in Software Product Line (SPL) analyses is essential due to the vast number of possible valid product combinations, which can grow exponentially with the number of features. Furthermore, as a SPL evolves, results from previous analyses could be used to optimize computations. However, these reuse opportunities are frequently discarded by current *state-of-the-art* SPL analysis techniques. We contribute a method for embedding memoization in Control-Flow Graph (CFG) static analyses implemented in Haskell and rewritten to be applied on Software Product Lines. We compared a set of six memoized analyses with their non-memoized counterparts in a set of ten revisions from the BusyBox SPL. We observed that the memoization technique was effective in reusing the results of the analyses applied in previous revisions, with reductions in total time computing analysis reaching up to two orders of magnitude in relation to the non-memoized analyses while having limited storage consumption impact.

**Keywords:** Software Product Lines, Product Line Analysis, Software Evolution, Functional Programming, Memoization

# Contents

# List of Figures

# List of Tables

# Listings

# Abreviations and Acronyms List

**AST** Abstract Syntax Tree.

**BDD** Binary Decision Diagram.

**BDT** Binary Decision Tree.

**CFG** Control-Flow Graph.

**CPP** C Pre-Processor.

**GQM** Goal Question Metric.

**PC** Presence Condition.

**SPL** Software Product Line.

**VCFG** Variational Control-Flow Graph.

# Chapter 1

# Introduction

A Software Product Line (SPL) is a collection of software systems that share a managed set of features, designed to meet the needs of a specific market segment, and built using a common set of core assets [1]. In the SPL domain, the concept of variability manifests itself in two dimensions: the dimension of space, which concerns the variants that coexist simultaneously, and the dimension of time, which relates to the evolution that a software faces through revisions and versioning [2].

An SPL is designed to generate a family of related products, or variants, each with shared and unique features. Regarding the space dimension, Software Product Line Analysis aims to identify and address issues or errors in each variant. However, analyzing every possible variant is often impractical, as the number of combinations is bounded by an exponential function related to the number of features [3]. Current literature presents methods for efficiently computing analysis considering the space variability aspect in specific scenarios [4, 5, 6, 7, 8, 9]. These methods work by leveraging commonalities in the code as much as possible, following the principles of *late split* — conducting the analysis without considering variability until it becomes necessary — and *early join* – merging identical intermediate results as soon as possible [8]. Many of these transformations have been custom-developed, though recent advances have enabled some to be computed automatically [4].

However, a recognized gap in current SPL analysis techniques is their limited ability to handle both space variability and evolution simultaneously [2]. Therefore, due to this lack of support for evolution-awareness in SPL analysis, it often occurs that neither results from previous computations nor the change information itself are taken into consideration in subsequent analysis. It may result in redundant computation on parts of the SPL that were unaffected by the evolution [2].

In this work, we address this problem by embedding memoization into existing SPL Control-Flow Graph (CFG) static analyses written in Haskell, aiming to reuse results

from previous analyses across evolving revisions of SPLs. This approach allows avoiding computations when valid data is present on cache from previous analysis. To preserve the validity of cached data, we also implemented a *DiffParser* component that detects relevant changes when the SPL evolves, allowing the removal of obsolete values from cache.

To evaluate the memoization technique, we conducted an experiment using ten revisions of the BusyBox project, a widely recognized SPL. The evaluation focuses on comparing the performance of memoized and non-memoized versions of six CFG static analyses, measuring computation time and storage costs across revisions. The results reveal performance improvements for several of the subject analyses, with computation times reduced by up to two orders of magnitude in specific scenarios, while maintaining limited storage consumption.

In summary, the contributions of the present work are the following:

1. The implementation of approach for embedding memoization in SPL CFG static analyses, addressing both spatial and temporal variability[1];

2. The development of supporting infrastructure for detecting relevant changes introduced by SPL source code evolution, allowing discarding no longer valid values from memory [2];

3. An experiment[3] comparing the proposed method to its non-memoized counterpart, assessing the performance impacts when analyzing multiple revisions of an established SPL project (BusyBox).

The remainder of this dissertation is structured as follows:

- Chapter 2 explains some foundational concepts related to SPLs, static analysis, and techniques for handling variability in software systems, establishing the context for better understanding of the method;

- Chapter 3 discusses work related to our research, emphasizing the lack of methods that address both space and time dimensions of variability simultaneously, and presents a motivating example;

- Chapter 4 demonstrates in details the introduction of memoization in CFG-based static analyses for SPLs;

[1] `https://github.com/BrunoMWorm/ProductLineAnalysis/tree/memoization`

[2] `https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments/blob/main/Experiments/source-code/Auxiliary-Scripts/analyze_cfg_diff.py`

[3] `https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments/tree/main`

- Chapter 5 details the experiment comparing the memoized analyses with their non-memoized peers;
- Chapter 6 presents the conclusion, the limitations of our method and future work.

# Chapter 2

# Background

This chapter presents concepts directly related to our research. Section 2.1 outlines key concepts of SPLs, the field that encompass our study. Section 2.2 and 2.3 introduce essential concepts of a technique for computing with variability-aware data, which supports a better understanding of our method.

## 2.1 Software Product Lines

A SPL is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [10]. The goal of applying SPL techniques in software development is to produce tailor-made software with reduced costs, improved quality and efficient time to market cycles [1].

### 2.1.1 Features

From an end-user perspective, a *feature* is a characteristic or visible behavior of a software [1]. Through the process of *feature selection*, a *configuration* can be defined, so that a specific *product* (also called *variant*) can be generated.

However, not all combinations of features are guaranteed to be available, as certain features may impose constraints on the selection of others. These relationships are represented through feature models, which define the rules governing feature selection [1].

### 2.1.2 Software Product Lines Analysis

Below, we briefly outline categories of software analyses that operate independently of code execution, highlighting adaptations that enable their employment in SPLs:

- **Type-Checking:** an analysis method that ensures programs are well-typed according to predefined rules [11], identifying type errors like incompatible casts, missing method declarations, and duplicate class names [3]. In the SPL domain, Type-Checking can be employed to detect variants with type errors [12];

- **Model Checking:** automated verification method for ensuring that a system model satisfies specified requirements, such as safety properties or application-specific requirements [13, 3]. Current literature presents reports of model-checking applied to SPLs for computing reliability properties [14];

- **Static Analysis:** encompasses a range of techniques for examining code without execution [15], including type and model checking. These analyses may vary from lightweight tools that detect stylistic issues and suspicious structures (e.g., linters [16]) to more comprehensive approaches that assess overall program behavior, such as control-flow and data-flow analysis [3]. The lifting method introduced by Shahin and Chechik [4] represents a significant advancement for paving the way for improved adoption of static analyses in SPLs projects.

### 2.1.3 Variability Representation

We presented the concept of feature as a software characteristic desired by the user of such system. To allow the generation of the desired products, an approach of product generation and variability representation must be defined.

The annotation-based approach involves embedding the code of all features into a single source code base, using code snippets with special annotations to associate them with the features [1]. Fig. 2.1 exhibits a classic annotation-based approach for representing variability in C/C++ code bases. The C code contains all product variations, and pre-processing directives are employed to select which code fragments are included or excluded based on feature selection. It is important to highlight that our method is based on transforming analyses that operate on data structures generated from source code managed by an annotation-based approach to variability.

On the other hand, a compositional-based approach is based on modularization of code of different features in separate composable units [1]. The process of product derivation involves the selection and composition of all units into a final valid product. A classic example is a plug-in architecture pattern for extending a framework with selected functionality.

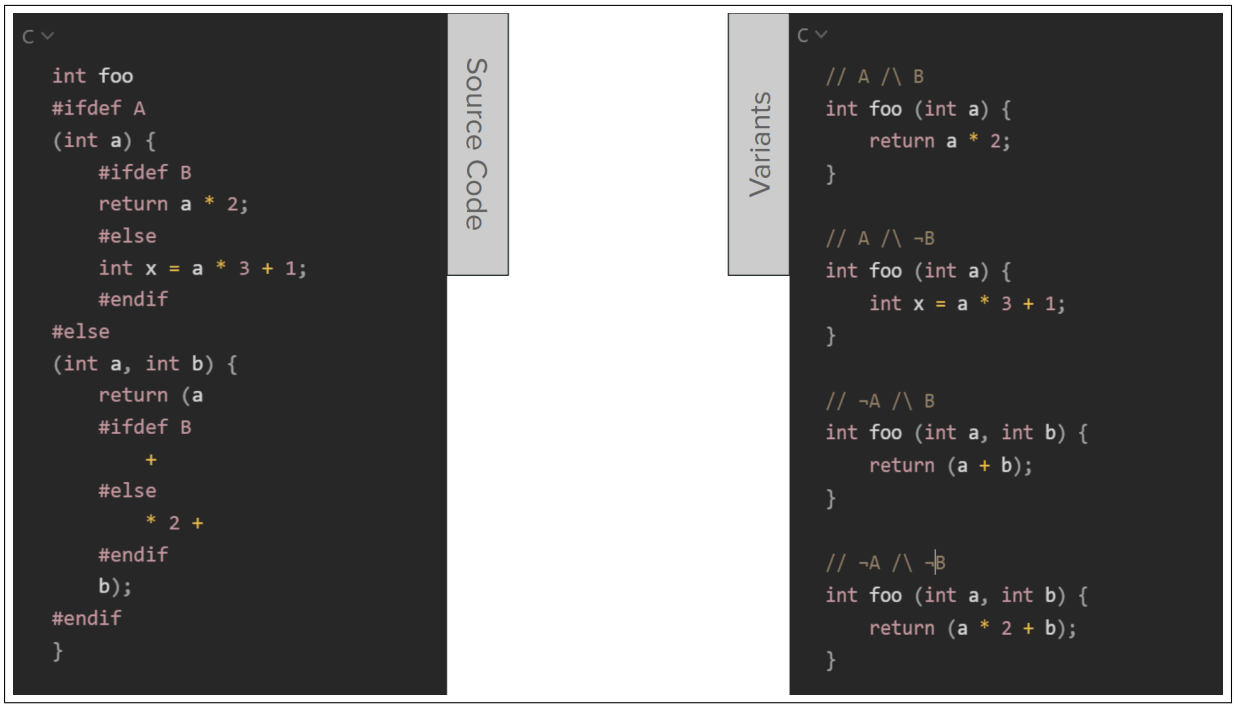



Figure 2.1: Annotation-based method of representing variability through C Pre-processor compiler directives.

## 2.2 Binary Decision Diagrams

In Section 2.3 we are going to explore a technique for encoding SPL variability in efficient data structures, an essential step for enabling effective SPL analysis. Before exploring this method, we introduce an essential structure for representing Boolean functions.

A Binary Decision Diagram (BDD) is a data structures heuristically efficiently in representing and manipulating Boolean functions [17]. It organizes logical expressions graphically, with nodes as decision points and branches showing variable outcomes. By structuring Boolean functions as directed acyclic graphs, BDDs optimize storage and computation by sharing common sub-graphs and reducing redundancy.

Fig. 2.2 displays a comparison between a complete Binary Decision Tree (BDT) and a BDD for the Boolean function A $\wedge \neg$ B. The BDD can be constructed trough a reduction algorithm [17], and its efficiency is based on the reuse of common sub-graphs, making them a more compact representation for Boolean functions.

## 2.3 Variability Encoding

To apply analyses in SPLs, it is essential to have a suitable representation that captures the variability of the underlying artifacts. Variational types and data structures, along

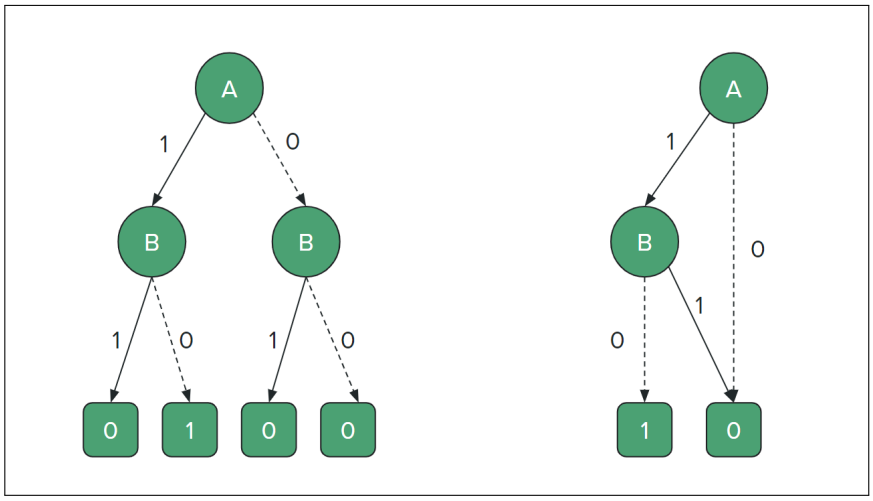



Figure 2.2: Binary Decision Tree (left) and Binary Decision Diagram (right) for the formula A $\wedge \neg$ B.

with various representations and implementations, have been studied in the literature [5]. In this section, we detail the specific implementation presented by Shahin and Chechik [4], which serve as a foundation for our work, as we build upon and extend their Haskell lifting implementation to incorporate memoization.

