Bachelor's Thesis

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

SIMON STEUER

November 10, 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 3.7, 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 CONCERNS OF A COMMIT

Often, multiple features are implemented in the same code space. This can occur due to various reasons, for example being a sub-feature of an overarching feature or depending on data generated by other features. Logically, the instructions stemming from the respective code are part of several feature regions. At the same time, only a subset of these instructions actually implement functionality of a specific feature. We say that these features are, at least partially, nested features. If a developer introduces changes to such a nested feature, the according commit will also structurally interact with other nested features. Now, we are faced with the issue that, while the commit structurally interacts with several features, it only changes functionality of one. In this case, it's impossible to tell which features the commit really changed just from its structural interactions alone. Therefore, we say that the commit only deals with a single concern as long as the features it interacts with are part of the same code space. In this work, we are working with instructions and therefore speak of the same code space of features when the instructions we are dealing with belong to the same set of feature regions. It follows that we can assign a commit a number of concerns according to its structural CFIs and the instructions they occur in:

Definition 7. The *concerns* of a commit are the different sets of features the commit structurally interacts with at the same time, i.e. in the same instructions.

They should be used as a measure to gauge the responsibilities of a commit that are difficult to decipher due to feature-nesting. An examplatory use-case is shown in the listing below.

```
if (FtrA) {                     ▷ b9ea2f3    ▷ CheckZero
          ▷ b9ea2f3    ▷ CheckZero
   if (FtrB) {                      ▷ e37a1c8    ▷ CheckZero
      throw Error();                ▷ e37a1c8    ▷ CheckZero
   }                                ▷ e37a1c8    ▷ CheckZero
}                                   ▷ b9ea2f3    ▷ CheckZero
```

Listing 3.2: Use-Case of the Concerns of a Commit

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.

3.6 FEATURE NESTING DEGREE

As dicussed in section 3.4, feature-nesting is an important issue worth dealing with. Instructions that are part of several feature regions, also consitute the size of the respective features. At the same time, it is unlikely that the instruction actually implements functionality of every one of them. Thus, the size of a feature can encompass many instructions not implementing its functionality. Technically, we can only be certain of an instruction's purpose if it is exclusively part of a single feature region. In order to achieve a more accurate description of a feature's size, we introduce the concept of a feature's nesting degree:

Definition 9. The *nesting degree* of a feature is the fraction of its size where its respective region does not appear exclusively.

3.7 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 an estimate on how many commits were used in the development of a feature and the according distribution among the features of a project. 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 of features **??**. In sections 3.4 and 3.6, we discussed feature-nesting and the risks it bears for our analysis. Namely that a fraction of a feature's size can consist of instructions that do not implement its functionality. The same is the case for commits structurally interacting with a feature, meaning that a portion of them possibly didn't participate in its development, but in the development of other features. We therefore consider the frequency of feature-nesting within a project, so that we can tell how accurately our initial data reflects the development of features.

Examining the Usage of Commits in Feature Development

Secondly, we investigate the number of features a commit usually changes by examining its structural CFIs. 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 have a single responsibility. As different features implement separate functionalities, it's unlikely for a commit to change several features while dealing with the same issue. 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. In section 3.4, we discussed that, due to feature nesting, a commit can structurally interact with multiple features at the same time while only changing functionality of one of them. Instead of considering the features a commit structurally interacts, we rather examine the number of its feature-related concerncs 7 for this RQ. This way we aim to achieve a more accurate predittction for the number of features a commit changed or implemented. 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.

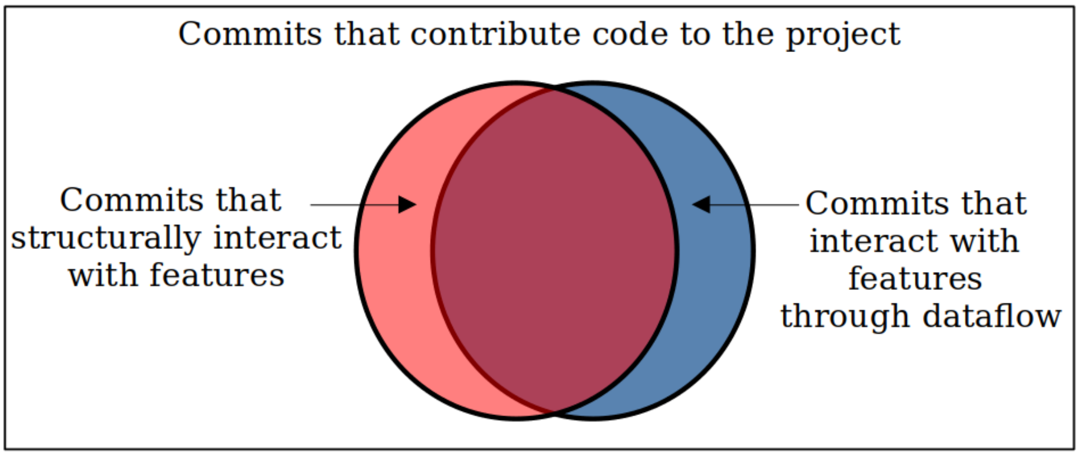
Commits that contribute code to the project
Commits that structurally interact with features
Commits that interact with features through dataflow

