Bachelor's Thesis

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

SIMON STEUER

November 2, 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

LIST OF FIGURES

LIST OF TABLES

LISTINGS

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 them. 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 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 7. 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.

```
if (CheckZero) {                ▷ b9ea2f3    ▷ CheckZero
   if (x == 0) {                ▷ b9ea2f3    ▷ CheckZero
      throw Error();            ▷ b9ea2f3    ▷ CheckZero
   }                            ▷ b9ea2f3    ▷ CheckZero
}                               ▷ b9ea2f3    ▷ CheckZero
...
...
...
if (Min && CheckZero) {         ▷ e49c7a1    ▷ Min,CheckZero
   x = min(x, y);               ▷ e49c7a1    ▷ Min,CheckZero
   if (x == 0) {                ▷ b9ea2f3    ▷ Min,CheckZero
      throw Error();            ▷ b9ea2f3    ▷ Min,CheckZero
   }                            ▷ b9ea2f3    ▷ Min,CheckZero
}                               ▷ 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.

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

With the used defintion, we face the issue that not every considered instruction definetly implements the feature. We therefore introduce the concept of a definite feature size, in which we only count instructions where the feature's regions appear exclusively. These are the same instructions that we consider for the normal size of a feature and which only have a nesting degree of one. For them, we are more certain that they have implemented the feature specifically and not another feature whose regions they are also part of.

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. To accommodate later evaluations using nesting degrees, we create reports featuring a more complex datatype than a sole CFI. For this, we specifically save which features a commit structurally interacts with at the same time, i.e. inside the same instructions. We know that an instruction is always part of exactly one commit region, but could possibly belong to any number of feature regions. According to definition 5, we encounter a structural CFI, if an instruction belongs to at least one feature region. For every such instruction, we store the discussed interaction, as the commit and the features present in the instruction. For each interaction, 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. By creating complex structural reports, we can not only determine the nesting degree of each structural CFI later on, but also the nesting degrees of the instructions they occur in. This allows us to calculate the definite size of a feature, additionally to its normal size defined in definition 8. We can compute the size of a feature as the sum over the instruction counters of all found interactions the feature is part of. To calculate a feature's definite size, we only consider interactions where the feature appears exclusively. These interactions stem from instructions with a nesting degree of one, which means that they are part of no further feature regions.

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.

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.

Investigating Patterns around Feature Development

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 3.4 of our collected structural CFIs. This allows us to additionally determine the definite size of a feature showing us to what extent its code is nested inside other features. Furthermore, we can differentiate between commits that most likely and those that only potentially implemented a feature's functionality. By doing so, we intend to achieve a more accurate analysis regarding the number of commits used during a feature's development.

Examining the Usage of Commits in Feature Development

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 a feature. In our analysis, we should ideally filter commits whose purpose was refactoring code, since they do not fit our criteria of changing or adding functionality to a feature. Exceptionally large commits might be best considered for this, as this is where we expect most commits used for large-scale refactoring to appear. A qualitative review of these commits is still necessary to make a final decision on whether they should be filtered.

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.

The Proportion and Dependencies of Commits Affecting Features through Dataflow

Firstly, we investigate how connected commits and features are by analyzing the number of features a commit 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 the dependency dicussed in section 3.3, that structural interactions heavily coincide with dataflow interactions. This implies that commits constituting code of a feature are very likely to influence said feature through dataflow as well. In section 3.3, we also mention that the dataflow of commits not structurally interacting with a feature, must stem from outside the regions of the feature. Naturally, said dataflow is less intentional and subsequently more interesting, than dataflow occuring inside the regions of a feature. Programmers are less aware that changes introduced with these commits might affect the data of seemingly unrelated features. Therefore, we intend to differentiate between dataflow interactions with an outside and those with an inside origin in our anaylsis. This allows us to especially focus on commits part of outside dataflow interactions and examine their proportion among all commits.

Understanding Features and the Commits Affecting them Through Dataflow

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 feature 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 them.

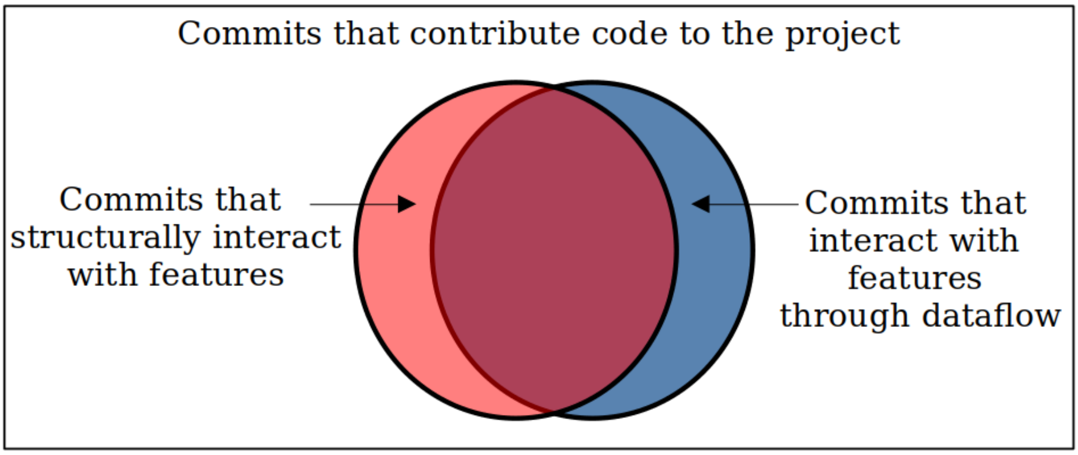

Figure 4.1: Illustration of different commit kinds

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 bar 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 the interactions contained in the structural reports created with our implementation in VaRA. As there could be vast differences between the projects, we also work with and present their respective data separately.