### 2.3.1 Variability-Aware Values

In variability-aware contexts, such as source code files with annotations associating specific code segments to features, a variable can hold different values simultaneously, depending on the selected feature set. We can use variability-aware types to represent such values in these contexts. A variability-aware value over a type is represented as a set of pairs $(v, pc)$, where $v$ is a value of the aforementioned type, and $pc$ is a presence condition. A *Presence Condition (PC)* is a boolean expression that specifies the set of configurations in which an element exists.

In the example below, the *int* variable $x$ has a value which depends on whether the feature $A$ is defined or not. By using a variability-aware type, we can model this scenario by lifting $x$ to $x^{\uparrow}$, giving it the type $int^{\uparrow}$ (lifted *int*). We can interpret that the variable $x^{\uparrow}$ evaluates to 2 on all configurations where feature $A$ is defined, and 7 otherwise.

```
1   int x;
2   #ifdef A
3   x = 2;
4   #else
5   x = 7;
6   #endif
```

$$x^{\uparrow} = \{(2, A), (7, \neg A)\}$$

For this representation of variability-aware values, there are also two important invariants that must hold: the *disjointness invariant* and the *full coverage invariant*.

### Disjointness Invariant

The disjointness invariant ensures that a variability-aware value has at most one atomic value in any given product configuration. This is achieved by requiring that two presence conditions are unsatisfiable if they represent non-overlapping product sets. Without this invariant, a valid configuration could produce multiple values, resulting in non-deterministic semantics for variability-aware values.

$$v^{\uparrow} = \{(v_1, pc_1), \ldots, (v_n, pc_n)\}, \forall i \neq j : \text{unsat}(pc_i \wedge pc_k)$$

### Full Coverage Invariant

The *Feature Model* $\Phi$ is a propositional formula over all possible features, defining the valid set of configurations. The full coverage invariant ensures that any variability-aware value $v^{\uparrow}$ encompasses the entire product space, including all valid feature combinations.

$$v^{\uparrow} = \{(v_1, pc_1), \ldots, (v_n, pc_n)\}, \bigvee_i pc_i = \Phi$$

### Variability-Aware Type Definition

With both *disjointness invariant* and *full coverage invariant* defined, we can proceed to formally define a variability-aware type. Given a type $T$, its corresponding variability-aware type $T^{\uparrow}$ consists of sets of $(T, PC)$ pairs that satisfy both the disjointness and full coverage invariants:

$$\frac{v_1, \ldots, v_n : T,\ pc_1, \ldots, pc_n : \text{PC} \quad \forall i \neq j \cdot \text{unsat}(pc_i \wedge pc_j) \quad \bigvee_i pc_i = \Phi}{\{(v_1, pc_1), \ldots, (v_n, pc_n)\} : T^{\uparrow}} \quad \text{lifted-type}$$

### Lifted Functions

Functions and operators are also lifted to their corresponding variability-aware types. We denote $f : (a \to b)^{\uparrow}$ as a lifted function $f$ from $a$ to $b$ and $x$ as a lifted value of type $a$. To perform the lifted function application, the **apply** operator is defined. The apply operator generates the cross product of functions and arguments from $x$, combines their presence conditions, and filters out pairs where the conjunction is unsatisfiable. The semantics of the apply operator is described below:

$$\frac{f : (a \to b)^{\uparrow} \quad x : a^{\uparrow}}{(\text{apply } f\ x : b^{\uparrow}) = \{(f'(x'), \text{fpc} \wedge \text{xpc}) \mid (f', \text{fpc}) \in f, (x', \text{xpc}) \in x,\ \text{sat}(\text{fpc} \wedge \text{xpc})\}}\ \text{apply}$$

Although we will not outline the proof here (which can be found in the reference work [4]), the apply operator preserves both the disjointness and full coverage invariants in its outputs.

#### Polymorphic Types

For monomorphic types, lifting involves transforming values into sets of pairs, each consisting of a value and a presence condition, as described in Subsection 2.3.1. For polymorphic types, deep-lifting operates differently: pairs and lists are lifted as follows:

$$[T]^{\uparrow} \Rightarrow [T^{\uparrow}]$$

$$(T_1, T_2)^{\uparrow} \Rightarrow (T_1^{\uparrow}, T_2^{\uparrow})$$

This implies that a lifted list is represented as a list of lifted values. Figure 2.3 illustrates a list of *int* values lifted to its variability-aware type. This approach is more efficient than allocating a complete list for every possible product configuration, as common elements of the list are shared whenever possible. Note that when a value is independent of a specific configuration, the presence condition is defined as the true condition $T$.

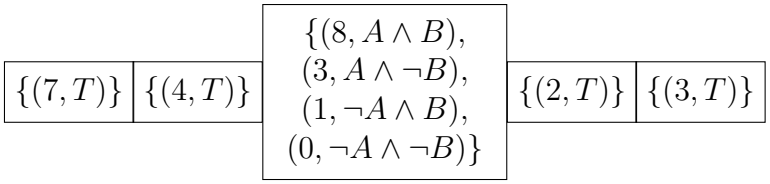



Figure 2.3: Representation of a lifted list of int values

### 2.3.2 Variational Types Implementation in Haskell

An implementation in Haskell for the definitions presented in Subsection 2.3.1 was also presented by Shahin and Chechik [4]. In this Subsection we are going to highlight some important definitions.

### Presence Conditions

The *Presence Condition* type represents a boolean proposition. In this Haskell implementation, boolean propositions are represented using Binary Decision Diagrams (BDDs) structures, with the support of the CUDD library [18].

Listing 2.1: Presence Condition definition in Haskell

```
1  import Cudd.Cudd
2
3  newtype Prop = Prop {
4      b :: DDNode
5  } deriving (Generic, NFData)
6
7  type PresenceCondition  = Prop
```

### Var Type constructor

Variability aware values for a given type $T$ are represented as a list of tuples of values of type $T$ with a corresponding presence condition.

Listing 2.2: Var Type definition in Haskell

```
1  type Val a = (a, PresenceCondition)
2
3  newtype Var t = Var [(Val t)]
4      deriving (Generic, NFData)
```

### Analysis Transformation

Shahin and Chechik also present a transformation method for rewriting functions into their variability-aware counterparts [4]. While we are not going to detail the transformation rules, which are described in the reference work [4], it is worthwhile to compare the type signatures of the original and lifted versions of a Return Checker analysis, designed to detect C functions lacking a return statement. Listings 2.3 and 2.4 show both signature versions. Notably, the lifted analysis's return type is now a variability-aware list of CFGNodes, as the analysis result depends on the specific configuration provided.

Listing 2.3: Original Return Checker Type Signature

```
1  -- Non-Variability-Aware Return Checker Type Signature
2  analyze :: CFG -> [CFGNode]
```

Listing 2.4: Variability-Aware Return Checker Type Signature

```
-- Variability-Aware Return Checker Type Signature
analyze :: Var CFG -> [Var CFGNode]
```

### Variational Control-Flow Graph (VCFG)

The lifting implementation presented by Shahin and Chechik [4] does not support user-defined types out-of-the-box. Nonetheless, for applying CFG analyses on variational data, a suitable variational representation of a CFG must be defined. Listing 2.5 outlines the Haskell implementation representing a Variational Control-Flow Graph (VCFG).

Listing 2.5: CFG and CFGNode type definitions

```
-- Variational CFG structure
data CFG = CFG {
    nodes :: M.ListMultimap Int (CFGNode,
      ↪ PresenceCondition)
}

-- Variational CFGNode
data CFGNode = CFGNode {
    _nID :: Int,
    _fname :: T.Text,
    text :: T.Text,
    ast :: C.NodeType,
    _preds :: [Var Int],
    _succs :: [Var Int]
} deriving (Show, Generic, NFData)
```

Figure 2.4 illustrates how the function defined in Figure 2.1 translates into a Variational Control-Flow Graph (VCFG) structure. Notice that all nodes and edges are associated with a Presence Condition.

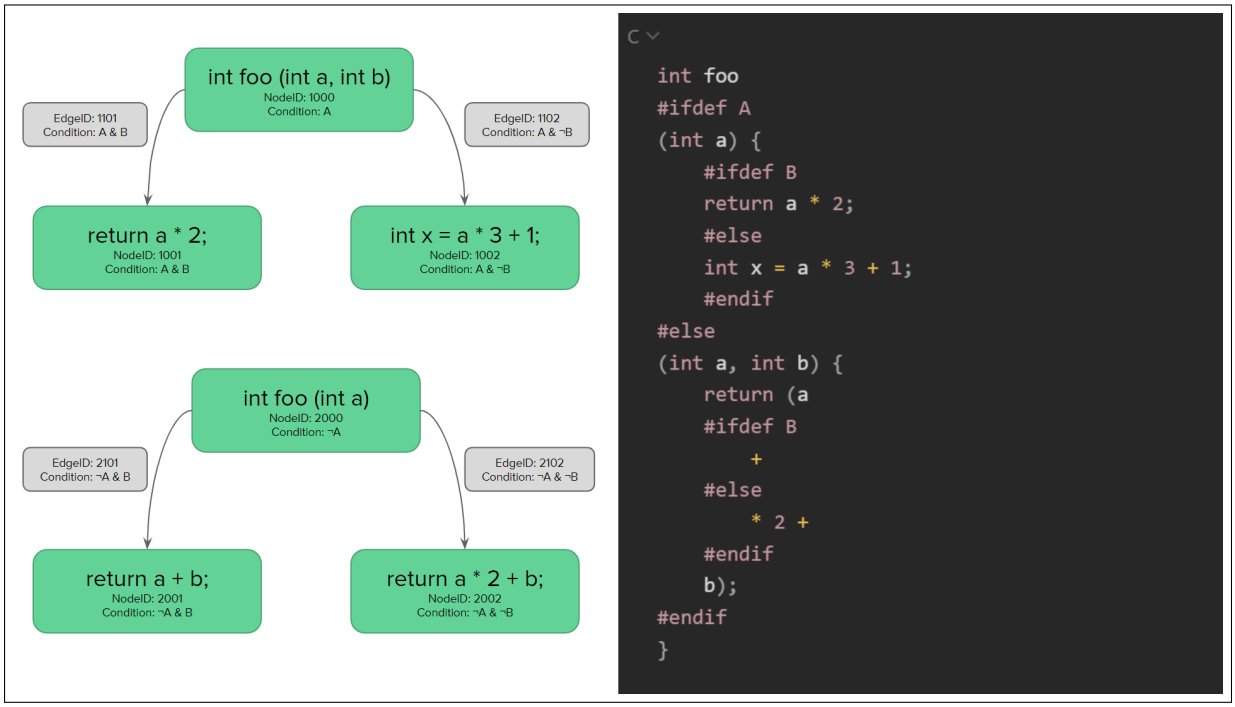



Figure 2.4: Variational Control Flow Graph (VCFG).

# Chapter 3

# Problem Statement

This initial part of this chapter is organized by examining related work within each variability dimension (Sections 3.1 and 3.2), outlining techniques for SPL analysis. Section 3.3 details efforts towards bridging the gap between both variability dimensions. Finally, Section 3.4 presents an example that highlights the problem addressed in this work.

## 3.1 Variability in Space

Variation encoding is a method for representing the inherent variability present in Software Product Lines. Walkingshaw et al. [5] explore trade offs of design decision for implementing data structures for computing data with encoded variability. However, none of the data structures analyzed in their study address dimension of temporal variability.

The TypeChef project [12] marked a significant advancement in variability-aware parsing [7] and type checking [19] for established SPLs like BusyBox and the Linux kernel. Although TypeChef does not handle variability over time, its variational parser was used in this work to parse source code from the Busybox SPL, enabling us to apply both memoized and non-memoized analyses.

Shahin and Chechik [4] propose two approaches for automatically lifting a functional program for embedding variability-awareness. A *shallow-lifting* approach handles the target program as a black-box, concerning variation computation as the combination of all variation inputs, pruning unsatisfiable combinations beforehand. On the other hand, the *deep-lifting* approach is based on source-code transformation, exploring more opportunities of reuse of common variational data. However, neither approach explores variation in time.

## 3.2 Variability in Time

Incremental computation is a technique suited for accommodating changes in the input a program receive. It consists in dividing a computation in sub-parts and reusing the sub-results to compute the desired output. It can reduce redundant computation in cases where the changes in input are small [20]. For functional programs, Carlsson [21] proposes a monadic approach for incremental computation.

The REVISER approach proposed by Arzt and Bodden [22] optimizes inter-procedural data-flow analysis using incremental program changes. Therefore, computation of unnecessary analysis on the unchanged parts of a library are avoided. Though REVISER does not address the space dimension of variability, it provides evidence of the optimization impacts of evolution-aware approaches to static analysis.

Memoization is a classic technique for storing the results of computations with a provided input in memory, so that subsequent calls with the same input are not reevaluated [23]. Wimmer et al. [24] introduce a memoization framework, implemented for Isabelle/HOL [25], that automatically generates a memoized version of an input function, whose correctness theorem is also proved on the process.

## 3.3 Bridging the Variability Dimensions

There has been an effort to bridge the gap between variability in time and variability in space. Thüm et al. [2] discuss possibilities for bridging the gap between analysis aware of either variability in time or variability in space. They explore the different types of analysis that are common to each dimension: regression analysis for variability in time, and product-line analysis for variability in space. Then, they raise the necessity of identifying strategies for lifting an analysis to cope with both dimensions simultaneously and efficiently. Considering this awareness, Ananieva et al. [26] develop a conceptual model that unifies concepts from variation in space and time. It paves the way for clarifying communication between researchers from the software product line engineering and software configuration management areas to achieve efficient solutions.

