Bachelor's Proposal

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

SIMON STEUER
(2579492)

July 26, 2023

Advisor:
Sebastian Böhm Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

PRESENTATION ABSTRACT

In software engineering, features play a pivotal role in implementing functionality of software applications. There, commits are used to introduce new source code and thus gradually build the overall application and its features. In this paper we connect both entities of software projects in the form of commit-feature interactions (CFIs). For this, the interaction analysis tool VaRA is extended to implement the detection of structural as well as dataflow-based interactions between commits and features. We carry out a comprehensive investigation of CFIs in software projects to provide data-driven conclusions on yet unexplored research areas related to features. These research areas include feature development and the effect of new changes on feature functionality. By analyzing CFIs, we shine light on the dependencies between commits and features, enabling developers to be more aware of them and thus reducing potential errors during software development.

INTRODUCTION

Within a program there exist many different abstract entities, such as commits and features, each serving different responsibilites. For a better understanding of them, it is advantageous to know whether they interact with or among each other. In many cases, it might be difficult to tell whether they do by just studying the program yourself. To facilitate this and allow for a complete overview of them inside a software project, Sattler [4] et al created the interaction analysis tool VaRA. VaRA assigns commits and features a concrete representation, namely code regions , inside a program. This makes it possible to detect different kinds of interactions between them by deploying different analyses techniques, such as a taint analysis to track dataflow-based interactions. In a prior study Sattler et al. [6] investigated commit interactions proving that research on this topic can be applied to improve many topics in software development. For example, their research allowed for a deeper understanding of interactions between authors by linking commit interactions to their respective authors. In this work, we pick up and extend on their research by investigating commit-feature interactions. Features play an important part in modern programming, which shows in program paradigms such as feature-oriented programming. They are used to implement specific functionalities, can be activated or deactivated and thus add configurability to software systems. Commits are an essential part of software development, which means that investigating interactions between commits and features can be used to give insights into the development of features and how changes outside of features affect them. To accomplish this, it's necessary to introduce the concept of structural and dataflow-based CFIs, each encompassing a different meaning.

- structural CFIs => overlap between their concrete representations
- => can produce data revolving around feature-development, as overlap means that commit changed functionality of a feature
- dataflow-based CFIs => dataflow from concrete representation of commit to feature
- => means that changes introduce by the commit influence functionality of a feature

Goal of this Thesis

The primary focus of this thesis is to gain an overview of how commits interact with features in software projects. Our goal is to lay basic groundwork regarding this subject, while leaving more detailed questions to future research. As previously mentioned we investigate two types, strutctural and dataflow-based commit-feature interactions. While using both types separately can already answer many research questions, we also show applications utilizing a combination of both. We aim to reveal insights about the development process of features and usage of commits therein with the help of structural commit-feature interactions and high-level repository information. We also investigate to what extend there exist commit-feature interactions throug dataflow that cannot be discovered with a purely syntactical analysis.

Overview

In related work **??**, we summarize previous studies investigating interactions inbetween commits and inbetween features to motivate that interactions between them is a topic worthy

of study. The background chapter serves as an introduction to the concepts of code regions and interaction analysis, which are necessary to properly define structural and dataflow-based CFIs. Their definition then takes place in the commit-feature interaction chapter, where we thouroughly dicuss their meaning in software projects as well as our implementation in VaRA. The research questions of this paper are established in the methodology chapter in which we also explain how we plan on investigating them. Furhtermore methodology contains the self-explanatory sections expectations **??** and threats to validity . To finish the proposal we summarize our proposed research questions and their planned evaluation in the conclusion .

RELATED WORK

Interactions between Features and Interactions between Commits have already been used to answer many research questions surrounding software projects. However investigating Feature Interactions has been around for a long time whereas 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. 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 and thus, logically, accounting for most author interactions. 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. [2] first implemented functionality to automatically track load-time configuration options along program flow. Said configuration options can be viewed analogously to feature variables in our research. Their analysis tool Lotrack can detect which features, here configuration options, must be activated in order for certain code segments 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.
Referencing this paper Kolesnikov et al. [1] 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.

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

research. In the next chapter, we will use the introduced definitions and concepts to explain and investigate structural as well as dataflow-based commit-feature interactions.

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 directly 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 last changed by it.

Definition 1. The sum 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.

Definition 2. The sum 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.

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 insights on how they are used inside software projects. An important component of computing interactions between concrete representations is the concept of interaction relations between code regions introduced by Sattler [4]. As we investigate structural and dataflow-based interactions, we make use of structural ◎ and dataflow ⇝ interaction relations in this work.
The structural interaction relation between two code regions r_1 and r_2 is defined as follows:

Definition 3. 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. 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 introduced by Sattler [4]. Their interaction analysis tool VaRA is implemented on top of LLVM and PhASAR. VaRA is used to determine dataflow interactions between commits by computing interactions between their respective commit regions. This computation is carried out using the dataflow interaction relation ⇝. 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. VaRA focuses 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 [4] 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-based interaction relation as defined in definition 4.