Investigating Patterns around Feature Development

From the collected structural reports, we first extract the number of structurally interacting commits and size for each feature. For each interacting commit, we also determine the nesting degree of its structural interaction with the feature. Additionally to the normal size of a feature, which we call potential size here, we calculate its definite size as well. The datapoints of the two properties are shown in two respective bar plots, where the values for each feature are shown in a separate bar. The bars are labelled after the name of the feature they present, to allow for comparisons between the two plots and between different projects. For comparisons between the investigated projects, we also calculate the average number of interacting commits and size of a feature for each project. In the plot displaying the number of interacting commits, we stack the commits according to the nesting degree of their respective interaction with the feature. The first part of each bar consists of commits interacting at a nesting degree of one, the second part of commits interacting at a higher nesting degree. We do the same in the second plot, where the definite size forms the base of each bar upon which the difference with the potential size is added. This also allows us to see how common feature nesting is inside a project and which features are nested inside other features. Logically, the stacked bars must be colored differently to see their respective distribution inside a feature. The y-axis represents the number of instructions implementing a feature, i.e. the number of instructions stemming from its feature regions. The definite feature-size only consists of instructions that are exclusively part of its respective regions, whereas we also count instructions part of mutltiple regions for the potential size of a feature. The features are sorted in an increasing order according to the number of commits interacting at nesting degree one or their definite size. Finally, we relate the two investigated properties in a SEABORN regression plot. Here, we show whether features increasing in size, in turn structurally interact with more commits, i.e. need more commits to be implemented. Therefore, the value of the x-axis shows the size of a feature, whereas the y-axis shows the number of interacting commits. Each feature is represented by a scatter point inside the plot combining its respective y-values from the two previous plots. A linear regression line is drawn in the plot matching the occuring scatter points, where a rising graph could already indicate a positive correlation. To check whether we are dealing with a statistically significant correlation, we compute the pearson correlation coefficient and its p-value in the STATS submodule of the SCIPY python namespace. To confirm our initial suspection of a strong positive correlation, the according correlation coefficient must be close to one, while the p-value must be close to zero falling below our rejection interval of 95%. Additionally to considering a dataset with the total number of interacting commits and size of a feature, we also consider a more restrictive dataset. For this, we only relate the number of commits interacting at a nesting degree of one with the definite size of a feature. It might be interesting to see whether only considering structural interactions and instructions with a nesting degree of one, could produce a stronger correlation.

Examining the Usage of Commits in Feature Development

For each commit part of at least one structural interaction, we determine the number of features it structurally interacts with. There, we also save at which nesting degree the commit and feature happen to interact with each other. We ignore this property at first, although we will consider it in a later analysis step. At first, we give a comprehensive overview of our initial results in a SEABORN histplot for every investigated project. The x-axis of the histogram shows the number of features a commit structurally interacts with, whereas the y-axis shows the number of commis for which the respective x-value matches. If 50 commits structurally interact with a single feature inside a project, the y-value of one will be 50 accordingly. Thus, we can quickly see to what extent our derived best practice surrounding commits in feature development is enforced in a project. The more commits are located at one in comparison to other x-values, the more commits only change a single feature. The histogram also shows us potential outlier commits, that structurally interact with a unusually many features. To facilitate comparisons between projects, we choose the same x- and y-labels for every histogram.

In a second analysis step, we intend to further specify the broad overview shown in initial the plots. As a baseline for the number of features a commit usually changes, we compute the average number of features a commit structurally interacts with in each project. Following this, we now disregard structural interactions with a nesting degree of more than one. Thus, we only consider features for which we are relatively certain that the commit was used to implement or change their functionality. Commits that exclusively interact with features at a high nesting degree, are now considered to change a single feature, as they could not be considered otherwise. Furthermore, we filter exceptionally large commits, whose purpose we expect to be something else than implementing features, such as code-refactoring. Similiarly to the regions of a commit, we determine its size in a repository as the number of source-code lines that were last changed or added by it. We then apply tukey's fence with a k-value of 3 to filter far outliers of commit sizes among commits interacting with features within their respective project. Afterwards, we perform a qualitative review of the filtered commits to check whether our initial suspicion about their purpose is correct. For both restrictions, we compute the average number of features changed, which should be lower than the calculated average without any restrictions. Lastly, we combine both restrictions to form a final lower bound for the number of features a commit usually changes. We show the four averages computed for each project in a cross-project table. By no means do we expect the actual average to be exactly at the final lower bound, but rather somewhere inbetween the final lower bound and the average with no restrictions. That is because, we do not expect every structural interaction of a high nesting degree to be justifiably disregarded.

RQ2: How do commits interact with features through dataflow?

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.

Analyzing the Proportion and Dependencies of Commits Affecting Features through Dataflow

We do not consider LRZIP here, as we are only able to detect regions of three out of 10 features in the project. This means that we can only find a fraction of all commits interacting with features, resulting in significantly lower determined proportions.