Significant research has been dedicated to studying evolution highly-variable systems such as the Linux kernel. Dintzner et al. [27] introduce FEVER, a tool that offers detailed insights into changes within variability models and related assets, with a focus on studying feature-oriented changes and artifact co-evolution. Kröher et al. [28] propose a fine-grained approach for assessing the intensity of variability changes across various artifact types, showing in their analysis of the Linux kernel that such changes are relatively rare and typically impact only small segments of the affected artifacts. While these studies do

not propose specific techniques for SPL analysis, their exploration of evolution patterns in highly-variable systems provides valuable information for guiding techniques for enhancing support for SPL evolution.

Higher-order delta modeling is an approach for encapsulating evolution operations in a delta model representing commonalities and variation points of a SPL [29]. The 175% modeling [30] is a formalism that includes possibilities for documenting and analyzing evolving SPLs.

Despite the raised awareness and conceptual advancements highlighted above, there remains a significant gap in the development of practical tools capable of performing static analysis in SPLs with consideration for evolution. Consequently, the field still lacks the necessary tooling to seamlessly integrate and manage both dimensions of variability within SPLs.

## 3.4 Motivating Example

To illustrate the challenges of our problem, we use an example from a configurable C program. This example reveals how different feature combinations in SPLs can introduce configuration-specific issues. While existing solutions address the challenge of efficiently analyzing multiple variants, we are going to highlight how they do not take into consideration changes over time.

Fig. 3.1 demonstrates an issue that can arise in configurable software systems when different code variants are generated based on feature combinations. Using C Pre-Processor (CPP) conditional compilation directives (#ifdef, #else), function *foo* is adapted to behave differently depending on whether features A and B are enabled. In the configuration where A is defined and B is not, the function calculates a value x but lacks a return statement, leading to a potential error in this variant. Only the $\mathbf{A} \wedge \neg \mathbf{B}$ variant displays this problem — a missing return statement, which could lead to unexpected behavior.

One approach to detecting this issue is a Return Checker analysis, a well-known static analysis technique for identifying C functions missing a return statement. However, as shown in Fig. 3.1, multiple code variants can be generated, and only one of these variants exhibits the missing return statement issue.

As discussed in Section 3.1, there are already state-of-the-art techniques for efficiently analyzing such programs. Given a Haskell non-variational implementation of the CFG Return Checker static analysis, one could employ the method described in 2.3 to lift the original Return Checker and obtain a variability-aware version counterpart.

However, consider the patch shown in Fig. 3.2, which fixes the C program by eliminating the missing return statement issue. This patch affects only the $\mathbf{A} \wedge \neg \mathbf{B}$ variant of the

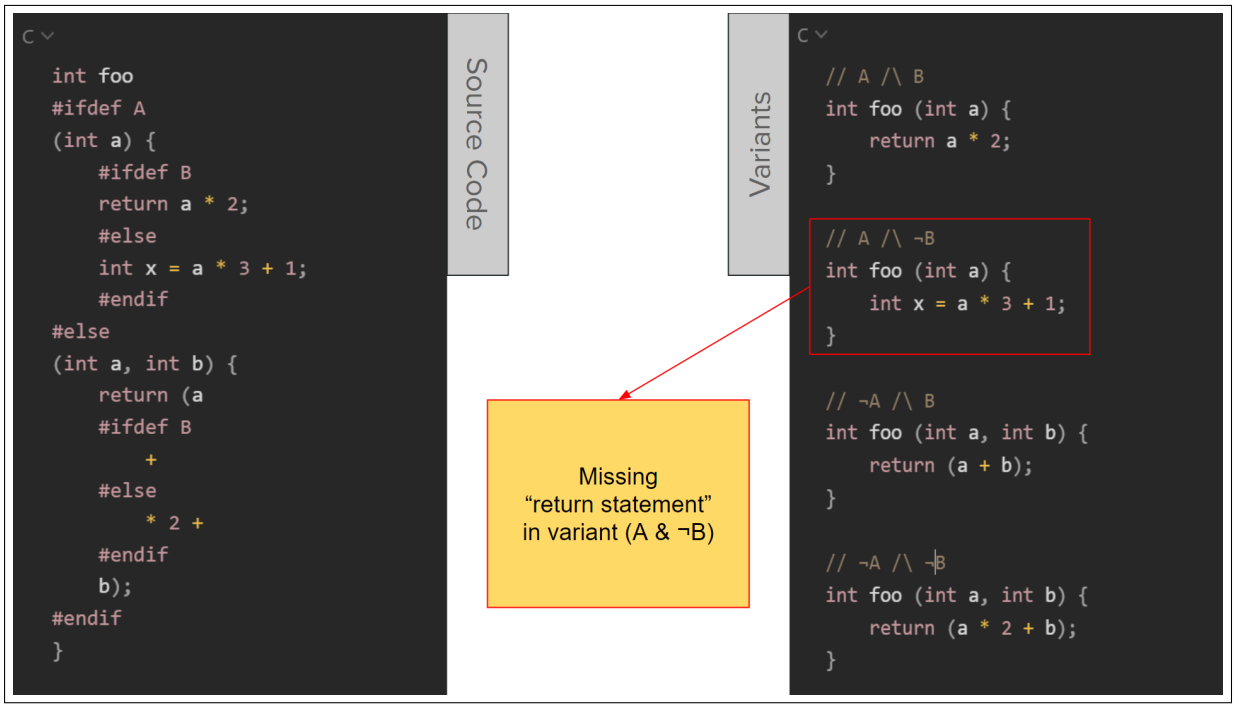



Figure 3.1: Problematic variant in C source code.

program. If we were to rerun the lifted analysis, it would incur in redundant computation on parts of the code which were unaffected by the patch, such as different variants and other functions of the same program. This problem would be aggravated if the number of variants were larger, which is often the case for larger code bases.

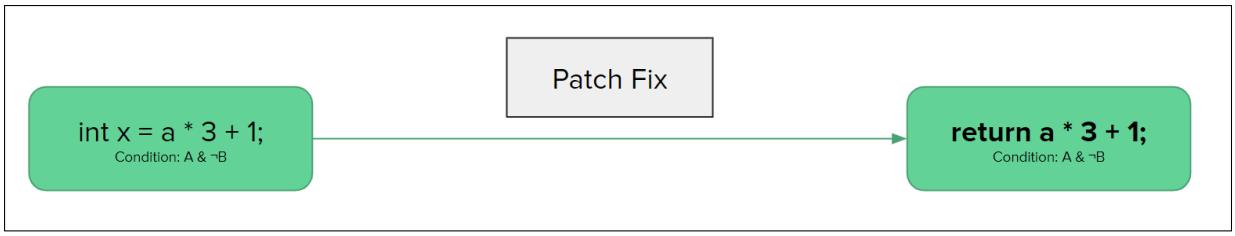



Figure 3.2: Missing return statement fix patch.

To effectively analyze large repositories with high variability and frequent evolution, it is important to consider the scale and complexity of these systems. For example, the BusyBox repository contains more than 17,000 revisions (i.e., commits) and hundreds of thousands of lines of code, each potentially impacting numerous product variants. Static analysis techniques must efficiently consider this variability aspect while accounting for the continuous evolution of the codebase.

To the best of our knowledge, state-of-the-art SPL static analysis methods do not adequately address evolution. While previous work in variability management has led to significant improvements in handling static variability, there is potential for further advancements by integrating these approaches with evolution-aware techniques, possibly enhancing performance by leveraging results from both dimensions.

However, combining these techniques is not straightforward, as each dimension presents its own challenges — variability in space involves handling numerous product combinations, while variability in time requires managing changes across evolving SPL versions.

**Problem Statement**

There is a recognized lack in the state-of-the-art of analysis techniques that are aware of both variability dimensions simultaneously.

# Chapter 4

# Method

To address the problem at hand, we present in this section a method for embedding memoization into SPL CFG static analyses, allowing results from previous analyses on past versions to be reused. In Section 4.1, we provide an overview of the approach. Section 4.2 explains the process of memoizing the selected static analyses, while Section 4.3 introduces our *DiffParser* module, responsible for detecting changes in the SPL over time and ensuring that only valid cached data is reused following each evolution.

## 4.1 Method Overview

Figure 4.1 represents the major components of our method, with their corresponding inputs, outputs and intermediate results. The core components are the following:

- **Lifting and memoization transformation**: responsible for the embedding of memoization into SPL CFG analyses implemented in Haskell and previously rewritten by the deep-lifting method proposed by Shahin and Chechik [4][1];
- **DiffParser:** a module[2] specifically designed to detect changes in the SPL source code, ensuring that relevant modifications are identified. This is necessary to assess which parts of the previous results can be reused through memoization.

### 4.1.1 Method Walkthrough

To illustrate our method and its two core ideas, we provide a step-by-step walkthrough of how an analysis can be memoized and executed:

[1] `https://github.com/BrunoMWorm/ProductLineAnalysis/tree/memoization`

[2] `https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments/blob/main/Experiments/source-code/Auxiliary-Scripts/analyze_cfg_diff.py`

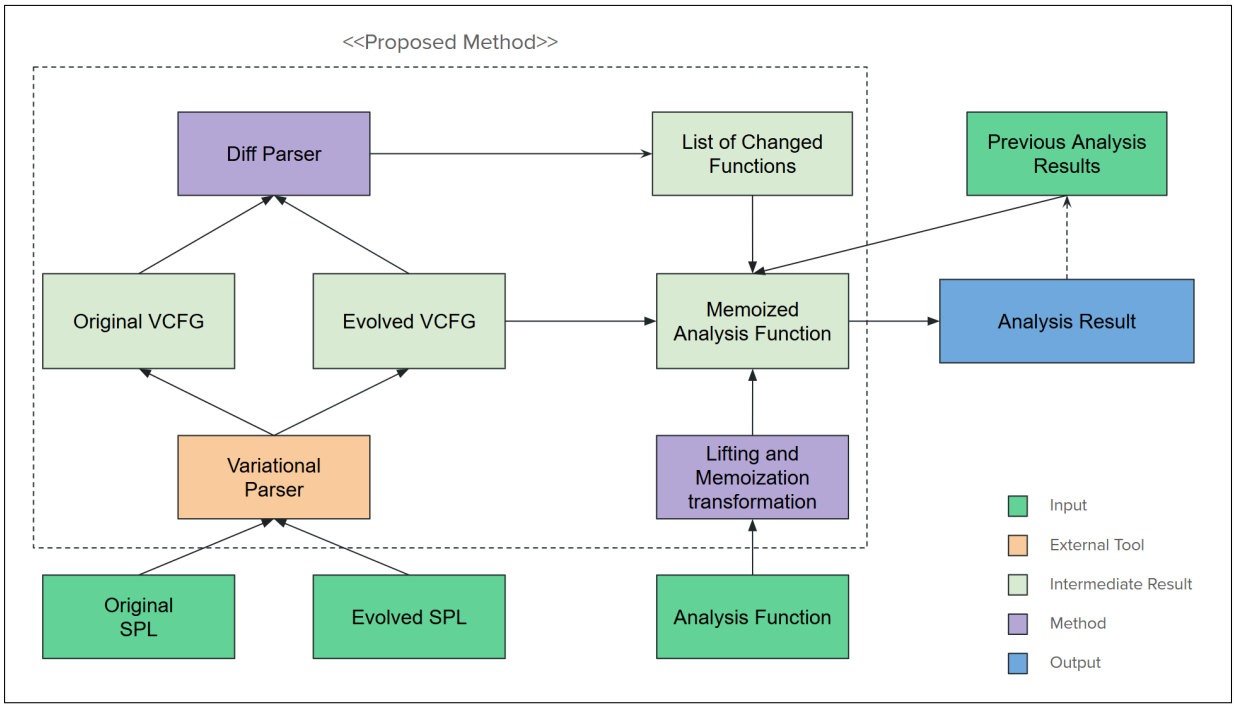



Figure 4.1: Proposed method overview.

1. The original **Analysis Function** goes through a **Lifting and Memoization Transformation**, becoming a **Memoized Analysis Function**;

2. Both **Original** and **Evolved SPLs** are processed by a **Variational Parser** into their corresponding **VCFG** structures;

3. Both **Original** and **Evolved VCFG** structures are passed to the **DiffParser** module that detects the **List of Changed Functions**;

4. Finally, the **Evolved VCFG** structure, the **List of Changed Functions** and the **Previous Analysis Results** (if present) are fed into the **Memoized Analysis Function**, yielding the evolved **Analysis Result**.

## 4.2 Memoization

The key idea in our method is to reuse results from previous computations in the current analysis. To achieve this goal, we used a memoization technique for transforming Haskell functions into their memoized counterparts. Thus, it is not necessary to rerun computations in parts of the input where no modification has occurred.

Our technique is inspired by the transformation rules presented by Wimmer et al. [24]. The aforementioned contribution introduces a framework for automatically rewriting Is-

abelle/HOL functions. We used this framework as a foundation to introduce memoization into each of our Haskell static analyses. However, the primary focus of this work is to demonstrate the feasibility of using memoization in SPL CFG analyses and to assess its performance impact (Chapter 5), rather than to develop a fully automated transformation tool. Therefore, we manually transformed each subject analysis function, and the implementation of a tool for automatically transforming Haskell programs is out of the scope of this dissertation, being regarded as future work.