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. 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. For a simpler recognition of a distribution, the features are sorted in an increasing order according to their value in the metric of the respective plot. This also allows us to see whether the rank of a feature in one metric is indicative of its rank in the other metric and vice versa. Additionally to the size of a feature, we also determine its nesting degree(9). The average nesting degree among the features of a project is subsequently shown in a LaTeX-table. The strength of a project's overall nesting degree can in turn give us a direct estimate on the frequency of feature-nesting inside of it. 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 linear correlation, we compute the pearson correlation coefficient and its p-value in the STATS submodule of the SCIPY python namespace. Both values are subsequently inserted into the regression plot with the text-function of the MATPLOTLIB. 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%.

Examining the Usage of Commits in Feature Development

For each commit part of at least one structural interaction, we determine the number of its concerns 7. For this, we examine the complex structural CFIs discussed in our Implementation. There, we increment the number of a commit's concerns upon finding a new set of features it structurally interacts with at the same time, i.e. within the same instructions. Following this, we give a comprehensive overview of our results in a SEABORN histplot for every investigated project. The x-axis of the histogram shows the number of a commit's concerns, whereas the y-axis shows the number of commis for which the respective x-value matches. If 50 commits deal with a single concern 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 are likely to have only changed a single feature. To facilitate comparisons between projects, we choose the same x- and y-labels for every histogram. Commits with exceptionally many concerns are shown in an additional x-tick to avoid an overly long x-axis. Instead, we mention these commits explicitly during our evaluation and relate the number of their concerns to their specific purpose.
In a second analysis step, 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. To asses the effect of filtering exceptionally large commits, we compute the average number of concerns within a project before and after the application of our restriction. Both averages are subsequently shown in a cross-project table so that we can compare them within a project and recognize a general trend among them.

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 in detecting 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 high number of interacting commits or a high proportion 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. In 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. We could also perform other regression analyses, for example a logarithmic regression, as they could fit our datapoints better. Following our explanations in section 4.1, we expect the ratio of outside to inside commits to decrease with increasing feature size. This would result in a negative linear correlation coefficient and a falling linear regression line. It remains to be seen whether the respective values are statistically significant across several projects.

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. To faciliate expressing this, we simply say that the respective authors interact with features based on a certain interaction type or that they are a feature's structural or dataflow authors. In section 4.1, we mentioned that we only consider commits outside of features when examining the authors interacting with them through dataflow. The computation of these commits 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 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. Finally, we perform the same linear regression anaylsis as we have done for feature-sizes and the number of commits that interacting with them. By calculating the strength of their correlation, we are able to determine whether the presumed extent of a feature inside a project is an accurate predictor for the number of developers needed to implement it. The authors interacting with a feature through outside dataflow are also indirectly involved in its development, since the data stemming from their contributed commits is necessary for the functionality of the feature.

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

In the first part of RQ1, we examine the number of commits involved in the development of a feature and relate it to its size. 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. For all projects, we notice a wide distribution of structurally interacting commits between features. Numerous features, across all projects, interact with less than 5 commits suggesting that they need very little work to be implemented in comparison to other features. Both xz and LRZIP have features structurally interacting with more than 20 commits, while GZIP even has four features that interact with more than 60 commits. With a mean of 29, the features of GZIP have by far the highest number of interacting commits on average. XZ, LRZIP and BZIP2 have much lower averages at 9, 16 and 4 respectively. As all projects are of a compression domain, we can find several features with the same functionality across the examined projects. The verbosity-feature structurally interacts with the most commits in xz and BZIP2, while the same is not the case for the verbose-feature of GZIP. Although GZIP encompasses the highest average number of interacting commits, its recursive-feature only interacts with 2 commits, while the recursive-feature of LRZIP interacts with 24 commits.
In the second column of figure 5.1, we display the calculated feature-sizes for each project. Again, we see a wide range of different feature-sizes within all projects. Most notabely, there are large jumps between adjacent features, for example from to_stdout with a size of 450 to no_name with a size of 7300 in GZIP. Similarly to the number of interacting commits, GZIP has by far the highest average feature size at 2500. The average feature sizes for the other projects range from 190 for xz to 390 for BZIP2. In table 5.1, we display the average nesting degree of features for all projects. Feature-nesting is the least common for xz and LRZIP and most common for GZIP with an average of 0.93 indicating that the majority of features are nested to large extents. This also the case for the four, by far largest, features of GZIP, namely no_name, method, force and decompress. Specifically, 90 to 100% of the instructions constituting their size are identical among the mentioned features. It is therefore not surprising that most, particularly 66, of the commits interacting with each feature are identical for all four features. Here, it is highly doubtful whether the size of a feature and the number of interacting commits reflect the actual number of `implementing` commits and instructions of a specific feature.