An initial step of evaluating what fraction of commits affect features through dataflow, is determining which set of commits to consider in the first place. Logically, we should only consider commits that could potentially be part of a dataflow-based CFI. The only prerequesite for this is that commits are represented by commit regions inside a program. This is the case when there exists at least one source-code line that was last changed or added by them. We call the respective commits, active commits as their contributed code is still "active" in the repository. Following the calculation of the number of active commits in the projects, we begin examining the created dataflow reports. For each commit part of at least one dataflow-based CFI, we save the set of features it interacts with. By dividing the number of these commits by the number of all active commits, we can determine their proportion within their respective project. To allow for an overview of all projects, we show the calculated percentages in a seaborn bar plot. To provide evidence for our claim that structural CFIs heavily coincide with dataflow-based CFIs, we now compute the proportion of commits with dataflow interactions among the commits with structural interactions. Given that the latter is significantly higher than the overall proportion of commits with dataflow interactions, our initial notion will be confirmed. Alongside the percentage of commits part of dataflow-based CFIs, we also show the percentage of active commits that are part of structural CFIs. As we expect most of the latter commits to be part of dataflow-based CFIs as well, the difference between their respective percentages indicates what proportion of commits interact with features exclusively through dataflow. The percentage of commits with dataflow interactions is certainly higher than the percentage of commits with structural commits, as the latter commits largely contribute to the former, while this does not need to be the case vice versa. Following the broad overview of commits affecting features through dataflow, we intend to specify and differentiate between the origin of said dataflow. For each feature a commit interacts with through dataflow, we check whether they also structurally interact with each other. If they do structurally interact, we speak of an inside dataflow origin, otherwise we speak of an outside origin. Afterwards we can split the commits with dataflow interactions into three categories. Next to commits that only affect features either through inside or through outside dataflow, a commit can also interact with multiple features, once through inside and once through outside dataflow. For each project, we calculate the respective percentages of each category and present them in a stacked bar plot. Logically, each bar representing a project adds up to 100% as every commit falls into exactly one category. The commits falling into categories of at least partial outside dataflow origins are part of interactions, which we could only detect with our dataflow analysis. Both bar plots are displayed next to each other in the same figure, where the sequence in which the projects are shown in both plots is determined by the proportion of commits with dataflow interactions. This allows for an easier comparison within a project, as its respective bars will have the same position in both plots. Statistical values, including the number of active commits and the probability for a commit to be part of a dataflow-based CFI given that it structurally interacts with features, are not included in the plots. As they can aid the understanding of the plots and add important context to them, we display the respective values in a LATEX table.

Exploring Features and the Commits Affecting them Through Dataflow

- intially, we determine the set of commits each feature interacts with through dataflow
- the according information is extracted from the same dataflow reports used in the previous section of this RQ
- depending on whether the commit also structurally interacts with a feature, we split the set of commits for each feature into two categories
- outside commits are fully located outside the regions of a feature, meaning that their data

flows into a feature
- inside commits are at least partially located inside a feature making it very likely that their dataflow occurs within the regions of a feature
- naturally, said dataflow affecting a feature is less interesting than dataflow stemming from seemingly unrelated commits
- to gauge what commit kind makes up the majority of commits affecting features through dataflow, we present both values for each feature in a SEABORN bar-plot
- for every examined project, we display the number of outside and inside commits with separate bars for the different features
- to enable an exact comparison, we also calculate the median for both outside and inside commits of a project
- we choose the median and not the mean, as we are especially interested to detect a general trend across the features of a project and intend to avoid outliers skewing the data in one direction
- besides that features with an exceptionally large number of interacting commits or higher number of commits of a certain kind are already shown in the bar plots
- the features are explicitly named to allow for cross-project comparisons of certain types
- in a second analysis step, we now relate the size of a feature to the number of commits affecting it through dataflow
- here, we are interested in determining whether and to what extent there exists a positive linear relationship between the two
- we employ two linear regression analyses, relating the size of a feature once to its outside and once to its inside commits
- similarly to our investigation in **RQ1**, the results are displayed in a SEABORN reg plot for each project
- with each regression plot, we also show the two computed pearson correlation coefficients and their respective p-values
- we decide for a 97.5% rejection interval again, meaning that the p-values must be lower than 0.025 to provide enough evidence that the two datasets are indeed not un-correlated
- furthermore, the correlation coefficient must range close to 1 in order to warrant a strong positive correlation
- lastly, we investigate our claim that smaller features have a higher proportion of outside to inside commits affecting it through dataflow
- for this, we examine the relation between the size of a feature and the ratio of its outside to inside commits
- again, we create linear regression plots for each project and calculate the acccording pearson correlation coefficients and the respective p-values
- potentially, we could also perform other regression analyses, for example a logarithmic regression, as they could fit our datapoints better
- following our explanations in section x, we expect the ratio of outside to inside commits to decrease with increasing feature size
- this would be shown by a negative linear correlation coefficient and a falling linear regression line
-

RQ3: How do authors interact with features?

