Bachelor's Thesis

# COMMIT-FEATURE INTERACTIONS: ANALYZING STRUCTURAL AND DATAFLOW RELATIONS BETWEEN COMMITS AND FEATURES

SIMON STEUER

October 17, 2023

Advisor:
Sebastian Böhm Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Andreas Zeller CISPA Helmholtz Center for Information Security

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

## Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

## Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

## Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

## Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,________________________ ________________________________

(Datum/Date) (Unterschrift/Signature)

# ABSTRACT

Short summary of the contents in English...a great guide by Kent Beck how to write good abstracts can be found here:

https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html

CONTENTS

1 Introduction 1
1.1 Goal of this Thesis . . . . . 1
1.2 Overview . . . . . 1
2 Background 3
2.1 Code Regions . . . . . 3
2.2 Interaction Analysis . . . . . 4
3 Commit-Feature Interactions 7
3.1 Structural CFIs . . . . . 7
3.2 Dataflow-based CFIs . . . . . 8
3.3 Combination of CFIs . . . . . 8
3.4 Feature Size . . . . . 9
3.5 Nesting Degree of Structural CFIs . . . . . 9
3.6 Implementation . . . . . 10
4 Methodology 13
4.1 Research Questions . . . . . 13
4.2 Operationalization . . . . . 15
5 Evaluation 19
5.1 Results . . . . . 19
5.2 Discussion . . . . . 24
5.3 Threats to Validity . . . . . 24
6 Related Work 27
7 Concluding Remarks 29
7.1 Conclusion . . . . . 29
7.2 Future Work . . . . . 29
A Appendix 31

Bibliography 33

# LIST OF FIGURES

Figure 4.1 Illustration of different commit kinds . . . . . . . . . . . . . . . . . . . . 15
Figure 5.1 Feature Structural CFIs Plot . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Figure 5.2 Proportional Dataflow-Plot for Commits . . . . . . . . . . . . . . . . . . 24

# LIST OF TABLES

Table 5.1 Additional Information for Dataflow Analysis of Commits . . . . . . 24
Table 5.2 Relating Inside Dataflow to Outside Dataflow . . . . . . . . . . . . . . . 25

# LISTINGS

Listing 3.1 Illustration of structural and dataflow-based CFIs . . . . . . . . . . . . 8
Listing 3.2 Use-Case of Structural CFI's nesting degree . . . . . . . . . . . . . . . . 10

# ACRONYMS

# 1 INTRODUCTION

## 1.1 GOAL OF THIS THESIS

## 1.2 OVERVIEW

# 2 BACKGROUND

In this section, we summarize previous research on the topic of code regions and interaction analysis. Thereby we give defintions of terms and dicuss concepts that are fundamental to our work. In the next chapter, we use the introduced definitions and concepts to explain and investigate structural as well as dataflow-based commit-feature interactions.

## 2.1 CODE REGIONS

Software Programs consist of source code lines that are translated into an intermediate representation (IR) upon compilation. IR accurately represents the source-code information and is utilized in many code improvement and transformation techniqes such as code-optimization. Furthermore, static program analyses are conducted on top of IR or some sort of representation using IR.
Code regions are comprised of IR instructions that are consecutive in their control-flow. They are used to represent abstract entities of a software project, such as commits and features. For this, each code region carries some kind of variability information, detailing its meaning. It should be noted that there can be several code regions with the same meaning scattered across the program. We dicuss two kinds of variablity information, namely commit and feature variability information, allowing us to define two separate code regions.
Commits are used within a version control system to represent the latest source code changes in its respective repository. Inside a repository revision, a commit encompasses all source code lines that were added or last changed by it.

**Definition 1** (commit regions)**.** The set of all consecutive instructions, that stem from source code lines belonging to a commit, is called a *commit region.*

As the source code lines changed by a commit are not necessarily contigious, a commit can have many commit regions inside a program. To properly address the entire set of these commit regions within a program, we introduce the term regions of a commit. We say that the *regions of a commit* are the set of all commit regions that stem from source code lines of the commit.

In general, features are program parts implementing specific functionality. In this work we focus on features that are modelled with the help of configuration variables that specify whether the functionality of a feature should be active or not. Inside a program, configuration variables decide whether instructions, performing a feature's intended functionality, get executed. Detecting these control-flow dependencies is achieved by deploying extended static taint analyses, such as Lotrack[3].

**Definition 2** (feature regions)**.** The set of all consecutive instructions, whose execution depends on a configuration variable belonging to a feature, is called a *feature region.*

Similarly to commits, features can have many feature regions inside a program as the instructions implementing their functionalities can be scattered across the program. Here, we also say that the *regions of a feature* are the set of all feature regions that stem from a feature's configuration variable.

## 2.2 INTERACTION ANALYSIS

In the previous chapter, we have already shown how different abstract entities of a software project, such as features and commits, can be assigned concrete representations within a program. We can use interaction analyses to infer interactions between them by computing interactions between their concrete representations. This can improve our understanding of them and give us insights on how they are used inside software projects. An important component of computing interactions between code regions is the concept of interaction relations between them introduced by Sattler [4]. As we investigate structural and dataflow-based interactions, we make use of structural interaction ($\circledcirc$) and dataflow interaction ($\rightsquigarrow$) relations in this work.
The structural interaction relation between two code regions $r_1$ and $r_2$ is defined as follows:

**Definition 3** (structural interaction relation)**.** The interaction relation $r_1 \circledcirc r_2$ evaluates to true if at least one instruction that is part of $r_1$ is also part of $r_2$.

The dataflow interaction relation between two code regions $r_1$ and $r_2$ is defined as:

**Definition 4** (dataflow interaction relation)**.** The interaction relation $r_1 \rightsquigarrow r_2$ evaluates to true if the data produced by at least one instruction i that is part of $r_1$ flows as input to an instruction $i'$ that is part of $r_2$.