Table 5.1: Additional Information for Structural Analysis of Features

Projects	**Average Feature Nesting Degree**
xz	0.34
gzip	0.93
bzip2	0.64
lrzip	0.37

Finally, the metrics used in the two previously dicussed plots are compared to each other using the regression plot of figure 5.1. That is, we compare the size of a feature with the number of commits that structurally interact with it. The values of the blue dots each represent a feature and are therefore identical to the values of the bars in the two previous columns. For both XZ and GZIP, we determine a strong positive correlation between the size of a feature and its number of structurally interacting commits. Their linear correlation coeffecients are close to 1 at 0.91 and 0.978 respectively with p-values smaller than 10^{-4}. This data provides strong evidence that the observed correlation is statistically significant. While BZIP2 and LRZIP have positive correlation coefficients of 0.585 and 0.968 respectively, their p-vlaues are relatively high at over 0.1. The high p-values can be explained by a lack of datapoints for LRZIP and conflicting datapoints for BZIP2. For example, do both the opMode- and srcMode-feature of BZIP2 structurally interact with 6 commits, but encompass vastly different feature sizes. Overall, the two projects do not produce conclusive statistical evidence that the size of a feature and the number of interacting commits is positively correlated.

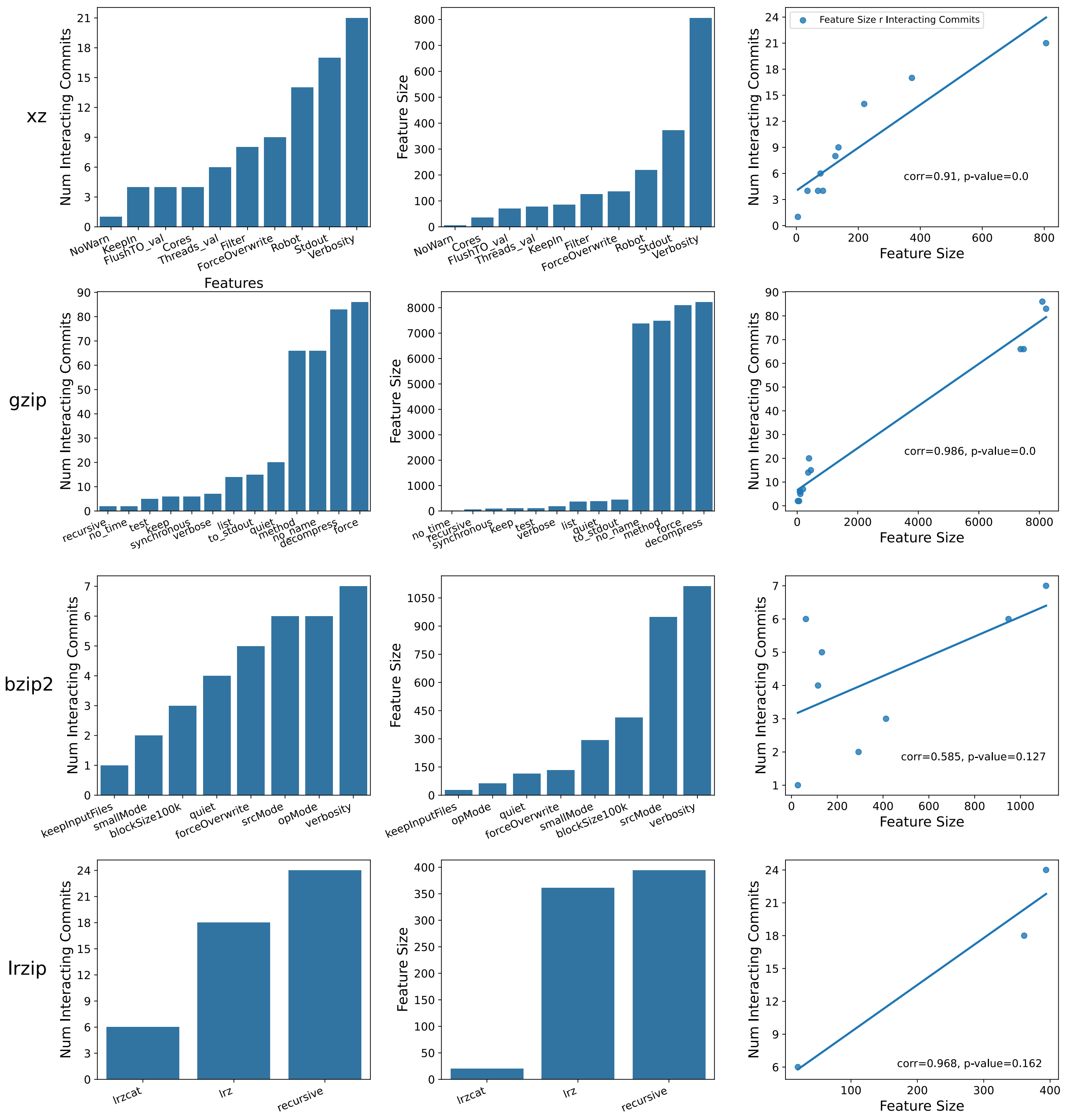

Figure 5.1: Feature Structural CFIs Plot

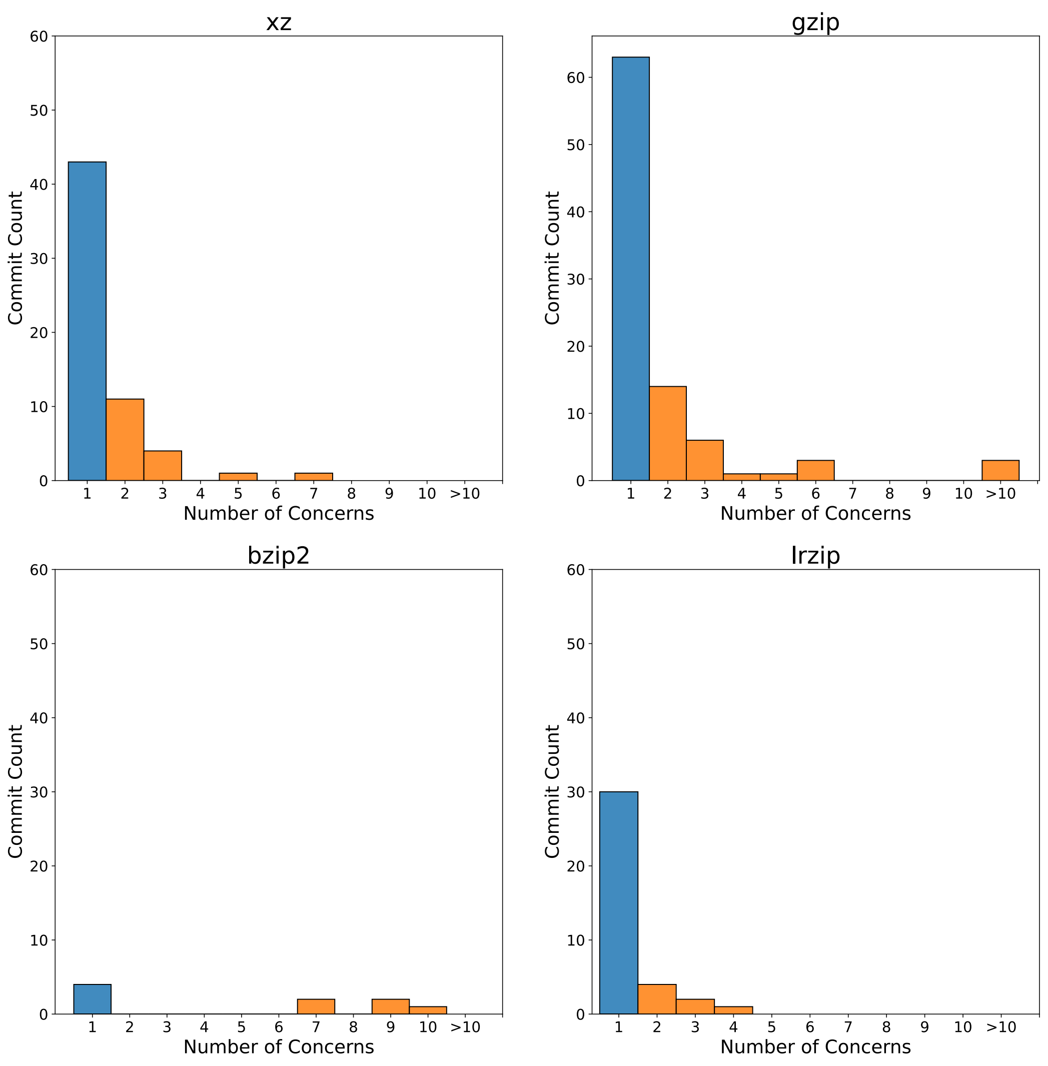

Figure 5.2: Commit Structural CFIs Plot

Usage of Commits in Feature Development