- technically, we perform the same analysis regarding features as discussed for the two previous RQs
- the only difference is that we replace the commits interacting with a feature with their respective authors
- thus, we can determine the authors contributing commits that structurally interact with features and those contributing commits affecting features through dataflow
- to faciliate expressing this, we simply say that the respective authors interact with features based on a certain interaction type
- in section 4.1, we mentioned that we only consider commits outside of features when examining the authors interacting with them through dataflow
- their computation has already been explained in the operationalization of RQ2, which we reuse for this RQ
- the number of authors that interact with a feature either structurally or through dataflow are presented in a SEABORN bar plot
- similarly to the previous RQs, the results of each project are shown in separate plots with labeled ticks for each feature
- we also calculate the mean number of dataflow and structural authors of a project to give concrete information from what type of interaction most author interactions stem from
- in a third bar for each feature, we display the number of authors that exclusively interact with them through dataflow
- there, we exclude authors who also structurally interact with the feature, i.e. the authors making up the first bar
- this shows us how many authors affecting a feature can only be determined via our dataflow analysis
- for the structural interactions of authors, we compute what fraction of them also affect the respective features through dataflow
- this can tell us to what extent the developers implementing a feature are likely to introduce the code changes whose data the feature uses later on
- additionally considering whether there are more structural than dataflow authors lets us determine whether the developers of a feature are also the ones mainly responsible for changes affecting them through dataflow.

5 EVALUATION

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

5.1 RESULTS

RQ1: Evaluation of Structural CFIs

Patterns of 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. There is a wide distribution of their respective sizes among the features of a project. Many consist of close to 0 and up to 150 instructions, which only is a small fraction of the size of the largest 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 ofOnly 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 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 rises from 0.59 for the broader to 0.89 for the more specific consideration. The p-values of XZ, GZIP2 and the more restrictive dataset of BZIP2 are all lower than 0.025. Thus, they lay outside of our rejection interval of 95% indicating a high probality that both datasets are not un-correlated. Combined with the high correlation coefficients, this gives us high confidence that the size of a feature and the number of commits used during its development are strongly positively correlated. The fact that LRZIP only has three datapoints, means that the p-values of their correlation coefficient are also relatively high at over 0.1. The broader consideration of BZIP2 also encompasses values that do not produce enough statistical evidence to support the notion that the two related datasets are correlated.