Essential to the research conducted in this paper is the interaction analysis created by Sattler [4]. Their interaction analysis tool VaRA is implemented on top of LLVM and PhASAR. In their novel approach SEAL[6], VaRA is used to determine dataflow interactions between commits. This is accomplished by computing dataflow interactions between the respective regions of commits. Their approach for this will be shortly discussed here, however we advice their paper for a more thorough explanation.
The first step of their approach is to annonate code by mapping information about its regions to the compiler's intermediate representation. This information is added to the LLVM-IR instructions during the construction of the IR. In their paper, Sattler et al. [6] focus on commit regions, which contain information about the commit's hash and its respective repository. The commit region of an instruction is extracted from the commit that last changed the source code line the instruction stems from. Determining said commit is accomplished by accessing repository meta-data.
The second analysis step involves the actual computation of the interactions. For this, Sattler et al. implemented a special, inter-procedural taint analysis. It's able to track data flows between the commit regions of a given target program. For this, information about which commit region affected it, is mapped onto data and tracked along program flow. In this context, we would like to introduce the concept of taints, specifically commit taints. Taints are used to to give information about which code regions have affected an instruction through dataflow. For example, an instruction is tainted by a commit if its taint stems from a commit region. It

follows, that two commits, with their respective commit regions $r_1$ and $r_2$, interact through dataflow, when an instruction is part of one's commit region, $r_2$, while being tainted by $r_1$. Consequently data produced by the commit region $r_1$ flows as input to an instruction within the commit region $r_2$, matching the dataflow interaction relation.

# 3 COMMIT-FEATURE INTERACTIONS

In this section, we define structural and dataflow-based commit-feature interactions as well as properties related to them. Furthermore, their meaning and relationship inside a software project is explained here. In the Overview chapter, we discussed what purpose commits and features serve in a software project. Commits are used to add new changes, whereas features are cohesive entities in a program implementing a specific functionality. In this work, structural interactions are used to investigate how commits implement features and their functionality. In addition, dataflow interactions are examined to gain additional knowledge on how new changes to a program, in the form of commits, affect features.

## 3.1 STRUCTURAL CFIS

In the background chapter, we dicussed the concept of Code Regions, especially commit and feature regions. Logically, we speak of structural interactions between features and commits when their respective regions structurally interact. This structural interaction between code regions occurs when at least one instruction is part of both regions. This is the case when code regions structurally interact through the structural interaction relation ($\circledcirc$).

**Definition 5.** A commit C with its commit regions $r_{1C}$, $r_{2C}$,... and a feature F with its feature regions $r_{1F}$, $r_{2F}$,... structurally interact, if at least one commit region $r_{iC}$ and one feature region $r_{jF}$ structurally interact with each other, i.e. $r_{iC} \circledcirc r_{jF}$.

Structural CFIs carry an important meaning, namely that the commit of the interaction was used to implement or change functionality of the feature of said interaction. This can be seen when looking at an instruction accounting for a structural interaction, as it is both part of a commit as well as a feature region. From the definition of commit regions, it follows that the instruction stems from a source-code line that was last changed by the region's respective commit. From the definition of feature regions, we also know that the instruction implements functionality of the feature region's respective feature. Thus, the commit of a structural interaction was used to extend or change the code implementing the feature of that interaction. Following this, we can say that the commits, a feature structurally interacts with, implement the entire functionality of the feature. That is because each source-code line of a git-repository was introduced by a commit and can only belong to a single commit. Thus every instruction, including those part of feature regions, is annotated by exactly one commit region.

Knowing the commits used to implement a feature allows us to determine the authors that developed it. This is made possible by simply linking the commits, a feature structurally interacts with, to their respective authors. By determining the authors of a feature, we can achieve a deeper insight into its development than solely focusing on the commits that implemented it.

## 3.2 DATAFLOW-BASED CFIS

Determining which commits affect a feature through dataflow can reveal additional interactions between commits and features that cannot be discovered with a structural analysis. Especially dataflows that span over multiple files and many lines of code might be difficult for a programmer to be aware of. Employing VaRA's dataflow analysis, that is dicussed in section 3.6, facilitates the detection of these dataflow interactions.
Commit interactions based on dataflow were explained in the Interaction Analysis section and can be considered as precursors to dataflow-based commit-feature interactions. Similarly to commits interacting with other commits through dataflow, commits interact with features through dataflow, when there exists dataflow from a commit to a feature region. This means that data allocared or changed within a commit region flows as input to an instruction located inside a feature region. This pattern can also be matched to the dataflow interaction relation ($\rightsquigarrow$) when defining dataflow-based commit-feature interactions.

**Definition 6.** A commit C with its commit regions $r_{1C}$, $r_{2C}$,... and a feature F with its feature regions $r_{1F}$, $r_{2F}$,... interact through dataflow, if at least one commit region $r_{iC}$ interacts with a feature region $r_{jF}$ through dataflow, i.e. $r_{iC} \rightsquigarrow r_{jF}$.

```
int calc(int val) {                  ▷ d93df4a
    int ret = val + 5;               ▷ 7edb283
    if (FeatureDouble) {             ▷ fc3a17d    ▷ FeatureDouble
        ret = ret * 2;               ▷ fc3a17d    ▷ FeatureDouble
    }                                ▷ fc3a17d    ▷ FeatureDouble
    return ret;                      ▷ d93df4a
}                                    ▷ d93df4a
```

Listing 3.1: Illustration of structural and dataflow-based CFIs

The above code snippet contains both structural as well as dataflow-based commit-feature interactions. Commit `fc3a17d` implements the functionality of `FeatureDouble` for this function. It follows that a structural commit-feature interaction can be found between them, as their respective commit and feature regions structurally interact. Commit `7edb283` introduces the variable `ret` that is later used inside the feature region of `FeatureDouble`. This accounts for a commit-feature interaction through dataflow, as data that was produced within a commit region is used as input by an instruction belonging to a feature region of `FeatureDouble` later on in the program.

## 3.3 COMBINATION OF CFIS

When investigating dataflow-based CFIs, it is important to be aware of the fact that structural CFIs heavily coincide with them. This means that whenever commits and features structurally interact, they are likely to interact through dataflow as well. As our structural analysis has already discovered that these commits and features interact with each other, we are more interested in commit-feature interactions that can only be detected by a dataflow analysis. That this relationship between structural and dataflow interactions exists becomes clear when looking at an instruction accounting for a structural CFI. From definition 5, we know that the instruction belongs to a commit region of the interaction's respective commit. It follows that data changed inside the instruction produces commit taints for instructions that use the data as input. Now, if instructions that use the data as input are also part of a feature region of the interaction's respective feature, the commit and feature of the structural interaction will also interact through dataflow. However, such dataflow is very likely to occur, as features are functional units, whose instructions build and depend upon each other. Knowing this we can differentiate between dataflow-based CFIs that occur within the regions of a feature and those where data flows from outside the regions of a feature into it. From prior explanations, it follows that this differentiation can be accomplished by simply checking whether a commit that influences a feature through dataflow, also structurally interacts with it.