### 4.2.1 Memoized Fibonacci

To illustrate how our memoization transformation operates, consider the code in Listing 4.1, written in Haskell, that for a given natural number, computes its corresponding value in the Fibonacci sequence:

Listing 4.1: Recursive Fibonacci function implementation in Haskell

```
1  fibonacci :: Int -> Int
2  fibonacci n = if n <= 2
3    then 1
4    else (fibonacci (n - 1)) + (fibonacci (n - 2))
```

This implementation is inefficient due to the exponential complexity caused by the multiple recursive calls. Therefore, it is a classic target for employing a top-down dynamic programming algorithm that employs memoization for caching overlapping calls to compute a certain Fibonacci sequence value.

Inspired on the contribution by Wimmer et al. [24], we implemented a set of functions in Haskell for supporting the process of memoizing other Haskell functions. Listing 4.2 shows a memoized version of the aforementioned Fibonacci function:

Listing 4.2: Memoized Fibonacci function in Haskell

```
1  import Memoization.Core.State
2  import Memoization.Core.Memory
3
4  type StateConc = State MemoryConc
5  type MemoryConc = KeyValueArray Int Int
6
7  memoizedFibonacci :: Int -> StateConc Int
8  memoizedFibonacci n = if n <= 2
9      then return 1
10     else retrieveOrRun n
```

```
            (\_ -> do
                fibonacciNMinus1 <- memoizedFibonacci (n - 1)
                fibonacciNMinus2 <- memoizedFibonacci (n - 2)
                return $ fibonacciNMinus1 + fibonacciNMinus2
            )
```

The *Memoization.Core* module is part of our solution and is available on our GitHub repository[3] and, as mentioned before, contains important functions for memoization purposes. Although we are not going to detail the complete library, it is worth explaining some of its definitions and functions in depth for better understanding of the memoized Fibonacci function presented before.

### 4.2.2 State Monad

In functional programming, functions are typically pure, meaning they don't have side effects or depend on external state. Therefore, a State-Monad based approach is a viable solution for achieving memoization of functions written in a functional programming language such as Haskell.

We implemented a library for the State type with its corresponding instances of the *Functor*, *Applicative*, and *Monad* typeclasses. Although Haskell provides a built-in state monad implementation, we explicitly implemented the *Monad* typeclass for didactic purposes. We also implemented the *runState*, *evalState*, and *execState* functions for working with our *Monad*, as detailed in Listing 4.3:

Listing 4.3: State type definition

```
newtype State m a = State {runState :: m -> (a, m)}

instance Functor (State s') where
    fmap :: (a -> b) -> State s' a -> State s' b
    fmap f am = State $ \s ->
        let (a, s') = runState am s
        in (f a, s')

instance Applicative (State s') where
    pure :: a -> State s' a
    pure x = State (x,)
```

[3] https://github.com/BrunoMWorm/ProductLineAnalysis/tree/memoization/haskell/benchmarks/ControlFlow/src/Memoization/Core

```
12          (<*>) :: State s' (a -> b) -> State s' a -> State s'
               ↪ b
13          fm <*> am = State $ \s ->
14              let (f, s') = runState fm s
15                  (a, s'') = runState am s'
16              in (f a, s'')
17
18      instance Monad (State s') where
19          (>>=) :: State s' a -> (a -> State s' b) -> State s'
               ↪ b
20          h >>= f = State $ \s ->
21              let (a, newState) = runState h s
22                  g = f a
23              in runState g newState
24
25      -- Runs the State computation, discards the new state
26      -- Use when you are only interested in the result of the
           ↪ computation
27      evalState :: State s a -> s -> a
28      evalState act = fst . runState act
29
30      -- Runs the State computation, discards the computation
           ↪ result
31      -- Use when you are only interested in the state
           ↪ transformation
32      execState :: State s a -> s -> s
33      execState act = snd . runState act
```

Therefore, Listing 4.4 shows an example of how to interact with a memoized computation.

Listing 4.4: Interaction with a memoized function

```
1       let initialState = []
2       (newState, result) = runState (memoizedFibonacci 40)
           ↪ initialState
```

### 4.2.3 Memory

For actually storing and retrieve values, we need a specification for types that can represent memory. The *KeyMemory* typeclass in Listing 4.5 illustrates the necessary function definitions.

Listing 4.5: KeyMemory typeclass

```
class KeyMemory k v m where
    mlookup :: k -> State m (Maybe v)
    mupdate :: k -> v -> State m ()
```

We can use any given instance of the *KeyMemory* typeclass for storing and retrieving values from memory. For interacting with the memory, we wrap the computation that is being memoized with the *retrieveOrRun* function highlighted in Listing 4.6. Therefore, given a certain key, computation is executed only if there is no corresponding value in memory for this key.

Listing 4.6: RetrieveOrRun function

```
retrieveOrRun :: (KeyMemory k v m) =>
    k -> (() -> State m v) -> State m v
retrieveOrRun x t =
    mlookup x
        >>= ( \case
            Just v -> return v
            Nothing -> t () >>=
                (\v -> mupdate x v >>= \_ -> return v)
        )
```

We declared a *KeyValueArray* type as a concrete member of the *KeyMemory* typeclass, as detailed in Listing 4.7. It is a simple structure consisting on an array of tuples - each corresponding in a pair of a key and a value.

Listing 4.7: KeyValueArray type

```
type KeyValueArray k v = [(k, v)]

instance Eq k => (KeyMemory k v) (KeyValueArray k v)
  ↪ where
    mlookup :: k -> State (KeyValueArray k v) (Maybe v)
    mlookup a = State (\s -> (lookup a s, s))
    mupdate :: k -> v -> State (KeyValueArray k v) ()
```

```
7           mupdate a v =
8               State
9                 ( \s ->
10                    ((), (a, v) : s)
11                )
```

### 4.2.4 Example of a CFG Static Analysis - Return Checker

Although the Fibonacci function is a suitable example for understanding some of the inner works and type classes of our memoization library, it is still a very simple function which, in its core, accepts an *Int* as an input and yields an *Int* as its output. To better understand how we introduced memoization at a full-fledged SPL CFG static analysis, lets consider again the Return Checker analysis presented in the Problem Statement section.

We are using the exact Haskell implementation which was presented by Shahin and Chechik [4]. Algorithm 1 outlines a simplified explanation of how the used implementation of the Return Check analysis operates:

**Algorithm 1** Variability-Aware Return Checker

1: **Input:** Variational CFG structure
2: **Output:** Variational list of nodes representing functions with no return statements

3: **procedure** VARIABILITY-AWARE RETURN CHECKER
4: **Step 1:** Filter nodes corresponding to function definitions
5: **Step 2:** For each node, apply a "hasReturn" check:
6: **2.1:** Perform a DFS (Depth-First Search) for any return statements
7: **2.2:** If at least one successor is a return statement, the search yields **true**
8: **Step 3:** The Return-Check result is the list of nodes for which the DFS search yielded **false**
9: **end procedure**

With the overall idea of the algorithm, now we are able to inspect its implementation details. Observe the *analyze* function signature in Listing 4.8, which serves as the CFG analysis' entry-point. It accepts a *Var CFG* as its input, representing a C program with a *#ifdef* style of compile-time variability encoding. As its output, it yields a list of *Var CFGNode* structures, which are the CFG nodes representing the roots of functions which do not have any return statement.

Listing 4.8: Analyze function signature

```
1     analyze :: Var CFG -> [Var CFGNode]
```

Both input and output are now more complex structures. Furthermore, as the analysis was transformed through the deep-lifting framework presented in [4], we are now dealing with *Variational* types, which is indicated by the usage of the *Var* type class.

In Section 2.3.2 we presented the definitions of the CFG and CFGNode types. We highlight those definitions in Listing 4.9, so that we can make the two following observations about those structures:

- The CFG structure associates each CFGNode in the multimap with a corresponding *PresenceCondition*. This reflects the fact that the existence of a specific CFGNode may depend on the configuration provided during the SPL compilation process;
- Both the predecessors (*_preds*) and successors (*_succs*) fields in the CFGNode type are variational. The rationale for this is similar: the connections between nodes can vary based on different configurations.

Listing 4.9: Revisiting the CFG and CFGNode type definitions

```
1  -- Variational CFG structure
2  data CFG = CFG {
3      nodes :: M.ListMultimap Int (CFGNode,
       ↪ PresenceCondition)
4  }
5
6  -- Variational CFGNode
7  data CFGNode = CFGNode {
8      _nID :: Int,
9      _fname :: T.Text,
10     text :: T.Text,
11     ast :: C.NodeType,
12     _preds :: [Var Int],
13     _succs :: [Var Int]
14 } deriving (Show, Generic, NFData)
```

With the input and output structures in mind, we can now explore the implementation of the Return Checker in Listing 4.10. The code below provides an overview of the key functions used in the Return Check CFG analysis. While the full implementation is available in our source code repository, for the purpose of explaining the memoization process, for now it is sufficient to focus on the function declarations:

Listing 4.10: Functions used in the Return Checker analysis

```
-- Analyzes a control flow graph (CFG) to find functions
-- that do not contain return statements.
analyze :: Var CFG -> [Var CFGNode]

-- Checks if the node is the root of a function
isFnRoot :: Var CFGNode -> Var Bool

-- Determines if a function (represented by its root node )
-- has a return statement.
hasReturn :: Var CFG -> Var CFGNode -> Var Bool

-- Follows the successors of a node in the control flow graph,
-- checking if it or its successors contain a return statement.
followSuccessor :: Var CFG -> [Var Int] -> Var CFGNode ->  Var Bool

-- Recursively follows all successors of a node,
-- checking if any contain a return statement.
followSuccessors :: Var CFG -> [Var Int] -> Var CFGNode -> Var Bool

-- Finds if a node is in a list of nodes.
find :: Var Int -> [Var Int] -> Var Bool

-- Determines if the node represents a return statement
isReturn :: Var CFGNode -> Var Bool

-- Checks if the node is a function call or declaration
isFuncCall :: Var CFGNode -> Var Bool
```

At this point, the challenges of memoizing the Return Checker analysis become clearer:

- The Fibonacci program uses a **single recursive function** with a **simple primitive type** input and output, making the memoization process relatively straightforward;

- In contrast, the Return Checker analysis involves more **complex data structures** and **multiple interacting functions**, increasing the complexity of the memoization process.

Now we begin to explain our process of memoization. There are two steps involved:

1. First, we inspect the analysis and select which function is going to be memoized;
2. Next, we define a *key* based on the function's input parameters, which will be used to store and retrieve values in memory.

This process requires manual inspection of the analysis source code, which is a current limitation of our method. In the case of the Return Checker analysis, we identified the *hasReturn* function as an ideal candidate for memoization. This decision is supported by the following factors:

- It can be parameterized with a key that uniquely identifies a CFGNode;
- It operates independently of other functions, meaning changes in one function do not propagate to others, preserving the integrity of cached results;
- If a result for a given CFGNode is already stored, memoization avoids the need for a full DFS search within the function definition.

Any Haskell function can be selected for memoization, provided that a suitable memoization key can be derived from its input arguments. As previously mentioned, this key can be a value that uniquely identifies a CFGNode representing a function root. We choose a pair consisting of the function name and the hash of its presence condition as the memoization key. In Listing 4.11 is the non-memoized version of the *hasReturn* function:

Listing 4.11: Non-memoized hasReturn function

```
1  hasReturn :: Var CFG -> Var CFGNode -> Var Bool
2  hasReturn cfg n  = followSuccessors cfg [_nID' n] n
```

And in Listing 4.12 is the memoized version, where the key is used to cache and retrieve results. Notice that now the return type now is a State Monad.

Listing 4.12: Memoized hasReturn function

```
1  -- Memory definition: Tuple of (functionName, hash(
     ↪ presenceCond))
2  type MemoryConc = KeyValueArray (String, Int) (Var Bool)
3  type StateConc = State MemoryConc
```

```
4
5     hasReturn :: Var CFG -> Var CFGNode -> StateConc (Var
        ↪ Bool)
6     hasReturn cfg n@(Var ns) =
7         -- Memoization key definition
8         let fname = show $ _fname (fst (head ns))
9           presenceCond = (snd (head ns))
10        in retrieveOrRun
11            -- Memoization key usage
12            (fname, hash (show presenceCond))
13            ( \_ ->
14                let follow = followSuccessors cfg [_nID' n] n
15                 in return follow
16            )
```

All parts of the program that use the *hasReturn* function must be adapted to handle its new monadic version. This adaptation mainly involves wrapping and unwrapping computations within the state monad, without requiring direct interaction with the memory. The complete implementation can be reviewed in the source code repository[4]. Below in Listing 4.13 is the updated and monadified signature for the *analyze* entry point for the Return Checker:

Listing 4.13: Monadified analyze function signature

```
1     analyze :: Var CFG -> StateConc [Var CFGNode]
```

Now we have successfully obtained a memoized version of the Return Checker CFG analysis. The final step is to implement a mechanism to detect whether the values stored in memory for each function remain valid after the SPL evolves or if the functions have been modified. This process is explained in detail in 4.3.