As discussed in 4.1, a best practice for commits in feature development is that one commit should generally change only one feature. We investigate to what extent this holds in our investigated software projects by examining the number of feature-related concerns of a commit. The distribution of how many commits have a certain number of concerns is displayed in figure 5.2. XZ, GZIP and LRZIP show similar distributions, which is why we discuss them now and BZIP2 later. For XZ, GZIP and LRZIP, we note that the majority of commits only have single feature-related concern. The according percentages for commits with a single concern among all considered commits are 72%, 69% and 81%. The commit count gradually drops with an increasing number of concerns until it subsides to 0 at an x-value of 7, 6 and 4 respectively. Interestingly, three commits of GZIP have more than 10 concerns with the commit 33ae4134cc even encompassing 47 concerns. Said commit originally introduced the gzip.c-file containing the *main*-function among other things. The exceptionally high number of

Table 5.2: Additional Information for Structural Analysis of Commits

Projects	Average Number of Concerns	After Filtering Large Commits
xz	1.48	1.29
gzip	2.4	1.55
bzip2	5.11	4.62
lrzip	1.3	1.3

33ae4134cc's concerns can be explained by the fact that many of gzip's 14 features are partially implemented within gzip.c. Additionally, the code of features is heavily nested inside each other, which means that the instructions stemming from said code encompass many different combinations of feature regions. For bzip2, there are much less commits used in feature-development compared to the other projects with only 9 commits part of structural CFIs. The distribution consists of two clusters, the first being at a single concern and the second cluster being located between 7 and 10 concerns. 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 to the number of their concerns. 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.

In table 5.2, we display the average number of feature-related concerns within a project both before and after filtering exceptionally large commits. The first column contains the project to which the averages in their respective row belong. In the following column, we show the initial averages before applying the mentioned restriction. xz and lrzip encompass the lowest average number of concerns at 1.48 and 1.3 respectively. Although the distribution of gzip shown in figure 5.2 is similar to those of the formerly mentioned projects, its initial average is comparatively high at 2.4. The main cause for this are the discussed three commits of gzip with exceptionally many concerns. After filtering large commits within a project, we note that the averages of xz, lrzip and gzip become more similar. While the average slightly decreased for xz to 1.29, it stayed the same for lrzip. The three dicussed commits of gzip were part of the filtered commits, which explains the large drop of its average to 1.55. Unsurprisingly, the initial average of bzip2 is by far the highest among all projects. Since most bzip2 commits are relatively large, this is also the reason why they are not excluded from our analysis. Here, only one commit with 9 concerns is removed, which lowers the average to 4.58. We also 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

Table 5.3: 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

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. 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.

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 5.3 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 5.3. 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 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 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

Table 5.4: 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

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 5.4. 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%.