## 3.4 FEATURE SIZE

When examining commit-feature interactions in a project, it is helpful to have a measure that can estimate the size of a feature. We can use such a measure to compare features with each other and, thus, put the number of their interactions into perspective. Considering our implementation, it makes most sense to define the size of a feature as the number of instructions implementing its functionality inside a program. As the instructions inside the regions of a feature implement its functionality, we can the define the size of a feature as follows:

**Definition 7.** The *size* of a feature is the number of instructions that are part of its feature regions.

It's possible to calculate the defined size of a feature by calculating the number of instructions in which structural CFIs occur. That is, because every instruction that is part of a feature region accounts for a structural commit-feature interaction, as every instruction is part of exactly one commit region as shown in the beginning of this section. It follows, that we do not miss any instructions that are part of feature regions and do not count any such instruction more than once.

## 3.5 NESTING DEGREE OF STRUCTURAL CFIS

In contrast to commit regions, multiple regions of different features can appear in a single instruction. Here, we say that the number of feature regions inside an instruction equals its nesting degree. With increasing nesting degree of an instruction, it becomes less certain which purpose the instruction serves specifically. It could implement functionality of only a single feature present in the instruction's feature regions or any combination of them. For an instruction with a nesting degree of one, we can be most certain about its purpose, namely implementing the functionality of the one feature present in the instruction. This observation also bears consequences in the way we look at and judge structural CFIs, as we can also define a nesting degree for them:

**Definition 8.** The *nesting degree* of a structural CFI is the minimum nesting degree of all instructions the interaction occurs in.

The nesting degree of a structural CFI is a useful measure to estimate the likelihood for which a commit was used to specifically change or implement functionality of a feature. We can be most certain for a structural CFI with a nesting degree of one and become less certain for interactions with higher nesting degrees.

```
26. if (CheckZero) {                      ▷ b9ea2f3    ▷ CheckZero
27.    if (x == 0) {                      ▷ b9ea2f3    ▷ CheckZero
28.       throw Error();                  ▷ b9ea2f3    ▷ CheckZero
29.    }                                  ▷ b9ea2f3    ▷ CheckZero
30. }                                     ▷ b9ea2f3    ▷ CheckZero
.   ...
.   ...
.   ...
73. if (Min && CheckZero) {               ▷ e49c7a1    ▷ Min,CheckZero
74.    x = min(x, y);                     ▷ e49c7a1    ▷ Min,CheckZero
75.    if (x == 0) {                      ▷ b9ea2f3    ▷ Min,CheckZero
76.       throw Error();                  ▷ b9ea2f3    ▷ Min,CheckZero
77.    }                                  ▷ b9ea2f3    ▷ Min,CheckZero
78. }                                     ▷ e49c7a1    ▷ Min,CheckZero
```

Listing 3.2: Use-Case of Structural CFI's nesting degree

The above code-snippet illustrates how considering the nesting degree of a structural CFI can add important context to its meaning.

We can see that the commit `b9ea2f3` exclusively implements functionality of the `CheckZero`-feature. Still, it also structurally interacts with the `Min`-feature in lines 75-77. The according structural CFI only having a nesting degree of 2, allows us to be less certain of its specifc purpose. At the same time, the structural interaction between `b9ea2f3` and `CheckZero` has a nesting degree of 1, confirming that the interaction's commit was used to specifically implement the `CheckZero`-feature.

## 3.6 IMPLEMENTATION

The detection of structural as well as dataflow-based commit-feature interactions is implemented in VaRA [5]. Additionaly to commit regions, VaRA maps information about its feature regions onto the compiler's IR during its construction. Commit regions contain the hash and repository of their respective commits, whereas feature regions contain the name of the feature they originated from. VaRA also gives us access to every llvm-IR instruction of a program and its attached information. Thus, structural CFIs of a program can be collected by examining its compiled instructions. According to definition 5, we can store a structural interaction between a commit and a feature, if an instruction is part of a respective commit and feature region. As discussed in the previous section, the nesting degree of an instruction equals the number of feature regions it is part of. When encountering another instruction a structural interaction occurs in, we update the interaction's nesting degree as the minimum of its current value and the nesting degree of the inspected instruction. For each structural CFI we also save the number of instructions it occurs in. This is accomplished by incrementing its instruction counter if we happen to encounter a duplicate.

With this, we are also able to calculate the size of a feature based on our collected structural CFIs and their respective instruction counters. Following the explanations from section 3.4, we can compute the size of a feature as the sum over the instruction counters of all found structural CFIs the feature is part of.

In the Interaction Analysis section, we discussed the taint analysis deployed by VaRA. There, VaRA computes information about which code regions have affected an instruction through dataflow. Checking whether a taint stems from a commit region allows us to extract information about which commits have tainted an instruction. Thus, dataflow-based commit-feature interactions can also be collected on instruction level. According to definition 6, we can store a dataflow-based interaction between a commit and a feature, if an instruction has a respective commit taint while belonging to a respective feature region. Consequently said instruction uses data, that was changed by a commit region earlier in the program, as its input.

jetzt auch VaRA-TS erwähnen ??

# 4 METHODOLOGY

The purpose of this chapter is to first formulate the research questions that we examine in our work and then propose our method of answering them.

## 4.1 RESEARCH QUESTIONS

In the Interaction Analysis chapter we dicussed the different meanings of structural and dataflow-based interactions. With this knowledge, we can answer many interesting research topics. These topics include patterns in feature development and usage of commits therin as well as findings about how likely seemingly unrelated commits are to affect features inside a program.