### 4.2.5 List of memoized CFG Static Analyses

In the previous section we described the basic mechanism employed for memoizing a Haskell SPL CFG static analysis. We applied this process in each of the *deep-lifted* analyses used in the evaluation chapter of the work proposed by Shahin and Chechik [4], as they are also subjects of our experiment detailed in Chapter 5. Table 4.1 describes each static analysis.

[4] https://github.com/BrunoMWorm/ProductLineAnalysis/blob/memoization/haskell/benchmarks/ControlFlow/src/ReturnDeepMemo.hs

| Analysis | Description |
|---|---|
| Call Density | Calculates the average number of function calls per C function within a source file. |
| Case Termination | Checking whether each non-empty case in a C-language switch statement ends with a break statement. |
| Dangling Switch | Checking if there is any dead-code (anything other than a declaration) between a switch statement and the first case or default. |
| Goto Density | Calculates the average number of goto statements per label within a C source file. |
| Return Checker | Checking whether each non-void C function has at least one return statement. |
| Return Average | Calculates the average number of return statements per C function within a source file. |

Table 4.1: Description of the static analyses in which we introduced memoization. The list of analysis is the same as presented by Shahin and Chechik [4].

We inspected each analysis looking for candidate functions suitable for memoization. The rest of each program was rewritten to account for calls to the new memoized function, as they are transformed in State-Monad computations. Table 4.2 describes, for each analysis, the selected functions for memoization. The complete implementation of each analysis is available in our GitHub repository[5].

| Analysis | Memoized function |
|---|---|
| Call Density | *callDensity*: for a given CFGNode representing a function declaration, calculates its number of function calls. |
| Case Termination | *analyze*: for a given complete CFG structure, returns the list of non-empty cases without a break statement. |
| Dangling Switch | *analyze*: for a given complete CFG structure, returns the list of nodes representing switch statements with dead code. |
| Goto Density | *analyze*: for a given complete CFG structure, calculates the average number of goto statements per label. |
| Return Checker | *hasReturn*: for a given CFGNode representing a function declaration, checks if it has at least one return statement. |
| Return Average | *returnAvg*: for a given CFGNode representing a function declaration, computes the number of return statements. |

Table 4.2: Description of the functions selected for memoization in each analysis.

The memoization targets of the Call Density, Return Checker and Return Average analyses are functions that operate on individual CFGNodes. On the other hand, the

[5] `https://github.com/BrunoMWorm/ProductLineAnalysis/tree/memoization/haskell/benchmarks/ControlFlow/src`

memoization targets for the Case Termination, Dangling Switch and and Goto analyses are their entry-points functions, operating on a complete CFG structure level. In such cases, the memoization key is the relative path and name of the C program inside the source code repository.

This approach was taken because no other suitable candidates for memoization were identified within the source code of these analyses. As a result, we opted to memoize the entry-point function itself. A trade-off of this decision is that any change to a function within the program will invalidate the entire cache for that file, rather than just the affected nodes.

## 4.3 DiffParser

The DiffParser module is responsible for tracking changes in the SPL revisions. Figure 4.2 displays an overview of its inputs, outputs and inner components, which are going to be thoroughly detailed in following sections.

- **Inputs:** both original and evolved revisions of the SPL source code are passed as inputs to our DiffParser;
- **Output:** the output of our Diff Algorithm is a list of functions modified by the evolution.

### 4.3.1 Definition of a Modified Function

We explained that the expected output of our diff algorithm is a list of modified functions. First, it is important to clarify this definition. A function is considered modified if any of the following conditions occur in its Abstract Syntax Tree (AST):

- Insertion or deletion of any AST node within the function;
- Modification of the content of any AST node within the function;
- Changes to the *Presence Condition* of any AST node within the function (or of the function itself);
- Alteration in the order of AST nodes within the function.

For the specific analyses outlined in Table 4.1, changes inside a function do not impact the results of nodes outside it, even if the modified node interacts (calls or is called by) with external nodes. These analyses verify properties and compute metrics that are scoped

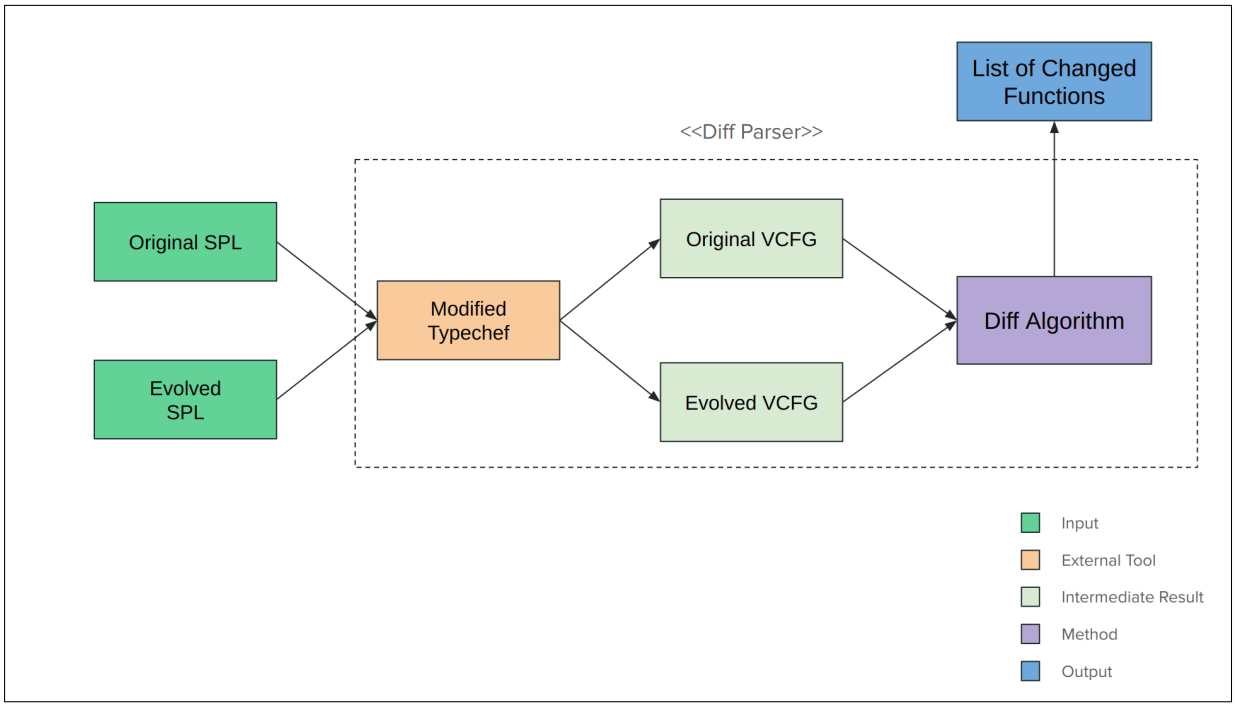



Figure 4.2: DiffParser overview.

to the internal structure and behavior of individual functions. This allows the analysis of each function to be conducted independently.

However, for analyses where modifications within one function may propagate and impact other functions, a different diff algorithm would be necessary. While these scenarios present paths for further exploration, it is beyond the scope of the current work.

### 4.3.2 Modified TypeChef

The first step of the DiffParser method is to generate the adequate data structures. Considering that the actual input of the analyses are VCFG structures parsed by TypeChef [7], we also considered them to be solid input options for our Diff Parsing algorithm.

However, the original TypeChef has a specific limitation for our use case. When it generates identifiers for the VCFG nodes, it uses the *System::identityHashCode* method from the JDK[6]. This method does not guarantee that generated identifiers for the same objects remain consistent from one execution of an application to another, which is a requirement of our algorithm, as we need to track nodes by their identifiers to check for modifications.

[6] `https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()`

Therefore, we modified TypeChef[7] to use customized approach for generating identifiers to nodes. When parsing a source file, it generates, for each CFG node, an identifier which is a hash function application on the following parameters:

- The content of the node;
- The presence condition of the node;
- The container of the node, which can be a specific function or the source code file itself;
- The position of the node inside its container, which can be relative to the source code file or inside a specific function.

### 4.3.3 Diff Algorithm

Having both VCFG structures with deterministically-generated identifiers in hand, we are ready to apply our Diff Algorithm. Algorithm 2 describes this process, whose implementation in our framework was made using Python[8]. The core principle of our algorithm is, for each pair of original/evolved source code files of the SPL, to detect inserted and removed identifiers on the new source code file. For each inserted/removed identifier, we add its container function to the result set. Therefore, the output is a list of names of the modified functions in each file.

Intuitively, the algorithm functions by detecting new identifiers for nodes. Thus, any change in one of the parameters specified in List 4.3.2 will result in a different node identifier being generated, which our diff algorithm promptly detects.

### 4.3.4 Sanitizing stored memoization values

The output of the Diff Algorithm is used prior to executing the memoized analyses. Before running each analysis, the list of modified functions is used to filter out any cached values that are no longer valid. The library that facilitates this process is available in our source code repository[9].

For memoized functions that work with individual CFGNodes (Return, Return Checker, Call Density), cached values corresponding to modified function names are removed from

[7] https://github.com/BrunoMWorm/TypeChef/commit/1e4eda0fe53ac9a91e88a7b70f7093903235a092#diff-200ec32540e611df2f909a6c1e984f55148e29a0c321136eff71c83a10b3dd38L193

[8] https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments/blob/main/Experiments/source-code/Auxiliary-Scripts/analyze_cfg_diff.py

[9] https://github.com/BrunoMWorm/ProductLineAnalysis/tree/memoization/haskell/benchmarks/ControlFlow/src/Serialization

**Algorithm 2** Diff Algorithm for Indentifying Modified Functions from VCFGs

**Input:** $VCFG_{orig}$, $VCFG_{evolv}$ ▷ Original and evolved VCFGs
**Output:** ModifiedFunctions
Construct $Map_{orig}$: Map from node identifiers to their container function in $VCFG_{orig}$
Construct $Map_{evolv}$: Map from node identifiers to their container function in $VCFG_{evolv}$
$InsertedNodes \leftarrow \{id \mid id \in VCFG_{evolv} \wedge id \notin VCFG_{orig}\}$
$RemovedNodes \leftarrow \{id \mid id \in VCFG_{orig} \wedge id \notin VCFG_{evolv}\}$
$ModifiedFunctions \leftarrow \emptyset$
**for all** $id \in InsertedNodes \cup RemovedNodes$ **do**
    **if** $id \in InsertedNodes$ **then**
        $func \leftarrow Map_{evolv}[id]$
    **else if** $id \in RemovedNodes$ **then**
        $func \leftarrow Map_{orig}[id]$
    **end if**
    **if** $func \notin ModifiedFunctions$ **then**
        Add $func$ to ModifiedFunctions
    **end if**
**end for**
**return** ModifiedFunctions

memory. For memoized functions that operate on entire CFG structures (Case Termination, Dangling Switch, Goto Density), if any function in a file has been modified, the entire cached value for that file is invalidated and removed.

# Chapter 5

# Empirical Evaluation

In this chapter we describe the evaluation of the proposed method and report the observed results and discussions. Section 5.1 defines experiment structure. Section 5.2 presents the experiment planning. The results are reported in Section 5.3. Sections 5.4 and 5.5 discuss the findings and threats to validity, respectively.

## 5.1 Definition

We employ the Goal Question Metric (GQM) approach [31] is employed to describe the structure of the experiment. Table 5.1 synthesizes the evaluation goal. The baseline is the set of non-memoized CFG static analyses presented in Section 4.1. Each analysis was previously lifted by the deep-rewriting method presented by Shahin and Chechik [4].

Table 5.1: GQM goal

| | |
|---|---|
| **Purpose** | Compare |
| **Issue** | Performance |
| **Object** | {Evolution,non-evolution}-aware SPL CFG Static Analyses |
| **Viewpoint** | Quality Assurance |
| **Context** | Evolving Annotative SPL |

We defined the following questions and metrics to allow comparison of our method to the baseline. The measurement is addressed in terms of time computing analysis and the associated storage cost for the memoized analyses.

- **Q1:** Compared to the baseline, how faster are memoized lifted analyses?
  - **M1.1:** Average time analysing each revision of the SPL.
- **Q2**: What is the storage cost of employing memoization in SPL CFG static analyses?
  - **M2.1:** Total disk usage.

## 5.2 Planning

### 5.2.1 Subject System Selection

The BusyBox SPL was chosen as the evaluation target due to its use in the experiments conducted by Shahin and Chechik [4]. This also helps to ensure consistency in our comparative analysis and allows us to build on previous infrastructure.

The analyses selected for this study are outlined in Table 4.1, which is the same set of analyses used on the experiment presented by Shahin and Chechik [4]. Our primary objective is to evaluate the impact of memoization on SPL analysis. To achieve this, we compared two versions of each *deep-lifted* static analysis: one incorporating memoization and the other without it. The specific steps where memoization was applied for each analysis are detailed in Table 4.2.

### 5.2.2 Revision Selection

Given that the evaluation target is a well-established SPL with its source code and history accessible on GitHub, we were able to select a set of Git *commits* to serve as our revision points. This selection ensures that our evolution scenario reflects real-world modifications within the SPL.