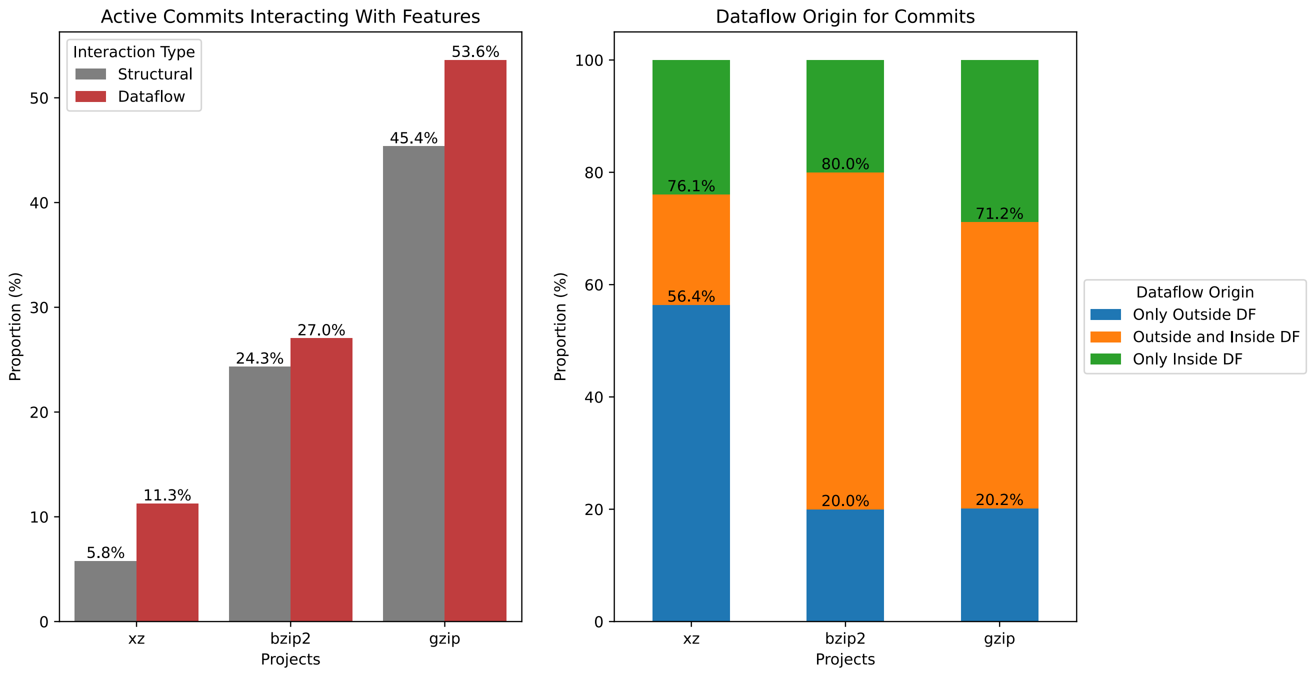

Figure 5.3: Proportional Dataflow-Plot for Commits

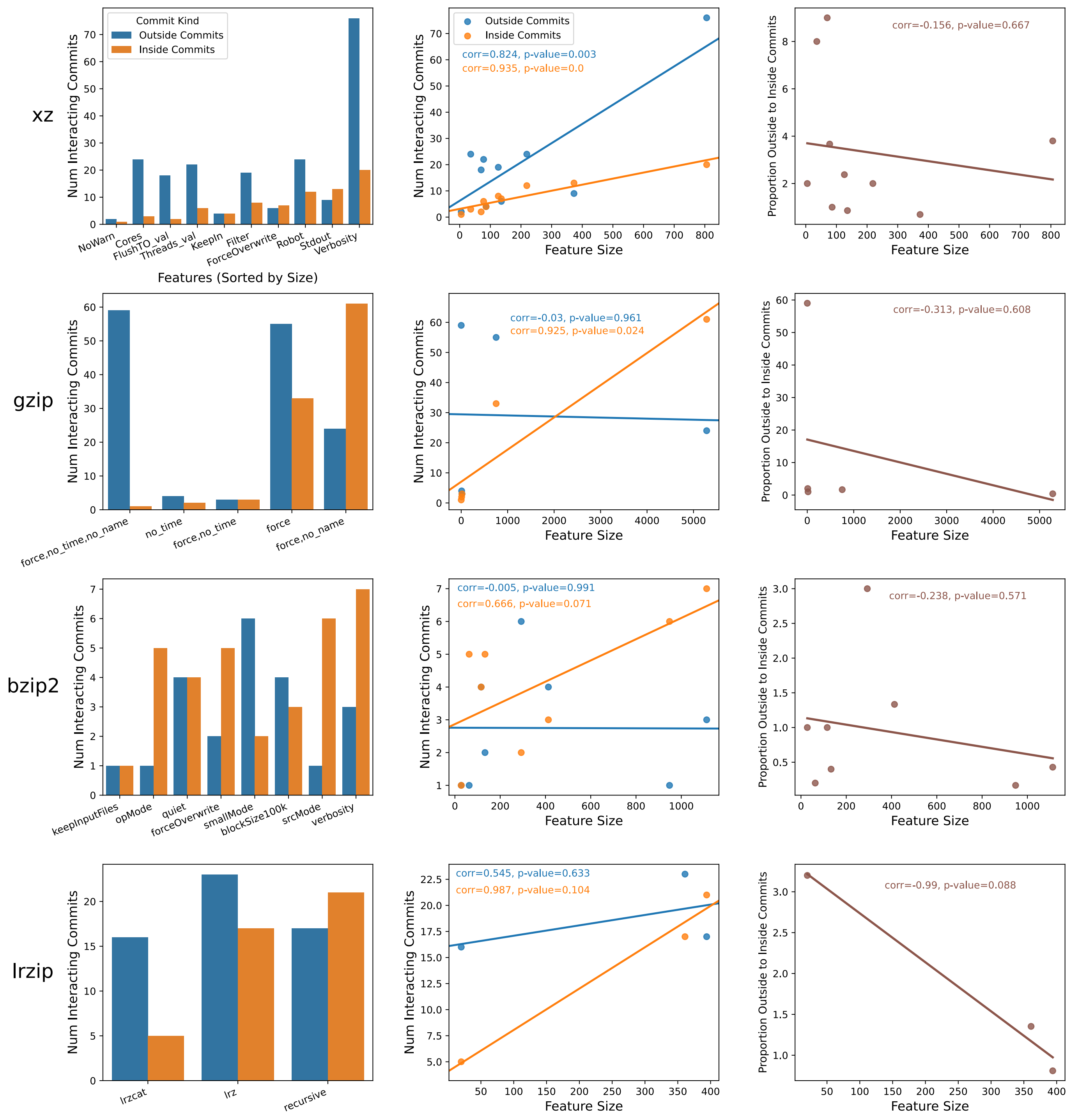

Figure 5.4: Proportional Dataflow-Plot for Features

Understanding Features and the Commits Affecting them Through Dataflow

RQ3: Evaluation of Author Interactions

- the project xz has mostly been developed by a single author, which is reflected in our results
- as can be seen in figure 5.5, all except one feature have exclusively been implemented by the same author
- we can conclude that it also is the developer affecting features through outside dataflow, since no feature has unique dataflow authors
- it follows that the linear regression relating the size of a feature to the number of its interacting authors shows no significant correlation

- BZIP2 inconclusive data, no correlation to be found
- LRZIP more authors of features, but no general trend to be found due to lack of features

- GZIP, on the other hand, provides us with much more comprehensive results
- it can immediately be seen that every feature has a large number of interacting authors
- the overall number of authors that interact with a feature in any way can be computed by the sum of its structural and its unique dataflow authors
- this means that the lowest number of interacting authors is 7 for the recursive-feature and the highest number 14 for the force-feature
- on average, a feature has more dataflow than structural authors with respective averages of 9.1 and 5.7
- the average number of unique dataflow authors is 5.9, which means that our dataflow analysis discovers many additional interacting authors in GZIP
- the number of structurally interacting authors, i.e. authors implementing a feature, range from 2 to 13 between the features of GZIP
- there is less variance in the number of a feature's outside dataflow authors ranging from 7 to 12