***RQ1: How do commits and features structurally interact with each other?***

We intend to research two main properties which already provide a lot of insight into the development process of features and best practices of commits therein.
Firstly, we examine the number of commits features interact with structurally. This gives us a direct estimate on how many commits were used in the development of a feature. Our analysis also allows us to measure the size of feature, which can put the number of commits used to implement a feature into perspective. We refine our investigation by factoring in the nesting degrees 3.5 of our collected structural CFIs. This allows us to differentiate between the DEFINITE and POTENTIAL size of a feature as well as most likely and potentially implementing commits.
Secondly, we examine how many features a commit interacts with structurally, i.e. how many features a commit usually changes. This is especially interesting when considering best practices surrounding the usage of commits. It is preferred to keep commits atomic[1] meaning they should only deal with a single concern. As different features implement separate functionalities, it's unlikely for a commit to change several features while dealing with the same concern. Transferring this to our work, high quality commits should mostly change a single feature. Acquiring data on this issue might show how strictly this policy is enforced in the development of features across different projects. Here, differentiating between the nesting degrees of structural interactions provides us with more informative results. We can form a lower bound for the number of features a commit usually changes by only considering structural interactions with a nesting degree of one. This way we only take into account commits for which we are certain that they were used to specifically implement functionality of the feature .

***RQ2: How do commits interact with features through dataflow?***

Investigating dataflow inside a program can unveil interactions between its entities that were previously hidden from programmers. This can help them understand the extent to which different parts of a program influence each other. Furthermore, deploying the introduced analysis in a direct manner could ensure improvements in the daily life of a developer. In the context of our work, an example for this could be the faciliation of finding the cause of and subsequently fixing bugs in features. Bugs occuring in certain features could be traced back to the commits responsible for them by factoring in recent commits affecting said features through dataflow.

Previous studies have laid the groundwork for researching dataflow interactions between different abstract entities of a program. While it has shown a wide range of interesting use-cases, it has focused solely on dataflow between commits. That's why we aim to provide first insights into the properties of dataflow-based CFIs. Specifically, we investigate how connected commits and features are by analyzing the number of features a commit usually affects through dataflow. Knowing what fraction of all commits contributing code to a project are part of dataflow-based interactions can show how often new commits affect the data of a feature. Regarding this, it is worth considering that commits constituting code of a feature are very likely to influence said feature through dataflow, as dicussed in section 3.3. Since structural interactions coinciding with dataflow interactions are so obvious, programmers are also much more aware of them. Depending on the prevalence of feature regions in a project's code space, this could heavily skew the data in one direction, as a large portion of all dataflow interactions would stem from these obvious interactions. Therefore we especially focus on commits that affect features through dataflow, while not constituting code of these features. Logically, programmers are less aware that changes introduced with these commits might affect the data of seemingly unrelated features.

Dataflow-based CFIs allow us to examine another interesting property, namely the number of commits affecting a feature through dataflow. Here, we differentiate between commits INSIDE and OUTSIDE of features, i.e. commits that either do or don't contribute code to them. We examine the proportion of outside to inside commits influencing a feature to gauge where most dataflow interactions of a feature stem from. Considering our previous assumptions, the ratio of outside to inside commits could decrease with increasing size of a feature. Smaller features are likely to structurally interact with less commits than their larger counterparts, resulting in them also interacting with less inside commits through dataflow. However, the number of outside commits affecting data of a feature is not dependent on a feature's size in such a way. This leads us to another relationship worth investigating, namely the relation between the size of a feature and the number of outside and inside commits interacting with it through dataflow. We examine the two commit kinds separately, as we already have a strong supposition for inside commits, but are less certain for outside commits. Determining to what extent feature size is the driving factor in this relation, could tell us whether it is worth considering other possible properties of features in future analyses.

*RQ3: How do authors interact with features?*

A simple yet promising way to extract more information out of the collected CFIs is to link the interaction's commits to their respective authors. Thus, we are able to investigate how authors interact with features both structurally and through dataflow. Similarly to the number of commits implementing a feature, we can now calculate the number of authors that participated in its development. The same applies to commits interacting with features through dataflow, for which we can now determine the authors contributing these commits. Here, we only consider outside commits, i.e. commits not constituting code of the feature, since we have already considered all inside commits when investigating the developers implementing a feature. We are especially interested in how many authors exclusively interact with a feature through dataflow in comparison to the number of its developers. This lets us determine whether the developers that implement features are also the ones mainly responsible for changes affecting them through dataflow.
Finally, we relate both types of author interactions to the respective size of a feature. Regarding structural interactions, this allows us to determine whether a feature's extent in the source-code of a project is driving factor in the amount of authors needed to implement it. Software companies could use evidence on this issue as advice on how to allocate programmers on to-be implemented features. Findings on the investigated dataflow interactions of authors could tell developers that their changes might affect small and, at first sight neglegible, features surprisingly often. We also have a look at the specific functionality of features since their purpose could also play an important role in the number of authors interacting with it.

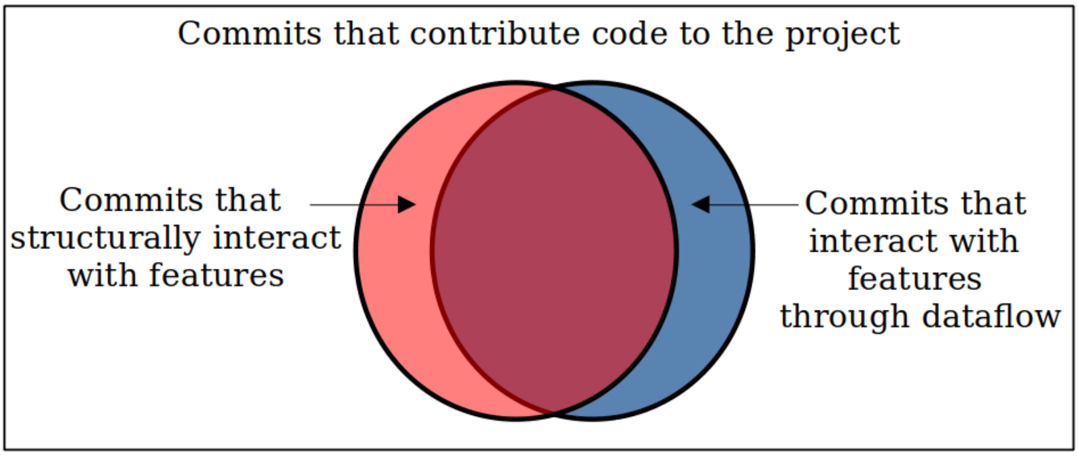


Figure 4.1: Illustration of different commit kinds

In the first two **RQs** we have discussed different kinds of commits and the ways in which they interact with features. The above figure showcases them in a venn diagram and illustrates the dependencies and divisions between them.