Table 5.2 displays the selected revisions along with the number of C source code files. The sampling process involved using the command *git log –pretty=oneline 1_18_0...1_19_0* to generate a list of *commits* between these versions. Starting from the first *commit* in version 1_18_0, we selected nine additional *commits* at intervals of ten *commits* each. This approach was chosen to capture a representative subset of modifications, particularly those affecting C source-code files. The decision to focus on only 10 *commits* was driven by the need to balance comprehensiveness with the computational cost of running the subsequent experiments, which are time-consuming.

### 5.2.3 Experiment Design and Analysis Procedures

For each Busybox file of each revision, we used the Haskell library *criterion*[1] to benchmark the time spent computing analyses with both our method and the baseline. At least 4 measurements were made to each file. This aims to gather metrics relevant for answering our first research question (***Q1*** - *Comparing to the baseline, how faster are memoized lifted analyses?*). From these benchmarks, we obtained the average and standard deviation for each file. To compute the metrics for the entire SPL, we summed the individual averages

[1] `https://hackage.haskell.org/package/criterion`

| Revision Label | Git *commit* hash | Source code files |
|---|---|---|
| R1 | 5ab20641d687bfe4d86d255f8c369af54b6026e7 | 508 |
| R2 | 1c31e9e82b12bdceeec4f8e07955984e20ee6b7e | 508 |
| R3 | 3f2477e8a89ddadd1dfdd9d990ac8c6fdb8ad4b3 | 508 |
| R4 | 31905f94777ae6e7181e9fbcc0cc7c4cf70abfaf | 509 |
| R5 | 0d6a4ecb30f596570585bbde29f7c9b42a60b623 | 509 |
| R6 | 2f7d9e8903029b1b5e51a15f9cb0dcb6ca17c3ac | 509 |
| R7 | 6088e138e1c6d0b73f8004fc4b4e9ec40430e18e | 509 |
| R8 | 0cd4f3039b5a6518eb322f26ed8430529befc3ae | 509 |
| R9 | 642e71a789156a96bcb18e6c5a0f52416c49d3b5 | 509 |
| R10 | df1689138e71fa3648209db28146a595c4e63c26 | 509 |

Table 5.2: Set of selected *commits*, extracted from the Busybox SPL GitHub mirror, as evaluation revisions.

and calculated the square root of the total variance sum, as we considered measurements in different files to be linearly independent.

As for the second research question (***Q2*** - *What is the storage cost of employing memoization in SPL CFG static analyses?*), we measured the storage costs using the *du* command on the memoization artifacts, which were persisted through the usage of the Haskell bindings[2] for the CUDD library [18]. The exact parameters for both *criterion*[3] and *du*[4] are in the source code repository.

### 5.2.4 Instrumentation and Operation

Each analysis was executed on an Ubuntu 22.04 VM instance hosted on the Digital Ocean cloud provider. Each instance was configured with 2 dedicated Intel(R) Xeon(R) Platinum 8168 vCPUs, with clock frequency of 2.70GHz, 50GB of SSD and 8GB of RAM. Correctness was assessed by comparing the textual result of the outputs of both non-memoized and memoized analyses. The reproducibility package is available at our source code repository[5].

## 5.3 Results

According to the plan laid out in Section 5.2, we now report corresponding results for analysis time (Section 5.3.1) and storage cost (Section 5.3.2).

[2] `https://hackage.haskell.org/package/cudd`

[3] `https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments/blob/main/Experiments/source-code/Auxiliary-Scripts/run_analyses.sh`

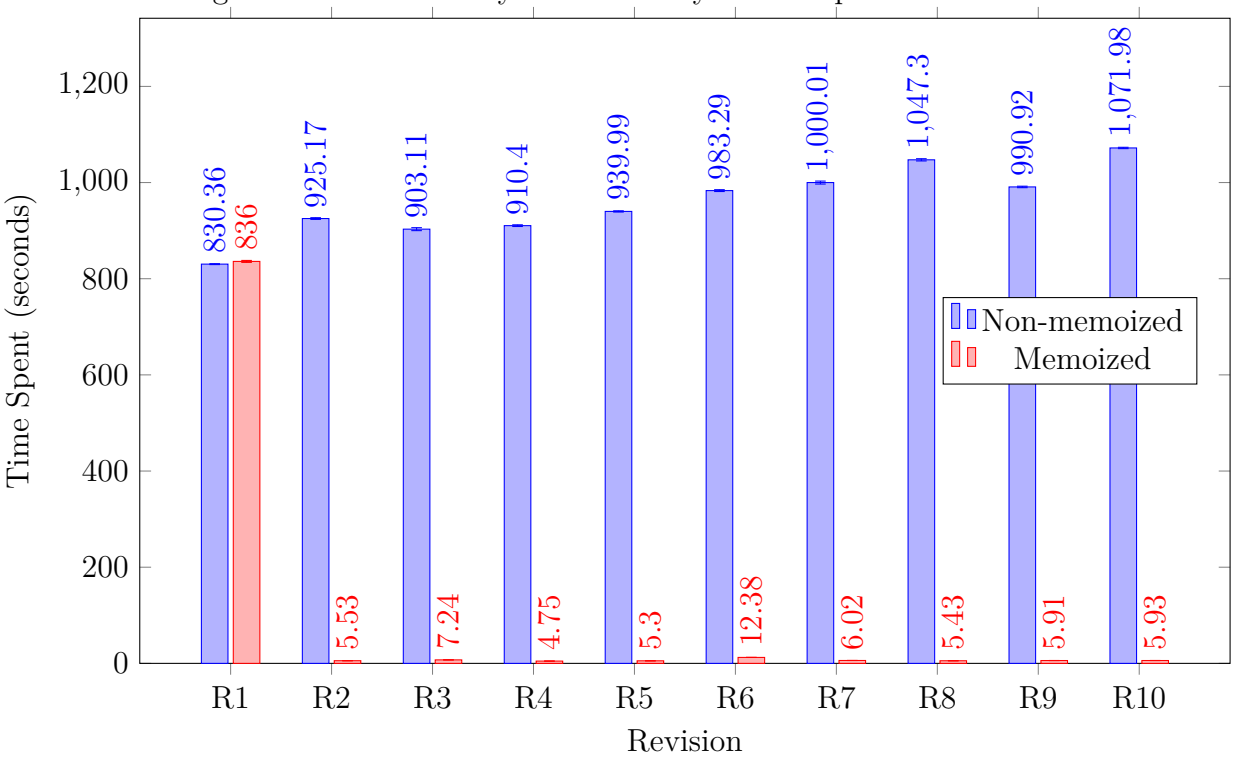
[4] `https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments/blob/main/Results/disk_space.sh`

[5] `https://github.com/BrunoMWorm/Evolution-Aware-SPL-Analysis-Experiments`

### 5.3.1 Analysis Time

Figures 5.1 – 5.6 present the results for analysis time. The X-axis labels each revision as [R1, R2, ..., R10] for conciseness. The Y-axis represents the time it took (in seconds) to complete each analysis on the BusyBox source code files.

Figure 5.1: Call Density - Total Analysis Time per Revision.




### 5.3.2 Storage Cost

Regarding the storage cost specific to the memoized analysis, Table 5.3 displays the total disk space used after applying each analysis in all revisions.

Table 5.3: Storage cost for memoized analyses (kB)

| Analysis | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Call Density | 27,140 | 26,932 | 27,140 | 26,844 | 27,220 | 27,076 | 26,804 | 27,164 | 27,192 | 27,160 |
| Case Termination | 9,084 | 9,056 | 9,056 | 9,072 | 9,072 | 9,072 | 9,072 | 9,072 | 9,072 | 9,072 |
| Dangling Switch | 8,144 | 8,112 | 8,112 | 8,128 | 8,128 | 8,128 | 8,128 | 8,128 | 8,128 | 8,128 |
| Gotos Density | 12,396 | 12,348 | 12,348 | 12,368 | 12,372 | 12,372 | 12,372 | 12,372 | 12,372 | 12,372 |
| Return Checker | 23,624 | 23,592 | 23,592 | 23,672 | 23,672 | 23,672 | 23,676 | 23,536 | 23,688 | 23,684 |
| Return Average | 23,912 | 23,656 | 23,912 | 23,736 | 23,992 | 23,736 | 23,996 | 23,920 | 23,856 | 23,916 |

Figure 5.2: Case Termination - Total Analysis Time per Revision.

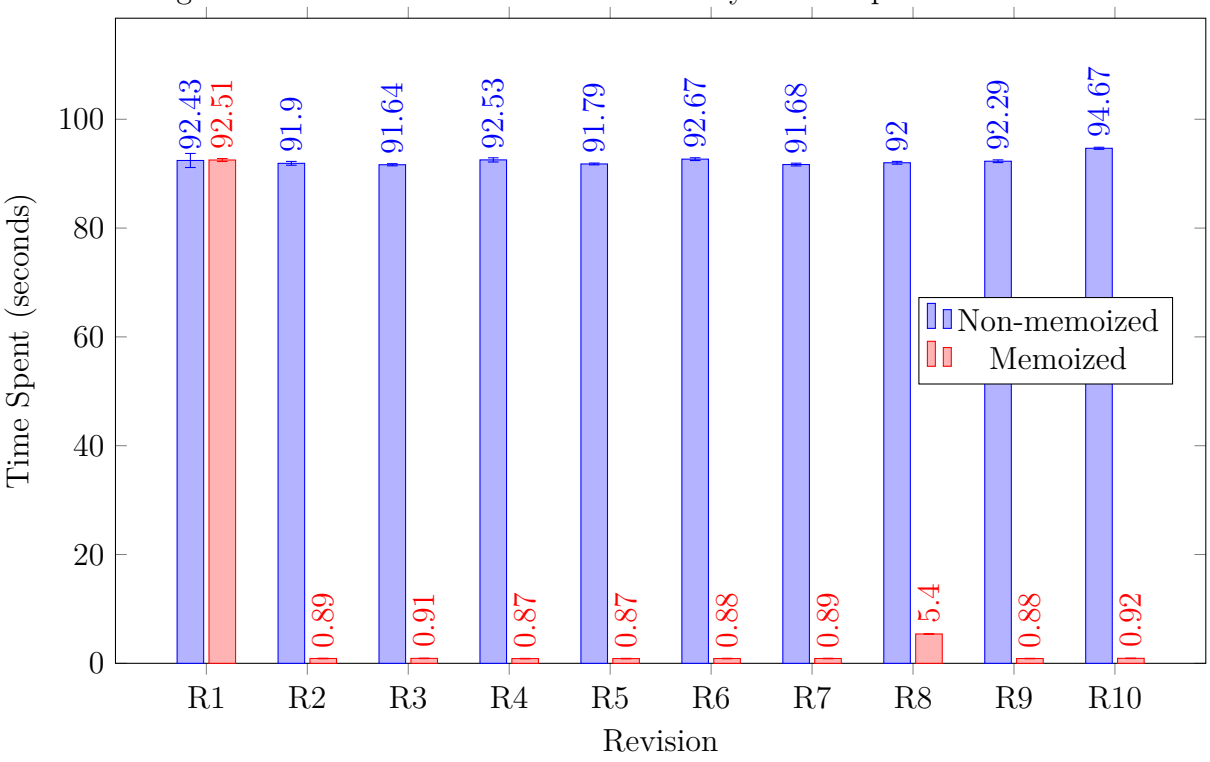



Figure 5.3: Dangling Switch - Total Analysis Time per Revision.

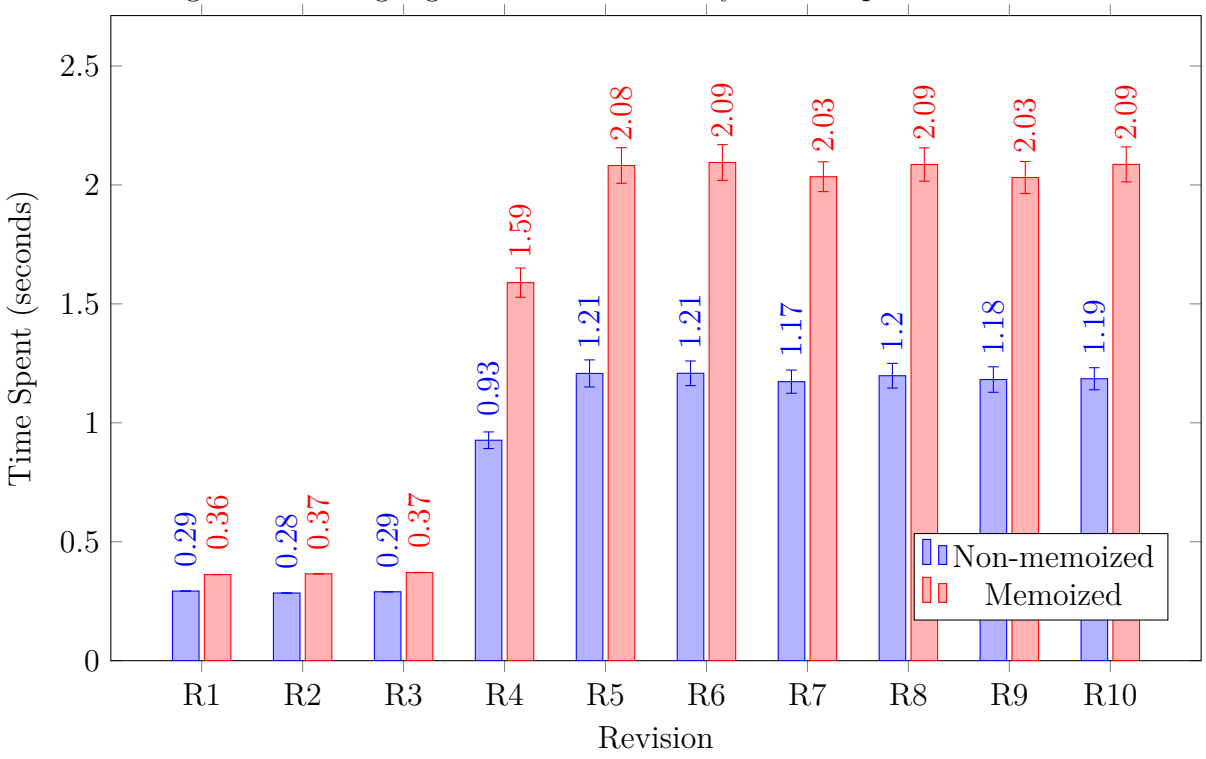


Figure 5.4: Gotos Denisity - Total Analysis Time per Revision.