- features with relatively few structural authors, have a comparatively high number of dataflow authors
- features with many structural authors, starting from the NO_NAME-feature, have less outside dataflow authors
- at the same time, the number of their unique dataflow authors is rather low, implying that most of them also structurally interact with the according features
- the fact that the group of features with few structural authors has much more dataflow authors conditions them to have many unique dataflow authors
- still, more than 80% of their structural authors also interact with them through outside dataflow
- the same cannot be said for the group of features with many structural authors, where this is the case for less than 45% of them
- overall, there is a 57% likelihood for the structural author of any feature to also interact with the feature through outside dataflow
- similarly to the bar-plot of GZIP, we notice two clusters when examining its regression plots in figure 5.5

- the first cluster is made up of features with few structural authors, whose respective sizes range between 50 and 500
- the second cluster consists of the four features who all have over 10 structural authors and feature-sizes ranging from 5500 to 6500
- it is unsurprising that we compute a strong positive correlation between the size of a feature and its structurally interacting developers
- the respective correlation coefficient is 0.976, while the p-value is smaller than 10^{-4}
- regarding the dataflow authors of features, we note two similar clusters that consist of the same features of smaller and large sizes respectively
- now, the smaller features have slightly more dataflow authors centering around 10 than the much larger features with an average of 8
- this shows itself in a negative correlation coefficient of -0.746 and a p-value of 0.003 falling out of our rejection range
-