## 4.2 OPERATIONALIZATION

Here, we explain how the proposed RQs can be answered. The general experiment process is the same for all RQs. At first, we collect data comprimising all structural or dataflow-based

CFIs by creating reports of a specific type for a chosen software project. The collected data is then processed in order to gain information for each commit or feature in the project, such as the number of interacting commits and size of a feature. The processed data is used to calculate statistical information, such as the mean and variance, or the strength of a correlation. To facilitate a faster and better understanding of the processed data and calculated statistics, we display them graphically via distribution or regression plots. The projects we investigate in this work, for example XZ and GZIP, are of a small size and are used in a compression domain. This choice is based on the fact, that our research intends to only lay basic groundwork, where smaller projects can already offer a lot of insight. Since we only investigate a few projects, we chose them to be of a similar domain, such that a comparison between them makes more sense. In the following sections, we explain our method of investigation in more detail for each RQ.

***RQ1: How do commits and features structurally interact with each other?***

For this RQ, we examine reports comprising structural commit-feature interactions. From the collected data, we extract the size and the number of commits a feature structurally interacts with as well as the number of features a commit has structural interactions with. Concerning the first property of structural CFIs mentioned by us, we calculate the mean and variance of the number of commits used to implement a feature. This gives us an overview of how many commits were used during a feature's development and tells us how much this number varies between different features. A regression analysis on the relation between the size of a feature and the number of commits used to implement it, allows us to compute the strength of their correlation. From the commits structurally interacting with features, we calculate the average number of features that a commit changes. Using the calculated average, we can determine if commits are primarily used to change just one feature. Additionally, we determine what fraction of commits that structurally interact with features do so with more than one feature, allowing us to see how likely commits are to affect code of only a single or more features. Here, filtering outliers can help produce more sensible data, as commits responsible for refactoring could change many features, while not implementing any functionality.

***RQ2: How do commits interact with features through dataflow?***

an initial step of evaluating what fraction of commits affect features through dataflow, is first determining which set of commits we consider in our analysis
- logically, we should only consider commits that could potentially be part of a dataflow-based CFI
- the only prerequesite is for commits is to be represented by commit regions inside a program

The projects investigated for dataflow-based CFIs are the same projects as investigated for structural CFIs. This choice gives us more insight into a single project and allows us to combine both analysis results as will be discussed below. For this RQ, we consider all commits that currently contribute code to the repository, which we can extract from high level repository information of the project. We process the collected data, such that, for each commit, we save how many features they interact with through dataflow. Here, we also indicate whether a commit structurally interacts with a feature, which we can simply check by examining the according structural report. With this information, we are able to separate commits into ones that implement parts of a feature and those that do not. We have already dicussed that commits used to implement a feature are much more likely to account for dataflow interactions. We provide evidence for this by comparing the average amount of features the two types of commits affect through dataflow. Besides that we compute what fraction of commits have dataflow-based interactions with other features, once for all commits, once for commits that are part of a feature and once for commits that aren't part of a feature. This shows how often commits affect features through dataflow based on their intended purpose, e.g. actively changing the functionality of a feature or changing something seemingly unrelated to a feature. With the data collected for each commit, we also plot the distribution of how many features they interact with. This lets us recognize potential outliers, for example commits with much more dataflow interactions than the mean would suggest. To check whether feature size is the driving factor in the number of outside commits affecting a feature through dataflow, we compute how strongly the two are correlated. While we already know the size of each feature from RQ1, it's still neccesary to calculate, for each feature, which commits have dataflow-based interaction with them and subsequently filtering the commits that are part of a feature.

***RQ3: How do authors interact with features?***

Here, we also examine the same projects as in the previous RQs. That way we can reuse data produced in RQ1 to map each feature to the authors that implemented it. For RQ1 we have already mapped each feature to the commits it interacts with, i.e. that contribute code to it. It's possible to extract the authors of these commits by accessing high-level repository information. This directly gives us the authors that implemented a feature. With the processed data, we are able to calculate the average number of authors used to implement a feature and plot their distribution for a sound overview. The size of each feature has also been calculated to answer the previous RQs. With this information we aim to correlate the size of a feature with the amount of authors that implemented it in a regression analysis.

# 5 EVALUATION

This chapter evaluates the thesis core claims. - for this, we investigate four projects

## 5.1 RESULTS

### *RQ1: Evaluation of Structural CFIs*

*Patterns around Feature Development*

Aside from the number of commits used during its development, we can also extract the size of a feature from its structural CFIs. These are the main properties we consider when investigating patterns around feature development. Additionally, we factor in the nesting degree of structural CFIs during our analysis allowing us to extract more information from both properties. Figure 5.1 illustrates our results in three different plots for each project. Each row displays the results for one project with the name of the respective project being shown on the far left.

In the first column, we show the number of structurally interacting commits for each feature in a bar-plot. We differentiate between interactions with a nesting degree of 1, which are colored in blue, and ones with a higher nesting degree colored in orange. The orange bar represents commits for which we are less certain of their specific purpose, where they could have been used to implement its respective feature and others at the same time. Overall, we notice a large spread of structurally interacting commits between features across all projects. Most interactions occur at a nesting degree of 1, especially for features with many interacting commits in comparison to other features of their respective project. Interestingly, for features with few interacting commits, a large proportion of their interactions have a higher nesting degree than 1. We also notice that the range of interacting commits varies greatly between the different projects. In BZIP2, the highest number of interacting commits is 7 for the VERBOSITY feature, but at over 60 for the FORCE,NO_NAME feature in GZIP. The lowest number of interacting commits is 1 for all projects, except lrzip with 6, indicating that some features need very little work to be implemented. The projects XZ and BZIP2 bear some similarities, such as the VERBOSITY-feature having the highest number of interacting commits. Both XZ and BZIP2 also share a FORCEOVERWRITE-feature, which has an average number of interacting commits in both projects, with a large portion of interactions having a nesting degree higher than one.