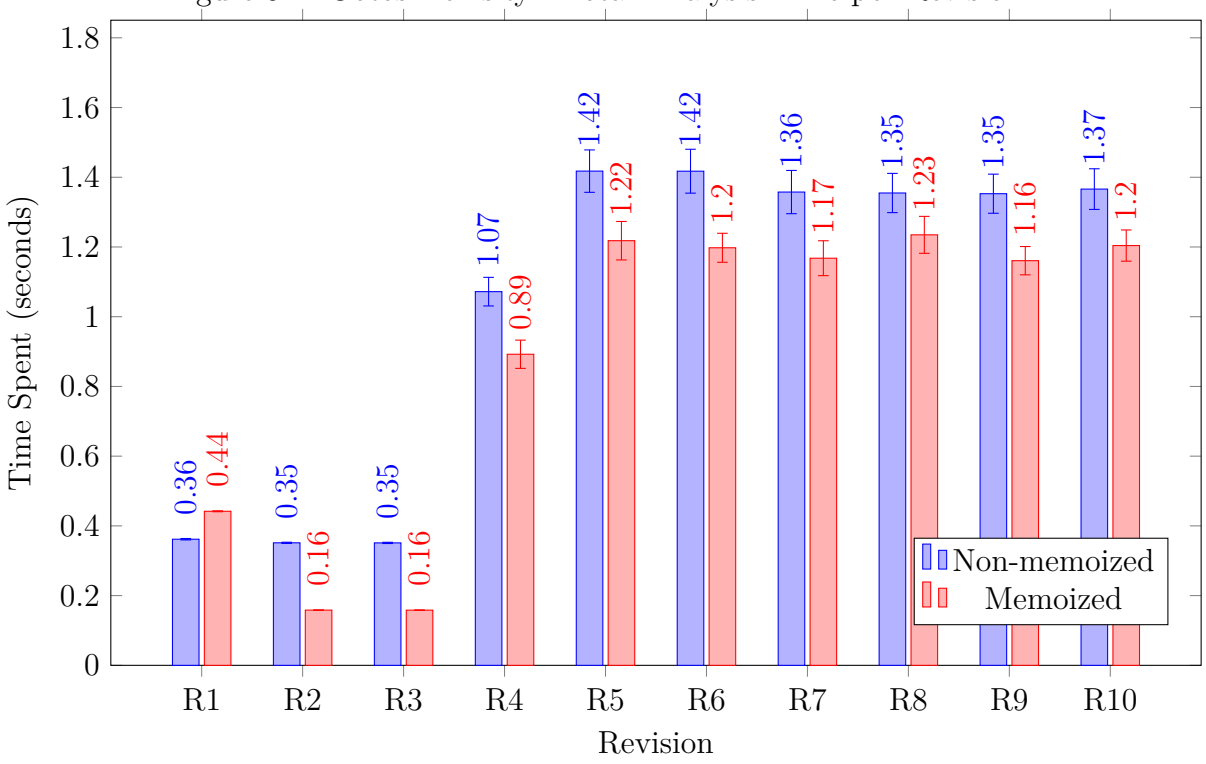



Figure 5.5: Return Checker - Total Analysis Time per Revision.

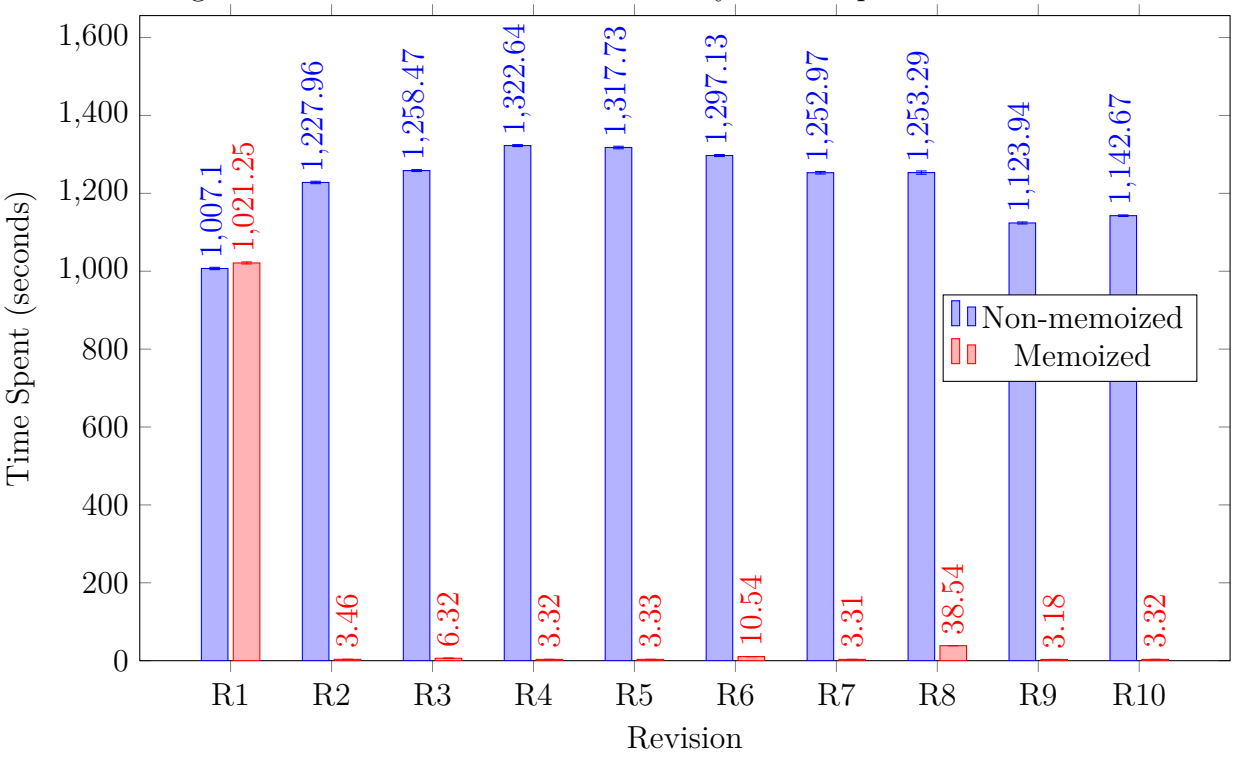
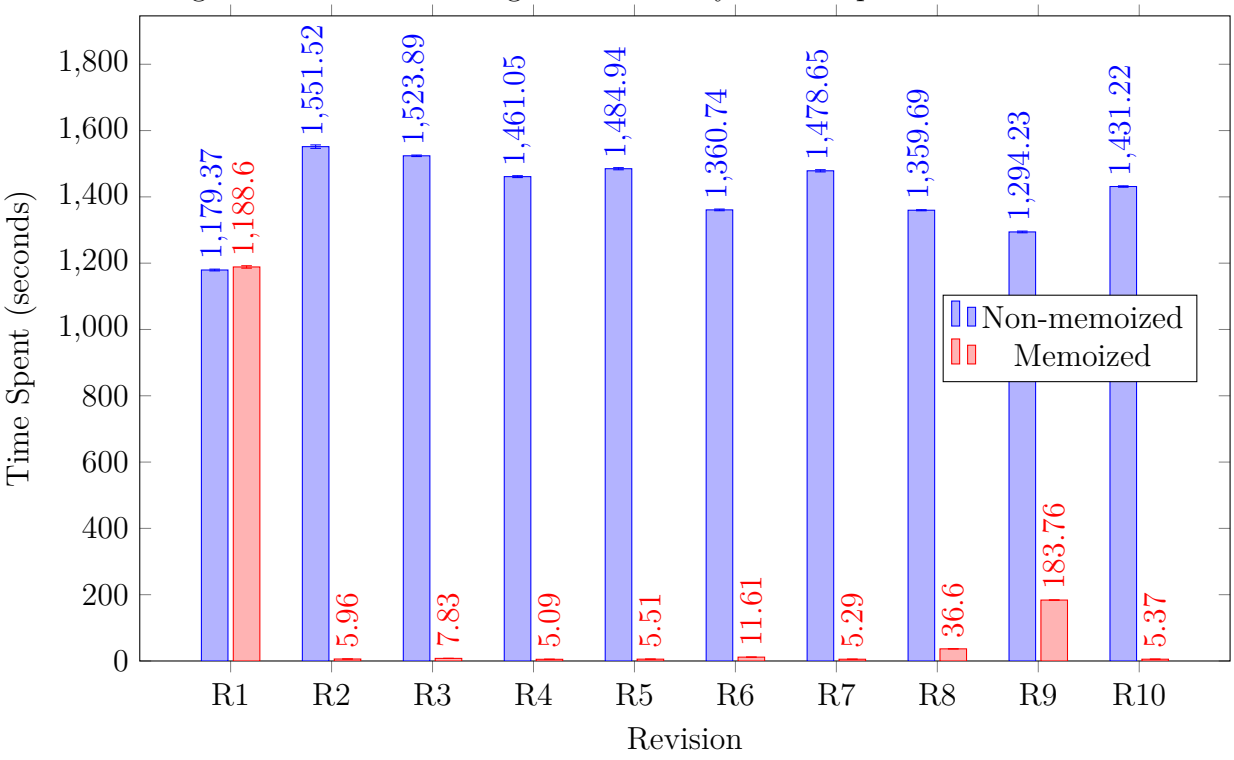


Figure 5.6: Return Average - Total Analysis Time per Revision.




## 5.4 Analysis and Discussion

In this section we examine the results obtained from applying the memoized and non-memoized versions of our SPL static analyses, highlighting performance differences, storage costs and defining metrics for better understanding of the results.

### 5.4.1 Analysis Time

From the results presented in 5.3.1, we observe two patterns of results:

- Figures 5.1, 5.2, 5.5 and 5.6 show similar analysis times for the R1 revision, followed by a consistent performance improvement in favor of the memoized subjects across the remaining revisions;
- In contrast, Figures 5.3 and 5.4 do no exhibit this trend. Notably, Figure 5.3 actually displays a performance penalty for the memoized analysis.

A suitable metric for assessing the performance impact of memoization is the obtained speedup obtained through memoization. Given the presence of multiple revisions, it is helpful to use metrics such as the aggregate, minimum and maximum speedups to better assess performance across revisions. Also, special attention should be given to the measured speedup in the first revision (R1), as for this specific case there are no previous

Table 5.4: Speedup (slowdown) metrics for time spent computing analysis when employing memoization

| Analysis | Aggregate | Minimum | Maximum | R1 |
|---|---|---|---|---|
| Call Density | 10.7x | 79.4x | 193.0x | 0.993x |
| Case Termination | 8.8x | 17.0x | 106.2x | 0.999x |
| Dangling Switch | 0.593x | 0.568x | 0.782x | 0.809x |
| Gotos Density | 1.178x | 1.097x | 2.219x | 0.819x |
| Return Checker | 11.1x | 32.5x | 398.3x | 0.986x |
| Return Average | 9.7x | 7.0x | 286.9x | 0.992x |

values stored for memoization purposes. Therefore, this speedup provides insight into the performance overhead associated with loading and storing values in memory.

The following list summarizes the metrics computed for assessment of our method:

- **Aggregate speedup:** calculated as the ratio of the sum of time spent computing the analysis across all revisions, comparing non-memoized and memoized methods;
- **Minimum/maximum speedup (ex-R1):** calculated by determining, for all revisions except R1, the minimum and maximum speedup of the time spent computing the analysis between non-memoized and memoized approaches;
- **R1 speedup:** calculated as the ratio of time spent computing the analysis specifically for R1, comparing non-memoized and memoized methods.

Table 5.4, presents these metrics for all six analyses. Greater values indicate greater performance improvements from using memoization—for example, a 1.5x speedup means the memoized analysis performed 50% times faster than its non-memoized counterpart. It is important to note that speedup values lower than 1.0x indicate a slowdown, which means that the use of memoization yields slower performance.

The aggregate speedup, calculated by comparing the total time spent on analysis across all revisions, provides insight into the return on investment of using memoization across multiple revisions. The most notable performance gains are seen in the Return Checker (11.1x), Call Density (10.7x), Return Average (9.7x), and Case Termination (8.8x). However, the performance gap narrows for Gotos Density (1.178x), and a performance penalty is observed for Dangling Switch (0.593x).