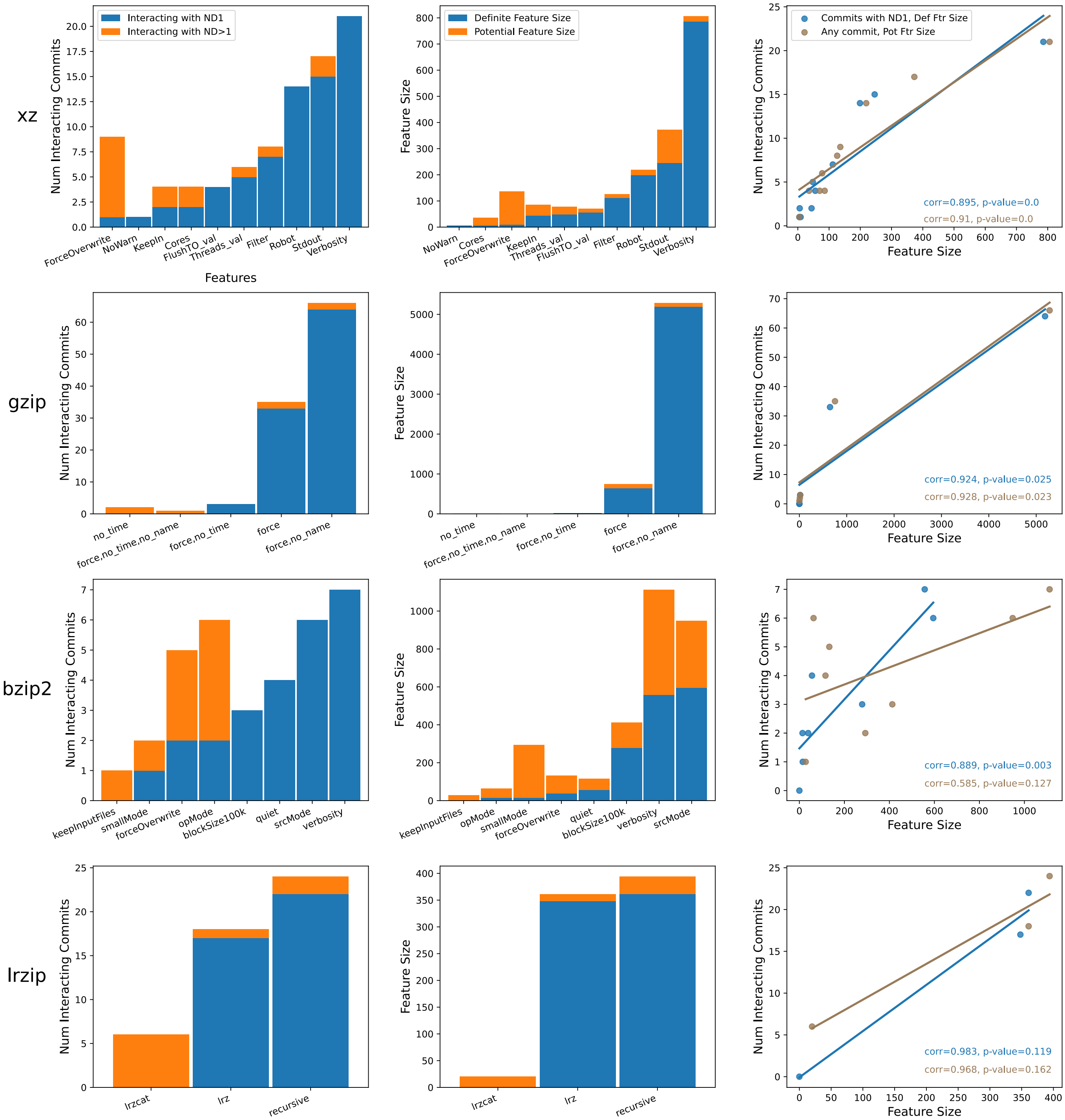

Figure 5.1: Feature Structural CFIs Plot

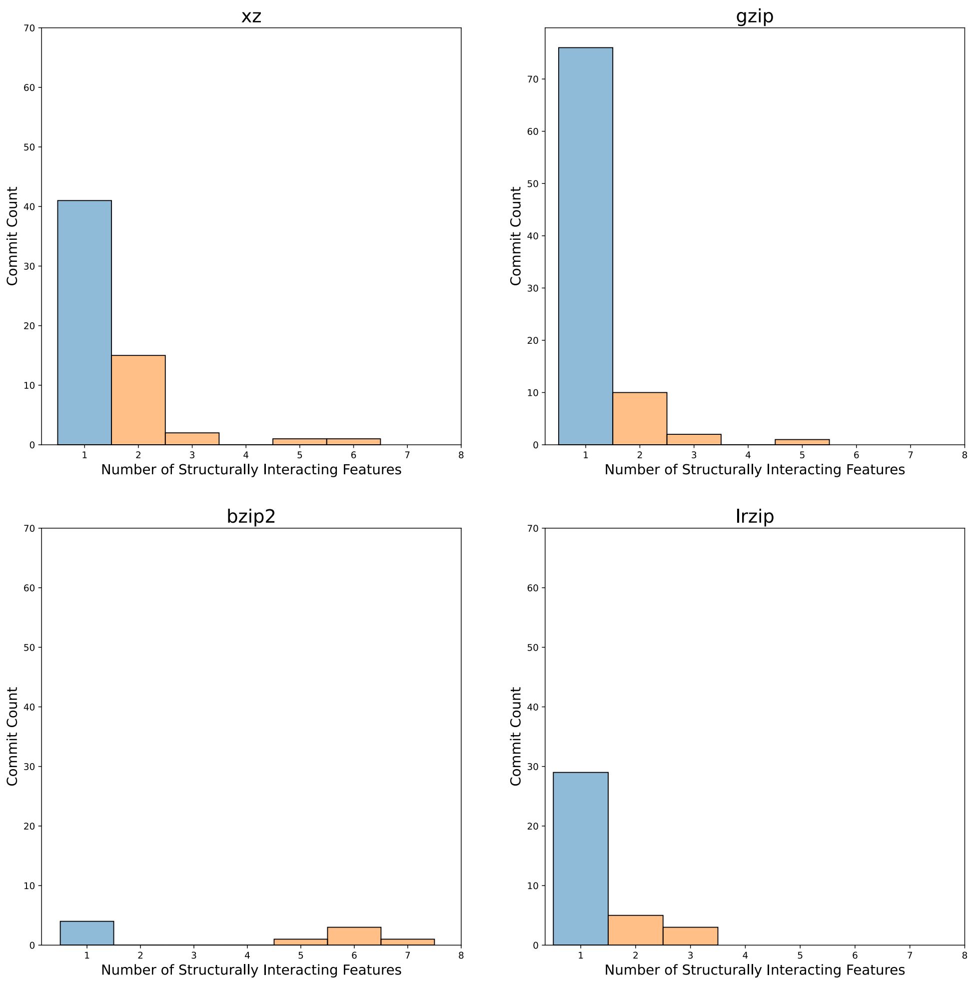

Figure 5.2: Commit Structural CFIs Plot

Usage of Commits in Feature Development

When proposing RQ1, we derived a best practice of commits used in feature development, namely that they should mostly change a single feature. We investigate to what extent this is enforced in our investigated software projects by examining the features a commit structurally interacts with, since these are the features a commit changes or implements. The distribution of how many commits interact with a certain number of features is displayed in figure 5.2. xz, GZIP and LRZIP have similar distributions, which is why we discuss them now and BZIP2 later. For xz, GZIP and LRZIP, we note that the majority of commits structurally interact with a single feature. GZIP has the strongest contrast, with over 70 commits interacting with only one feature and just 13 commits interacting with multiple features. Among the three mentioned projects, xz has the weakest contrast with around one third of commits interacting with more than one feature. The commit count gradually drops with an increasing number of structurally interacting features until it subsides to 0 at an x-value of 4. Both xz and GZIP have some outlier commits interacting 5 and 6 features. Surprisingly, the outlier commit 30ba4a2b of GZIP only changes 47 source-code lines being used to fix ”theoretical races in signal handling”. One

Table 5.1: Additional Information for Structural Analysis of Commits

Projects	Average	Only SCFI with ND1	Filter Large Commits	Both Restrictions
xz	1.47	1.23	1.3	1.11
gzip	1.2	1.14	1.18	1.1
bzip2	3.78	2.78	3.38	2.38
lrzip	1.3	1.08	1.3	1.09

outlier commits of xz imported the initial project to git and encompassed over 40000 additions. The other commit had many different responsibilites changing 118 files and over 3000 lines of code. Moving on to BZIP2, we notice a vastly different distribution compared to the previously discussed projects Overall there are much less commits used in feature-development with only 9 commits part of structural CFIs. The distribution consists of two clusters, the first being at one interacting feature and the second being centered around six interacting features. The second cluster encompassing 5 commits is slightly bigger than the first cluster with a commit count of 4. Since we are only dealing with a few commits here, that still produce quite interesting and unique data, we have a closer look at the purpose of these commits and relate it with the number of features they structurally interact with. We find that all commits part of the second cluster were used to publish new versions of BZIP2, always encompassing over 1000 additions. Three out of the four commits part of the first cluster introduced minor changes and bug fixes, never changing more than 30 source-code lines. The remaining commit updated BZIP2 to version 1.0.4 fixing minor bugs from the previous version with many of the added source-code lines being used for comments.