In the second column, we display the different feature sizes for each project in a bar-plot. The y-ticks represent the number of instructions implementing a feature, i.e. the number of instructions stemming from its feature regions. The blue bar represents the definite size of a feature, where we only count instructions that are exclusively part of its respective feature. The orange bar adds the potential size of a feature on top, for which we also count instructions that are part of multiple feature regions at once. For these instructions there is less certainty whether they really implemented functionality of that specific feature. It also allows us to see how common feature nesting is inside a project and which features are nested inside other features. Upon examining the shown feature sizes, we notice a pattern similar to that of the first column. Here, larger features mostly consist of definite feature size, whereas smaller features have a high proportion of potential feature size. This is not the case for BZIP2, where all features have a high amount of potentially implementing instructions. Surprisingly, this is also true for features that only have commits structurally interacting with them at nesting degree one. The largest combined feature size we encounter is above 5000 for the feature FORCE,NO_NAME, which is also the feature with most interacting commits. For the projects XZ, BZIP2 and LRZIP the average feature size ranges between 200 and 400, while GZIP has an average around 1.200. This is the due to the very large feature we mentioned, which encompasses 4000 more instructions than the second largest feature.

In the third column of figure 5.1, the datasets used in the two previously dicussed plots are related inside two regression plots. Broadly speaking, we plot the size of a feature against the number of commits that structurally interact with it. For the regression plot colored in blue, we only consider the definite size of a feature and commits structurally interacting with them at a nesting degree of one. The values of the displayed scatter points are identical with the values of the blue bars in the two previous columns. In the second regression plot, we consider all structurally interacting commits and all instructions regardless of any nesting degree. As we now factor in the combined values of the blue and orange bars of the previous columns, the color of the plot is a combination of the two. The respective correlation coefficents and p-values are shown at the bottom-right of each graph. They are calculated with the pearson correlation coefficent, which measures the linear relationship between two datasets. Across all projects and both linear regressions, we note a clear positive correlation between the size of a feature and the number of structurally interacting commits. For both linear regressions, the correlation coefficients are quite high, ranging from ca. 0.9 for XZ to ca. 0.98 for LRZIP. BZIP2 makes for the only exception here, where the correlation coefficent drops from 0.89 for the broader to 0.59 for the more specific consideration. The p-values are very low for all pearson correlations ranging from close to zero for XZ to 0.16 for LRZIP indicating a high probality that both datasets are not un-correlated. Combined with the high correlation coefficients, this gives us high confidence that size of a feature and the number of commits used during its development are strongly positively correlated.

*Usage of Commits in Feature Development*

### *RQ2: Evaluation of Dataflow-based CFIs*

*Proportion and Dependencies of Commits Affecting Features through Dataflow*

The initial step in evaluating what fraction of commits affect features through dataflow, is determing the number of active commits. Active commits are represented by at least one commit region inside a program, meaning that they fullfil the minimum requirement to be part of a dataflow-based CFI. The respective values are shown in the first column of table 5.1 for each project. We determined xz to have the highest number of active commits at 1039, while BZIP2 only has a tiny fraction of that at 37. In the first plot of figure 5.2, the bars colored in red show what percenateg of commits interact with features through dataflow. We notice that the respective percentages are vastly different from project to project. The majority of GZIP's active commits are part of dataflow-based CFIs at 53.6%. This percentage gets halved for BZIP2 with about 27%, with another large drop to 11.3% for xz. This means that roughly every second commit in GZIP, every fourth commit in BZIP2 and every ninth commit in xz affects features through dataflow. The bars colored in grey display the fraction of active commits structurally interacting with features. In addition to the obvious fact of how often commits are used to implement features, this also gives us an estimation on the extend of feature-code in a project. The more commits are part of structural CFIs, the more of the code contributed by them and therefore the overall code of a project will be part of feature regions. Logically, the accuracy of this estimation depends on many factors, such as the extend to which commits contributing code to features, also contribute code to other parts of a program. Still, given the large disparity in the percentage of commits with structural interactions, we can be relatively certain that xz has the lowest proportion and GZIP the largest proportion of feature-code with BZIP2 somewhere inbetween. Comparing the two bar-types, we notice that the percentage of commits with structural interactions is lower than the percentage of commits with dataflow interactions for each project. This phenomenon can be explained considering the values presented in the second column of table 5.1. There, we show the probablity for a commit to be part on any dataflow-based CFI, given that said commit is part of any structural CFI. We see that the probabilities are roughly the same for each project at around 90%. This means that by only taking into account commits part of structural CFIs, we already encounter a lot of commits that affect features through dataflow. If we then consider the entire set of active commits, the number of commits with dataflow interactions will likely exceed the number of commits with structural interactions. Across all projects, the probability of a commit interacting with features through dataflow, given that it is part of structural CFIs, is much higher than the same probability for any active commit. This is a clear indication that our assumptions in section 3.3, that structural CFIs heavily coincide with dataflow-based CFIs, are correct. Since our intent is to especially focus on commits whose interactions with features can only be discovered by employing our dataflow analysis, we now aim to quantify these commits in the investigated projects. For this, we examine the relative difference between the percentages of commits with structural and commits with dataflow interactions. This difference roughly determines what share of commits are part of the discussed more interesting dataflow interactions. The difference is the smallest for BZIP2 at 10% and slightly higher than that for GZIP at 15%. xz has by far the biggest relative difference