A likely explanation for the limited effectiveness of memoization in Gotos Density and Dangling Switch analyses is their relatively low computational cost. Across all revisions, these analyses consistently complete in under 2.5 seconds for the entire Busybox project. This suggests that memoization may be more beneficial for computationally heavier analyses, where the overhead of (de)serializing data to and from disk is justified.

For all analyses except Dangling Switch, the minimum speedup occurs during the initial analysis (in R1). This is expected, as there are no pre-existing values in storage to reuse for computations. As a result, performance is impacted due to the need to serialize and store the initial data, which is observed by speedups smaller than 1.0x.

In subsequent revisions, we can observe substantial improvements in performance. For the Return Checker, a maximum speedup of 398.3x occurs for R4, indicating that the memoized analysis was nearly 400 times faster than the non-memoized analysis for this specific revision. Similar performance improvementes are observed for the Return Average (286.9x), Call Density (193.0x) and Case Termination (106.2x).

**Memoization Performance Impact**

Memoization can yield analysis times up to 398.3 times faster than non-memoized counterparts in later revisions.

### 5.4.2 Storage Cost

For all six analyses, storage consumption remained stable across all revisions, with the percentage difference between the maximum and minimum consumption never exceeding 1.6%. Table 5.5 shows the average storage cost and standard deviation considering all revisions.

Table 5.5: Average storage cost and standard deviation (kB)

| Analysis | Average storage cost (kB) | Standard deviation |
|---|---|---|
| Call Density | 27,067.2 | 143.16 |
| Case Termination | 9,070.0 | 7.85 |
| Dangling Switch | 8,126.4 | 8.62 |
| Gotos Density | 12,369.2 | 12.91 |
| Return Checker | 23,640.8 | 49.27 |
| Return Average | 23,863.2 | 109.70 |

**Memoization storage cost**

The storage cost of retaining previous results consistently remains below 28 MB for all analyses.

## 5.5 Threats to Validity

To address the potential internal validity threat of overlooking the time spent on additional steps specific to our memoized analyses, we explicitly benchmarked the (de)serialization

process of storing and retrieving values. As a result, the time spent on these steps was accounted for in our performance evaluation.

Potential threats to external validity arise from the selection of the subject systems. The selection of BusyBox, a well-established and widely used production SPL, is our first strategy to mitigate this threat. Additionally, we deliberately introduced an interval of 10 commits between each selected BusyBox revision to ensure that a substantial amount of modifications were considered in the analyses. This aims to avoid selecting a set of overly similar revisions, which could bias the results in favor of the memoized analyses. In fact, this setup might even bring a performance disadvantage for the memoized method, as it aggregates changes from multiple commits, and our method would benefit most from smaller changes.

The selection of analyses also poses a potential threat to external validity. Specifically, we relied on the exact set of CFG analyses used in the method evaluation by Shahin and Chechik [4], which may limit the generalizability of our findings to other types of analyses or contexts not covered in their study.

# Chapter 6

# Conclusion

We presented a practical application of memoization in SPL analysis analysis, specifically in CFG static analyses. By embedding memoization, we effectively enabled the reuse of results across evolving revisions of SPLs, avoiding redundant computations. Our implementation demonstrated how memoization, combined with a change detection mechanism like the DiffParser, can maintain data validity while handling variability in both space and time dimensions.

Our empirical evaluation of the memoization method confirms its effectiveness in improving the performance of SPL analyses across evolutions. By applying the method to multiple revisions of the Busybox SPL, we observed notable reductions in computation time. Notably, for the most resource-intensive analyses such as Return Checker and Return Average, computation times were reduced from thousands of seconds to just tens of seconds, demonstrating the substantial efficiency gains achieved through memoization while having a limited storage cost.

## 6.1 Limitations

Our proposed memoization technique, while effective in optimizing CFG analyses of evolving SPLs, has limitations that affect its general applicability across different contexts.

**Implementation language of analyses.** Our memoization technique was applied specifically to analyses implemented in Haskell, a functional programming language. While the concept of memoization is not inherently tied to any particular programming paradigm, our method leverages the State Monad construct, making it not directly translatable to analyses implemented in other paradigms, such as imperative programming.

**DiffParser dependency on specific analyses.** As discussed in Subsection 4.3.1, the DiffParser currently provides a list of modified functions, which was sufficient to maintain the validity of cached values for our subject analyses. However, this approach

may not be sufficient for all types of CFG analyses, where changes in one function could impact others. In such cases, the DiffParser would need to be extended to capture a broader scope of modifications.

**Dependency on knowledge of the subject analysis implementation.** A limitation of the proposed method is its dependence on knowledge of the implementation of each analysis to effectively embed memoization. As explained in Subsection 4.2.4 the memoization process requires a manual inspection of the analysis code to identify suitable functions for memoization and to define appropriate keys for caching, which is labor-intensive and error-prone.

## 6.2 Future work

Considering the promising results of the empirical evaluation of our method, and considering its current limitations, we look at the following directions for improvement and extension:

**Rewrite rules and automatic rewrite tool.** The most direct continuation of our work is the development of automatic rewrite rules and an accompanying tool for embedding memoization into Haskell-based static analyses. A dedicated framework implementing these rules would eliminate the inspection process of each analysis and thus make memoization technique more practical for broader adoption in SPL analysis.

**Formal proof.** While the proposed method has been empirically validated, future work should focus on developing a formal proof of correctness for the memoized SPL CFG analysis. With a set of rewrite rules, establishing a formal verification would provide stronger guarantees about the reliability and accuracy of the memoized transformations, ensuring that they yield the same results as their non-memoized counterparts.

**Application on other categories of analysis.** A key direction for future research is to extend this approach to other analysis categories, such as data flow analysis, variability-aware type checking, or dependency analysis. These analyses may have different structural requirements and computational patterns, necessitating adaptations of the memoization strategy. Successfully applying memoization to a broader range of analyses would demonstrate its versatility and effectiveness.

# References


[1] Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake: *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013. 1, 4, 5

[2] Thüm, Thomas, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer: *Towards Efficient Analysis of Variation in Time and Space*. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, pages 57–64, Paris France, September 2019. ACM, ISBN 978-1-4503-6668-7. `https://dl.acm.org/doi/10.1145/3307630.3342414`. 1, 14

[3] Thüm, Thomas, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake: *A Classification and Survey of Analysis Strategies for Software Product Lines*. ACM Computing Surveys, 47(1):1–45, July 2014, ISSN 0360-0300, 1557-7341. `https://dl.acm.org/doi/10.1145/2580950`. 1, 5

[4] Shahin, Ramy and Marsha Chechik: *Automatic and efficient variability-aware lifting of functional programs*. Proceedings of the ACM on Programming Languages, 4(OOPSLA):1–27, November 2020, ISSN 2475-1421. `https://dl.acm.org/doi/10.1145/3428225`. 1, 5, 7, 9, 10, 11, 13, 18, 24, 25, 28, 29, 34, 35, 43

[5] Walkingshaw, Eric, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden: *Variational Data Structures: Exploring Tradeoffs in Computing with Variability*. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 213–226, Portland Oregon USA, October 2014. ACM, ISBN 978-1-4503-3210-1. `https://dl.acm.org/doi/10.1145/2661136.2661143`. 1, 7, 13

[6] Erwig, Martin and Eric Walkingshaw: *The Choice Calculus: A Representation for Software Variation*. ACM Transactions on Software Engineering and Methodology, 21(1):1–27, December 2011, ISSN 1049-331X, 1557-7392. `https://dl.acm.org/doi/10.1145/2063239.2063245`. 1

[7] Kästner, Christian, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger: *Variability-aware parsing in the presence of lexical macros and conditional compilation*. SIGPLAN Not., 46(10):805–824, October 2011, ISSN 0362-1340. `https://doi.org/10.1145/2076021.2048128`. 1, 13, 31

[8] Liebig, Jörg, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer: *Scalable analysis of variable software*. In *Proceedings of the 2013*

*9th Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013, ISBN 978-1-4503-2237-9. https://dl.acm.org/doi/10.1145/2491411.2491437. 1

[9] Bodden, Eric, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini: *Spllift: statically analyzing software product lines in minutes instead of years*. SIGPLAN Not., 48(6):355–364, June 2013, ISSN 0362-1340. https://doi.org/10.1145/2499370.2491976. 1

[10] Clements, Paul and Linda Northrop: *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001. 4

[11] Pierce, Benjamin C.: *Types and Programming Languages*. MIT Press, 1st edition, February 2002, ISBN 0262162091. http://ropas.snu.ac.kr/~{}kwang/520/pierce_book.pdf. 5

[12] Kenner, Andy, Christian Kästner, Steffen Haase, and Thomas Leich: *TypeChef: toward type checking #ifdef variability in c*. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 25–32. ACM, 2010, ISBN 978-1-4503-0208-1. https://dl.acm.org/doi/10.1145/1868688.1868693. 5, 13

[13] Clarke, Edmund M., Orna Grumberg, and Doron A. Peled: *Model checking*. MIT Press, Cambridge, MA, USA, 2000, ISBN 0262032708. 5

[14] Lanna, André, Thiago Castro, Vander Alves, Genaina Rodrigues, Pierre Yves Schobbens, and Sven Apel: *Feature-family-based reliability analysis of software product lines*. Information and Software Technology Journal, 94:59–81, 2018, ISSN 09505849. https://linkinghub.elsevier.com/retrieve/pii/S0950584917302197. 5

[15] Nielson, Flemming, Hanne R. Nielson, and Chris Hankin: *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010, ISBN 3642084745. 5

[16] Johnson, S.C.: *Lint, a C Program Checker*. Computing science technical report. Bell Laboratories, 1977. https://books.google.com.br/books?id=b8nWGwAACAAJ. 5

[17] Bryant: *Graph-based algorithms for boolean function manipulation*. IEEE Transactions on Computers, C-35(8):677–691, 1986, ISSN 0018-9340. http://ieeexplore.ieee.org/document/1676819/. 6

[18] Somenzi, Fabio: *Cudd: Cu decision diagram package*. Public Software, University of Colorado, 1997. 10, 36

[19] Kästner, Christian, Klaus Ostermann, and Sebastian Erdweg: *A variability-aware module system*. SIGPLAN Not., 47(10):773–792, October 2012, ISSN 0362-1340. https://doi.org/10.1145/2398857.2384673. 13

[20] Leather, Sean, Andres Löh, and Johan Jeuring: *Pull-ups, push-downs, and passing it around*. In Morazán, Marco T. and Sven Bodo Scholz (editors): *Implementation and Application of Functional Languages*, pages 159–178, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg, ISBN 978-3-642-16478-1. 14

[21] Carlsson, Magnus: *Monads for incremental computing*. SIGPLAN Not., 37(9):26–35, sep 2002, ISSN 0362-1340. https://doi.org/10.1145/583852.581482. 14

[22] Arzt, Steven and Eric Bodden: *Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes*. In *Proceedings of the 36th International Conference on Software Engineering*, pages 288–298, Hyderabad India, May 2014. ACM, ISBN 978-1-4503-2756-5. https://dl.acm.org/doi/10.1145/2568225.2568243. 14

[23] Donald, Michie: *"memo" functions and machine learning*. Nature, 218(5136):19–22, 1968, ISSN 1476-4687. https://doi.org/10.1038/218019a0. 14

[24] Wimmer, Simon, Shuwei Hu, and Tobias Nipkow: *Verified Memoization and Dynamic Programming*. In Avigad, Jeremy and Assia Mahboubi (editors): *Interactive Theorem Proving*, volume 10895, pages 579–596. Springer International Publishing, Cham, 2018, ISBN 978-3-319-94820-1 978-3-319-94821-8. http://link.springer.com/10.1007/978-3-319-94821-8_34, Series Title: Lecture Notes in Computer Science. 14, 19, 20

[25] Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 14

[26] Ananieva, Sofia, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf Reussner: *A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications*. Empirical Software Engineering, 27(5):101, September 2022, ISSN 1382-3256, 1573-7616. https://link.springer.com/10.1007/s10664-021-10097-z. 14

[27] Dintzner, Nicolas, Arie Deursen, and Martin Pinzger: *Fever: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems*. Empirical Softw. Engg., 23(2):905–952, April 2018, ISSN 1382-3256. https://doi.org/10.1007/s10664-017-9557-6. 14

[28] Kröher, Christian, Lea Gerling, and Klaus Schmid: *Identifying the intensity of variability changes in software product line evolution*. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, pages 54–64. ACM, 2018, ISBN 978-1-4503-6464-5. https://dl.acm.org/doi/10.1145/3233027.3233032. 14

[29] Lity, Sascha, Matthias Kowal, and Ina Schaefer: *Higher-order delta modeling for software product line evolution*. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, pages 39–48, Amsterdam Netherlands, October 2016. ACM, ISBN 978-1-4503-4647-4. https://dl.acm.org/doi/10.1145/3001867.3001872. 15

[30] Lity, Sascha, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer: *175% Modeling for Product-Line Evolution of Domain Artifacts*. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*,

pages 27–34, Madrid Spain, February 2018. ACM, ISBN 978-1-4503-5398-4. https://dl.acm.org/doi/10.1145/3168365.3168369. 15

[31] Basili, Victor R., Gianluigi Caldiera, and H. Dieter Rombach: *The goal question metric approach*. In *The Goal Question Metric Approach*, 1994. https://api.semanticscholar.org/CorpusID:13884048. 34