To specify our initial observations, we show the average number of features a commit structurally interacts with in table 5.1. We apply two restrictions, where we exclude certain structural CFIs and commits, allowing us to form a lower bound for the number of features a commit usually changes. Logically, we only consider commits with at least one interaction as we have done in figure 5.2. The first column shows us that the averages for XZ, GZIP and LRZIP, are slightly above 1 ranging from 1.2 to 1.47. The average for BZIP2 represents the only outlier at 3.78, which is to be expected given its discussed distribution. In the column to the right, we compute the same average, but filter structural CFIs with higher nesting degrees than one beforehand. Across all projects, this ensures a decrease of the averages with the lowest value now being 1.08 for LRZIP and the highest being 2.78 for BZIP2. Here, it's important to clarify that we do not claim every CFI to be rightfully filtered or not filtered. Following our explanations in section 3.4, we do expect to exclude numerous CFIs whose commits did not implement the interaction's feature specifically. In a second restriction, we exclude commits we determined to be exceptionally large among the commits implementing features within their respective project. Except for LRZIP, this measure also ensures a slight, albeit less significant, decrease of the averages. We performed a qualitative analysis of the filtered commits to check whether our suspicion, that they were mostly used to refactor code, is true. We find that many of them were used to import the project or newer versions to git. The fact that such commits exist is unsurprising, considering that the investigated tools have been published before the release of git or before git was widespread. While their purpose might not be refactoring code, it does make sense to filter these commits, as they are not accurately depicting the development process of the project itself or of its features. Only two out of the eighteen commits we filtered in all projects, were actually used to refactor code, specifically moving contents between files or adapting the code to a newer version of C. We admit that for half of all exceptionally large commits we found no reason, neither refactoring code or importing the project to git, to be left out of our analysis. Finally, we apply both restrictions at the same time to form a lower bound for the number of features a commit usually changes. For XZ, GZIP and LRZIP this final average centers around 1.1, which is certainly closer to 1 than the general average. BZIP2 still has the highest average at 2.38, although we note the biggest decrease for it from an initial value of 3.78.

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 5.2 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 5.3, 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 5.2. 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,

Table 5.2: 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

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 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 5.3, 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

Table 5.3: Relating Inside Dataflow to Outside Dataflow

Projects	Probability for Commits Part of Structural CFIs to Affect other Features Through Outside DF	Same Probability Given Any Active Commit
xz	0.417	0.086
gzip	0.596	0.381
bzip2	0.667	0.216

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 definition, all commits interacting with features through 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 dataflow. We compare this probability to the probability of any active commit to interact with features through outside df in table 5.3. The strongest contrast occurs for xz, where 41.7% of commits structurally interacting with features, interact with other features through outside df, making them 4.8 times more likely to do so compared to any active commit of xz at 8.6%. The contrast is slightly weaker for BZIP2 with a 3.1 times higher likelihood and according percentages of 66.7 and 21.6% respectively. gzip has the weakest contrast at 1.58 and respective probabilites of 59.6 and 38.1%.