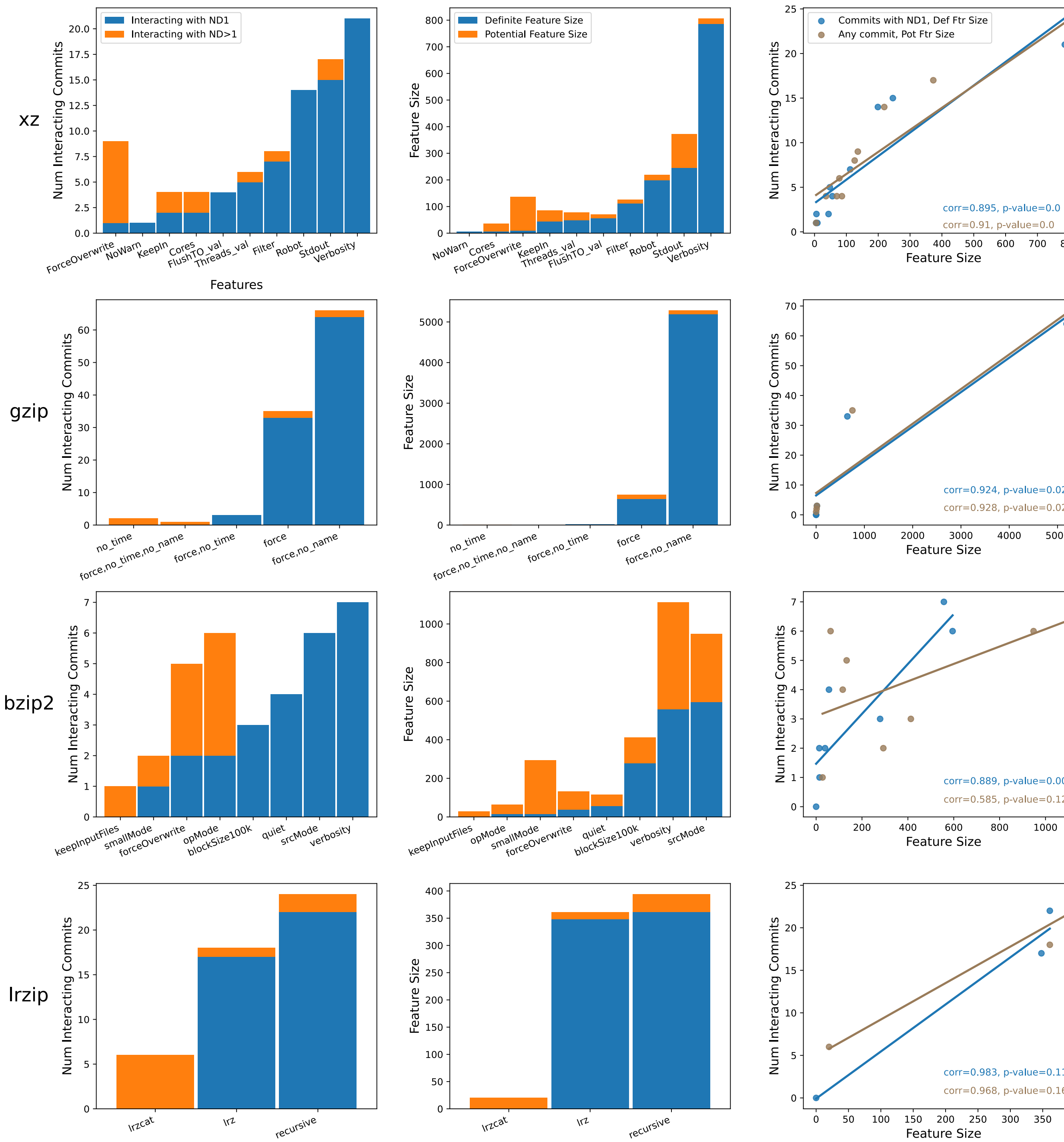


Figure 5.1: Feature Structural CFIs Plot

at almost 49%, which means that our dataflow analysis reveals many additional interactions here. We explore the topic of different dataflow interaction types more thoroughly in our second plot, which focuses on the origin of dataflow.
There, we have a closer look at the commits affecting features through dataflow and where their dataflow stems from. Naturally, dataflow occuring inside the regions of a feature is more intentional and therefore less interesting, than dataflow originating outside its regions. In section 3.3, we explain, that the dataflow of commits and features not structurally interacting with each other, must be outside dataflow. Even though the regions of a commit and regions of a feature must partially overlap in order for them to structurally interact, such a commit can still have regions not part of any feature regions. Therefore, we cannot be sure whether the dataflow of these interactions originates from outside or inside the regions of a feature. Due to the inherent properties of structural interactions, i.e. heavily coinciding with dataflow interactions, we assume their dataflow to originate inside the regions of a feature. In the second plot of figure 5.2, we separate the commits with dataflow interactions into three categories based on dataflow origin. The first category, shown as the bar colored in blue, represents commits that only interact with features through outside dataflow. They make up the majority of commits for xz at 56.4% and only 20% of commits for both BZIP2 and GZIP. The second category represents commits that interact with features through outside and inside dataflow. Logically, these commits affect at least two features through dataflow and only structurally interact with a subset of them. Surprsingly, they form the majority of commits for BZIP2 and GZIP at 60 and 51% respectively and around 20% of commits for xz. The proportion of commits that only interact with features through inside dataflow varies less between the projects. GZIP has the highest percentage at 28.8%, BZIP2 the lowest at 20% with xz inbetween the two at 23.9%.
We admit that the high proportions of commits with both inside and outside dataflow are rather unexpected. They hinder a clear separation of commits with dataflow interactions into less and more interesting commits, e.g. commits whose dataflow interactions are more and less obvious. We notice that commits interacting with features through inside dataflow, have a high chance to also interact with other features through outside dataflow. By definitionm, all commits interacting with features throhgh inside dataflow also structurally intercact with them. From previous explanations, we also know these commits are almost identical to the entire set of commits with structural interactions. It follows that commits part of structural CFIs must also have a high, albeit slightly lower, chance to affect features through outside df. We compare this probability to the probability of any active commit to interact with features through outside df in table 5.2. The strongest contrast occurs for xz, where 38% of commits structurally interacting with features, interact with other features through outside df, making them 4.5 times more likely to do so compared to any active commit of xz at 8.5%. The contrast is slightly weaker for BZIP2 with a 3.5 times higher likelihood and according percentages of 66.7 and 21.6% respectively. gzip has the weakest contrast at 2.2 and respective probabilites of 60.2 and 26.8%.