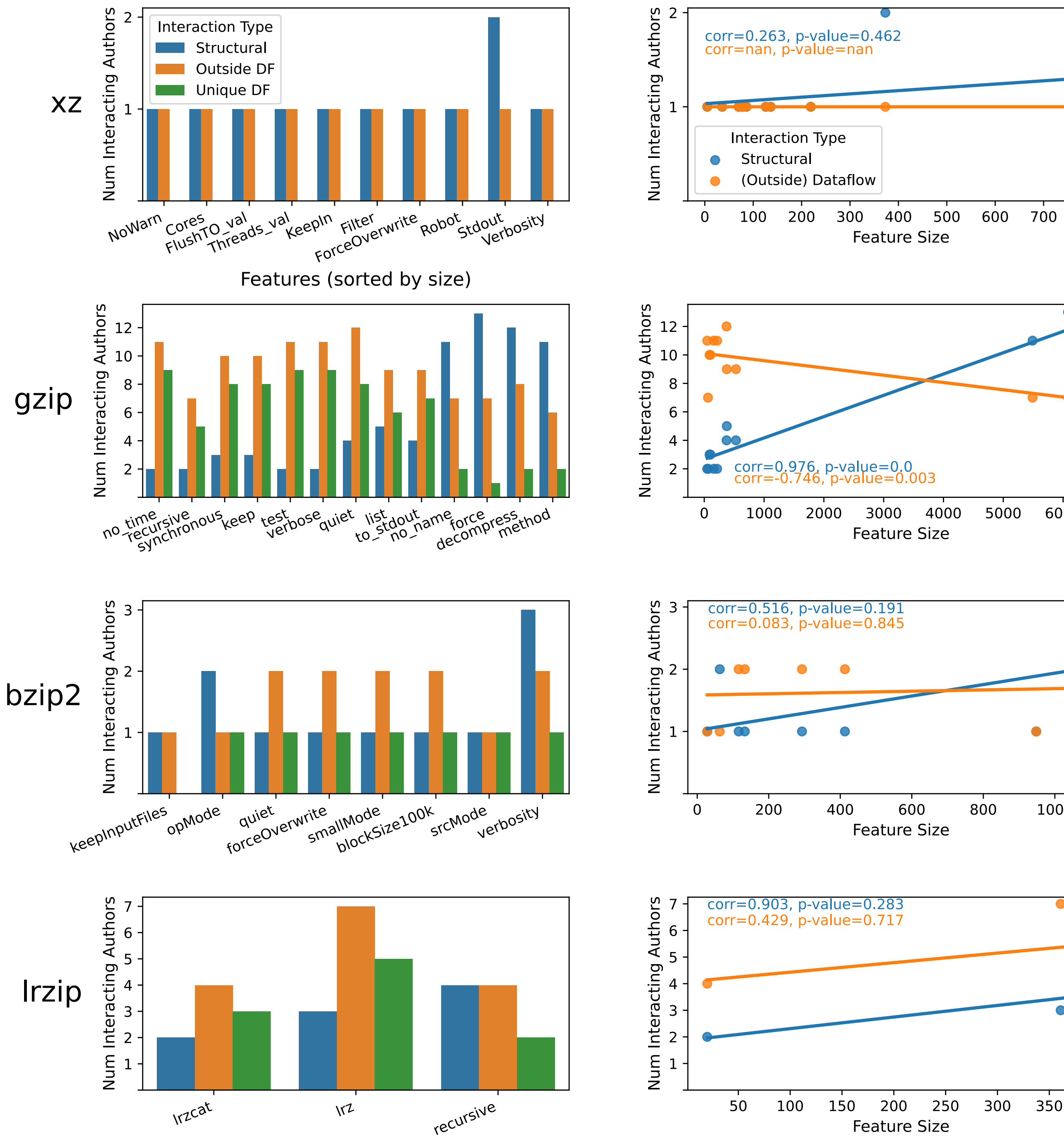

Figure 5.5: Author CFI Plot

5.2 DISCUSSION

RQ1: 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.
When evaluating feature sizes, we noticed that, especially in the case of GZIP, feature-nesting is more common than initially expected. Most of the computed size of the respective features is made up of instructions that are part of multiple feature-regions besides its own. This makes it difficult to clearly interpret the data, as the number of instructions that actually implement a feature can differ greatly from our observed size. 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 commit 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 features 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 and 14 features respectively, XZ and GZIP have by far the most datapoints and according p-values of $> 10^{-4}$ 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, 48 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.
We have already alluded that the 3-4 times higher general average of GZIP compared to XZ and LRZIP stems from its extremely high degree of feature-nesting. Even the large features, force, no_name, method and decompress are mostly nested inside each other. These are also the features all commits, constituting the peak of GZIP's distribution in figure **??**, interact with. The fact that features are implemented in the same code-space makes the discussed distribution rather arbitrary. This supports the notion, that we should rather investigate projects with little feature-nesting to answer our research question of how many features a commit usually changes. In projects with a lesser degree of feature-nesting, the code of different features is located in separate parts of a program. If a commit structurally interacts with several features there, we can be more certain that it was really supposed to change or implement multiple features.
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.