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 background 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 show how new changes to a program, in the form of commits, affect features.
In the background chapter, we also 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 interaction relation $\circledcirc$ (definition 3).

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 commit-feature interactions carry an important meaning, namely that the commit of the interaction was used to implement or change the functionality of the feature. This can be seen when looking at an instruction that is both part of a commit as well as a feature region. From definition 1, it follows that the instruction stems from a source code line that was last changed by the commit region's respective commit. From definition 2, we also know that the source line implementes the functionality of the feature region's respective feature. This means that the commit contributed to the code implementing the feature. 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. 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. Knowing the developers of a feature can provide a deeper insight into its development than focusing solely on the commits that implemented it.

Determining which commits affect a feature through dataflow can reveil 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 our dataflow analysis, that is dicussed here , can reveil previously hidden interactions between commits and features that programmers were unaware of before.
Commit interactions based on dataflow were explained in the Interaction Analysis**??** chapter 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 produced within a commit region flows as input to an instruction located inside a feature region. This pattern can also be matched with the dataflow interaction relation $\rightsquigarrow$ (definition 4) 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. $r_{iC} \rightsquigarrow r_{jF}$.

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

Listing 1: Commit Feature Interactions

This code example 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 a new variable 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.

When investigating dataflow-based commit-feature interactions, it is important to factor in that structural interactions heavily coincide with dataflow-based interactions. 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, it is important to know which commit-feature interactions can only be discovered by a dataflow analysis. That dataflow interactions coincide with structural interactions becomes clear when looking at an instruction accounting for a structural commit-feature interaction. 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 commit-feature interactions that occur within the regions of a feature and those where data flows from outside the regions of a feature into it. From prior explanations, it follows that this differentiation can be accomplished by simply checking whether a commit that influences a feature through dataflow, also structurally interacts with it.

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 amount of instructions implementing its functionality inside a program. As feature regions implement a feature's functionality, we can the define the size of a feature as follows:

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

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

Implementation

The detection of structural as well as dataflow-based commit-feature interactions is implemented in VaRA [5]. Additionaly to commit regions, VaRA maps information about its feature regions onto the compiler's IR during its construction. Commit regions contain the hash and repository of their respective commits, whereas feature regions contain the name of the feature they originated from. VaRA also gives us access to every llvm-IR instruction of a program and its attached information. Thus, structural commit-feature interactions of a program can be collected by iterating over its compiled instructions. According to definition 5, we can store a structural interaction between a commit and a feature, if an instruction is part of a respective commit and feature region. For each such 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.

With this, we are able to calculate the size of a feature by iterating over the found structural commit-feature interactions. We can increase the size of a feature when we encounter an interaction of the feature by the interaction's instruction counter.

In the Interaction Analysis **??** section we have discussed the taint analysis deployed by VaRA. There, VaRA computes information about which code regions have affected an instruction through dataflow. Focusing on commit regions 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 data, that was changed by a commit region earlier in the program, flows as input to an instruction belonging to a feature region.

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.

Research Questions

In the commit-feature interaction 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 eachother?

We intend to research two main properties which already provide a lot of insight into the development process of features and usage of commits therein. Firstly, we examine the amount 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 scope of feature, which can put the amount of commits used to implement a feature into perspective. Secondly, we want to examine how many features a commit interacts with structurally, e.g. 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 granular[3] meaning they should only fix a single bug or, in our case, change a single feature. Acquiring data on this issue might show how strictly this policy is enforced in the development of features.

RQ2: How do commits interact with features through dataflow?

Investigating dataflow can unveil interactions between parts of a program that were previously hidden from programmers. This can help a programmer understand the extent to which new changes affect other parts of a program. Deploying the introduced analysis in a direct manner could even aid a programmer when fixing bugs of features. Bugs occuring in certain features could be traced back to their cause by factoring in recent changes affecting said features through dataflow.
Previous research has laid the groundwork for researching dataflow interactions between different parts of a program. However, it has focused solely on dataflow interactions between commits. That's why we want to provide first insights on the properties of dataflow-based commit-feature interactions. Specifically, we investigate how connected commits and features are by analyzing the amount of features a commit usually affects through dataflow. Knowing what fraction of all commits contributing code to a project are part of dataflow-based interactions can show how often new commits affect the data of a feature. Regarding this, it is worth considering that commits constituting code of a feature are very likely to influence said feature through dataflow. Since dataflow interactions coinciding with structural interactions are so obvious, programmers are also much more aware of them. Depending on the prevalence of feature-regions in a project's code space, this could heavily skew the data in one direction, as a large portion of all dataflow interactions would stem from these obvious interactions. Therefore we want to especially focus on commits that aren't part of a feature, because programmers might not be aware or intend that changes introduced with these commits also affect features through dataflow.

RQ3: How do authors implement features?