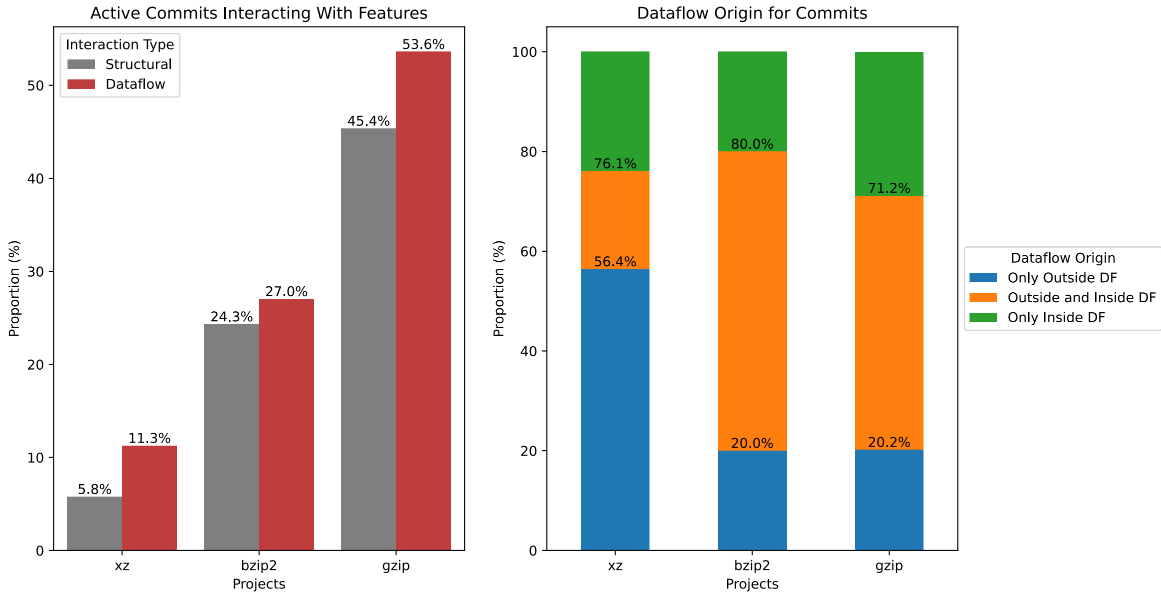


Figure 5.2: Proportional Dataflow-Plot for Commits

Table 5.1: Additional Information for Dataflow Analysis of Commits

| Projects | Number of Active Commits | Probability for Commit to be part of dataflow CFI given that it is part of structural CFI |
|---|---|---|
| xz | 1039 | 0.883 |
| gzip | 194 | 0.933 |
| bzip2 | 37 | 0.889 |

### *RQ3: Evaluation of Author Interactions*

## 5.2 DISCUSSION

In this section, discuss your results.

## 5.3 THREATS TO VALIDITY

In this section, discuss the threats to internal and external validity.

Table 5.2: Relating Inside Dataflow to Outside Dataflow

| Projects | Probability for Commits Part of Structural Interactions to Affect other Features Through Outside DF | Same Probability Given Any Active Commit |
|---|---|---|
| xz | 0.38 | 0.085 |
| gzip | 0.602 | 0.268 |
| bzip2 | 0.667 | 0.216 |

# 6 RELATED WORK

Interactions between features [2, 3] and interactions between commits [6] have already been used to answer many research questions surrounding software projects. However, investigating feature interactions has been around for a long time, while examining commit interactions is a more recent phenomenon.

In an article published in 2023, Sattler et al. [6] analysed several open-source projects with their novel approach, SEAL. SEAL merges low-level data-flow with high-level repository information in the form of commit interactions. The paper shows the importance of a combination of low-level program analysis and high-level repository mining techniques by discussing research problems that neither analysis can answer on its own. For example SEAL is able to detect commits that are central in the dependency structure of a program[6]. This was used to identify small commits affecting central code that would normally not be considered impactful to a program. Furthermore, they investigated author interactions at a dataflow level with the help of commit interactions. Thus, they can identify interactions between developers that cannot be detected by a purely syntactical approach. They found that, especially in smaller projects, there often exists one main developer authoring the majority of commits[6] and thus accounting for most author interactions logically. It was also explained how SEAL makes it possible to relate occurences of bad programming practices to developers. This is accomplished by SEAL enriching program analyses with computed repository information.

Lillack et al. [3] first implemented the automatic detection of features inside programs by tracking their load-time configuration options along program flow. In our research, we also focus on features who are configured via configuration options, which we call configuration variables. Their analysis tool Lotrack can detect which configuration options must be activated in order for certain code segments, implementing a feature's functionality, to be executed. They evaluated Lotrack on numerous real-world Android and Java applications and observed a high accuracy for the predicted code execution constraints[3].

Referencing this paper Kolesnikov et al. [2] published a case study on the relation of external and internal feature interactions. Internal feature interactions are control-flow feature interactions that can be detected through static program analysis as mentioned above. They concluded that considering internal feature interactions could potentially help predict external, performance feature interactions[2].

# 7 CONCLUDING REMARKS

## 7.1 CONCLUSION

## 7.2 FUTURE WORK

# A APPENDIX

This is the Appendix. Add further sections for your appendices here.

# BIBLIOGRAPHY


[1] Christopher Hundhausen, Adam Carter, Phillip Conrad, Ahsun Tariq, and Olusola Adesope. "Evaluating Commit, Issue and Product Quality in Team Software Development Projects." In: SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 108–114. ISBN: 9781450380621. DOI: 10.1145/3408877.3432362. URL: https://doi.org/10.1145/3408877.3432362.

[2] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. "On the relation of external and internal feature interactions: A case study." In: *arXiv preprint arXiv:1712.07440* (2017).

[3] Max Lillack, Christian Kästner, and Eric Bodden. "Tracking load-time configuration options." In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 445–456.

[4] Florian Sattler. "Understanding Variability in Space and Time." To appear. dissertation. Saarland University, 2023.

[5] Florian Sattler. *VaRA is an analysis framework that enables users to build static and dynamic analyses for analyzing high-level concepts using advanced compiler and analysis technology in the background.* https://vara.readthedocs.io/en/vara-dev/ [Accessed: (24.05.2023)]. 2023.

[6] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. *SEAL: Integrating Program Analysis and Repository Mining*. 2023.