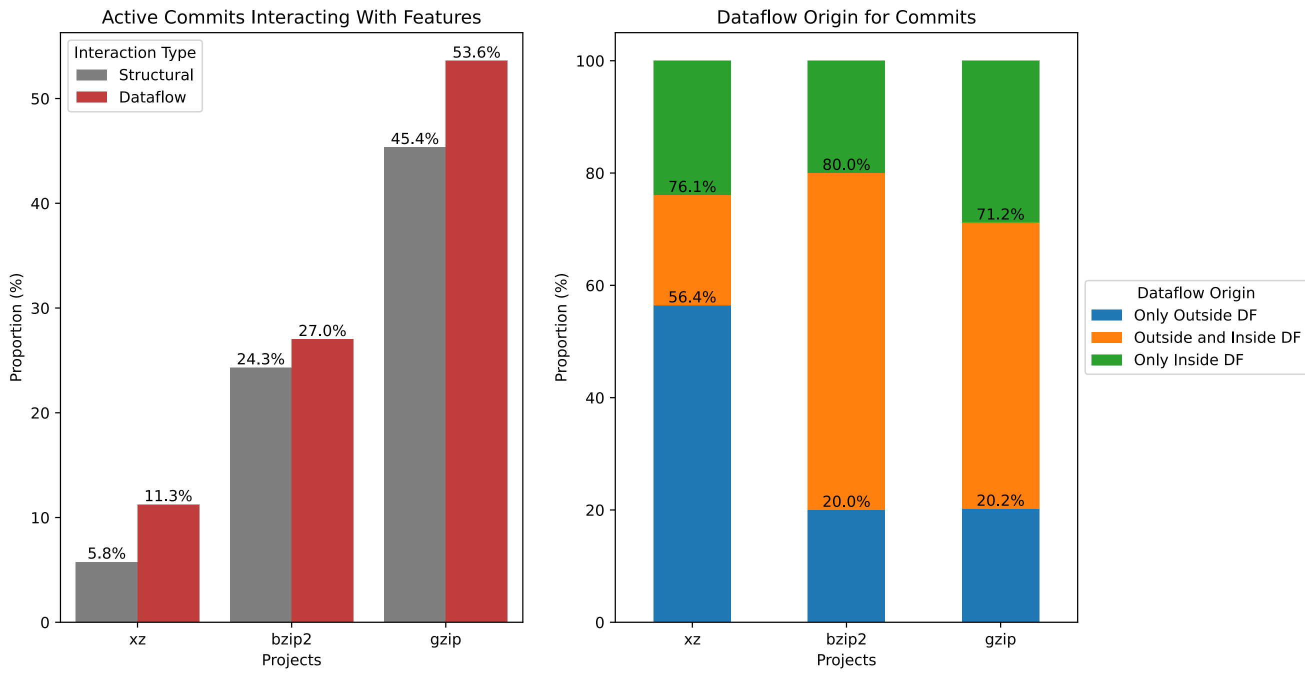

Figure 5.3: Proportional Dataflow-Plot for Commits

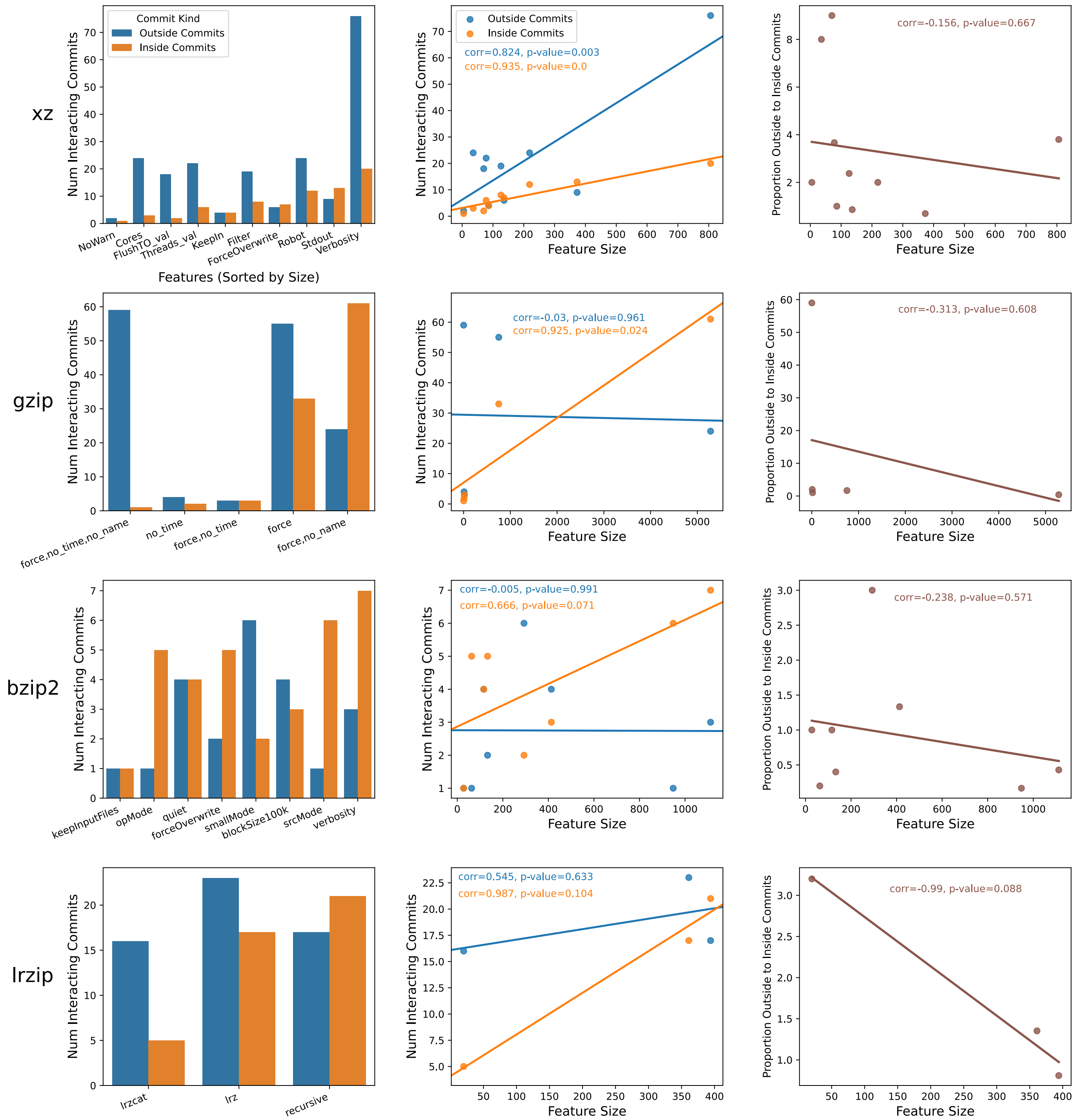

Figure 5.4: Proportional Dataflow-Plot for Features

Understanding Features and the Commits Affecting them Through Dataflow

- 1st plot

- data is rather unique for each project - describe why - median of outside commits compared to median inside commits - compare between projects - gzip has very interesting feature force,no$_time, no_namewithasingleinsidecommitandalmost60outsidecommits - explainwhy$

- 2nd plot

- 3rd plot

RQ3: Evaluation of Author Interactions

5.2 DISCUSSION

RQ1: Discussion of Structural CFIs

BZIP2 has shown to be a project whose results need to be treated carefully. Only 9 commits structurally interact with its features, encompassing much less datapoints compared to the other investigated projects. 6 out of 9 commits were used to import the initial project to git and iteratively update the code to newer versions. These commits are not exemplary of the common development process of a project and its features in a git repository. The actual development of the tool in the git repository began after BZIP2 was finally updated to version 1.0.6. The remaining 3 commits introduce minor changes and small bug fixes to the respective features they interact with. This means that BZIP2's features were largely implemented already when the actual development within the repository started. Thus, the number of features a commit changes on average as well as the commits structurally interacting with features are not good representatives of other projects developed in git.

Patterns of Feature Development

When evaluating feature sizes, we noticed that smaller features are often partially or completely nested inside other features. A high portion of their size stems from what we call potential size, i.e. instructions that are part of different feature-regions as well. In future work, it might be interesting to see whether this is a general phenonemon or a result of our project choice and their respective domain. For this, one could test to what extent there exists a correlation between the size of a feature and the relation of potential to definite feature size. According to our observations, the proportion of potential size decreases with an overall increase in the size of a feature. This phenonemon also bears consequences on they way commits are used in feature development and the CFIs resulting from this. If a developer wants to specifically change a feature nested inside other features, the according commits will necessarily produce structural CFIs with several features. In this context, it might be worth investigating whether such a nested feature implements separate, combined or additional functionality from the feature it is nested inside. This could help us decide whether a respective structural CFI implies that the commit actually implemented functionality of the feature. At least for partially nested features, we have shown a method that can help us better predict the purpose of a commit structurally interacting with it. If the commit also changes instructions that exclusively belong to a respective region of the feature, the according structural CFI will have a nesting degree of one. With this, we can now predict that the commit was used to specifically change the feature with more definity The fact that this method has some impact on our generated data can especially seen for features encompassing some potential feature size, but no commits interacting with it at nesting degree higher than one.
We employed the pearson correlation coefficient to investigate the linear relationship between the size of a feature and the number of commits structurally interacting with it. While we found high correlation coefficients of 0.9 and above, the p-values were often not low enough to justify the data to be statistically significant. We know that p-values are highly dependent on the number of datapoints, whereas the datapoints of our linear regression are the respective features of a project. Logically, projects with fewer features have less datapoints

resulting in their p-values being quite high and falling into our rejection interval. With 10 features, xz has by far the most datapoints and a respective p-value of >0.001 giving us most significant evidence for a strong positive correlation. Therefore, future research should prefer projects with at least 10 features so that correlations and other measurments produce trustful statistical data. Another possibility to circumvent the problem of too few datapoints could be to combine features of all projects into one big dataset. Here, we are faced with the issue that the way commits are used among projects might differ drastically. In our case, the number of instructions per structurally interacting commit of a feature varies greatly from project to project. The respective values are 17 for xz, 23 for GZIP, 87 for BZIP2 and 13 for LRZIP. This means that features of a similar size have vastly different numbers of structurally interacting commits on average. Upon these considerations, we are of the opinion that it makes more sense to test correlations within projects as we have done in this work.

Usage of Commits in Feature Development

In the previous section, we mentioned that the number of instructions per interacting commit of a feature varies between projects. Since instructions stem from source-code lines, their respective amount can in turn indicate the number of lines they stem from. Thus, we can derive that the lines of code a commit usually changes or contributes to a feature likely differs between projects as well. Disregarding BZIP2, the number of instructions per commit range from 13 to 23. Given that a single source-code line generally produces several llvm-IR instructions, we can predict the number of lines per commit to be below 10. This is a surprsingly low amount indicating that a commit often only introduces small changes to a feature. Of course, this is just a rough, and perhaps inaccurate, prediction, but it shows that our generated data can be used for many applications.
When determining a lower bound for the number of features a commit usually changes, we filtered structural CFIs with a higher nesting degree than one. Performing a qualitative analysis of the filtered CFIs could tell us whether this measure is justified. For this, one would have to check whether the commit was really not used to implement or change functionality of the interaction's feature. Our qualitive analysis of the excluded expectionally large commits has shown that while many commits were rightfully disregarded, there were still some for which this wasn't the case. However, performing the same verification is definitely more challenging for structural CFIs. Firstly, we are dealing with more than twice as many filtered structural CFIs than excluded commits. Secondly, the decision to exclude a commit can mostly be based of its commit message, as it already tells us its general purpose. The same cannot be done for a structural CFI, since it is plausible that the changed feature is not explicitly mentioned in the commit message. We would rather have to look into the changed source-code lines and decide whether the commit implemented the feature based on that. Without a thourough knowledge of the project and its respective features, this will take much longer to accomplish and will be error-prone as well. We therefore refrain from carrying out such an analysis of the filtered structural CFIs in this work.

5.3 THREATS TO VALIDITY

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

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.