Usually there are many programmers working on the same software project, implementing different features, sometimes alone, sometimes with the help of colleagues. We want to shine

some light on the exact statistics of this by combining structural commit-feature interactions with high-level repository information. One major question we want to answer is how many authors implement a feature on average, where considering feature-scope could help put this data into perspective. The collected results could serve as advice for software companies on how to allocate programmers on to-be implemented features.

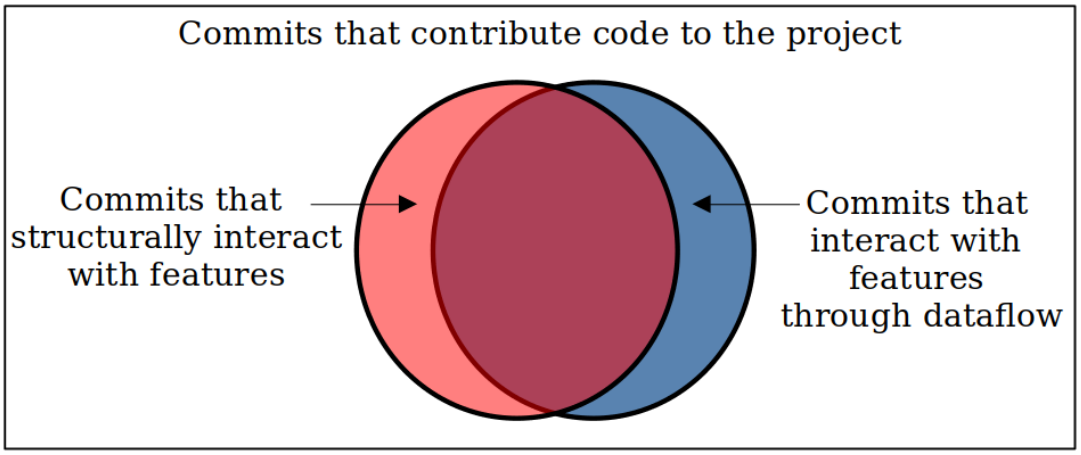

Figure 1: Kinds of Commits in a Software Project investigated in this work

In the first two **RQs** we have discussed different kinds of commits and the ways in which they interact with features. Figure 1 showcases them in a venn diagram and illustrates the dependencies and divisions between them.

Operationalization

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

RQ1: How do commits and features structurally interact with eachother?

For this RQ, we obviously examine reports comprising structural commit-feature interactions. From the collected data we extract the size and the number of commits a feature structurally

interacts with as well as the number of features a commit structurally interacts with. Concerning the first property of structural commit-feature interactions mentioned by us, we calculate the mean and variance of the number of commits used to implement a feature. This gives us an overview of how many commits were used for a feature's development and tells us how much this number varies for different features. A regression analysis on the relation between the size of a feature and the commits used to implement it, allows us to calculate the strength of their correlation. From the commits that structurally interact with features, we calculate the average number of features that a commit interacts with. Using the calculated average, we can determine if commits are primarily used to change just one feature. Showing the distribution of the number of features a commit changes, further allows us to see how often commits change just one or more features. Here, filtering outliers can help produce more sensible data, as commits responsible for refactoring could change many features, while not implementing any functionality.

RQ2: How do commits interact with features through dataflow?

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

RQ3: How do authors implement features?

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

to answer the previous RQs. With this information we aim to correlate the size of a feature with the amount of authors that implemented it in a regression analysis.

Expectations

How and to what extent features are used to implement functionality in the to be examined projects is not known and could vary from project to project. Thus, some results of the dicussed research topics are difficult to predict. For example, nesting of feature regions inside each other could lead to an increase in the amount of features a commit usually changes. Due to the discussed best practices of commits, we expect commits to change at most one feature on average if there happens to be little nesting. Because of the unknown size of features, it's not sensible to give an estimate about the amount of commits needed to implement a feature. We expect a rather strong positive correlation between the size of a feature and its commits however. It was mentioned that we examine small projects meaning that the pool of developers is limited in size. Besides that features implement specific functionality that some programmer's might have a better understanding of than others. This leads us to the expectation that a feature is implemented by a small share of all developers contributing to a project. Because of the small pool of developers and prior research findings, we expect the existence of a main developer that contributes most if not all of the functionality of a feature. We know that commits structurally interacting with features most likely are part of dataflow interactions with them as well. Excluding such commits, the extent to which commits interact with features through dataflow depends heavily on what fraction of the code space is made up of feature regions. The purpose of features is to implement additional and sometimes necessary functionality separate from the main program. For this they access and change specific data according to their intended functionality. Provided that feature regions only make up a small portion of the program, we do expect relatively few, albeit important dataflow interactions between commits and features.

Threats to Validity

There are some potential threats to the internal validity of our gathered data, which stem from our implementation in VaRA.
From definition 2 of feature regions, it follows that we implement feature regions in such a way that any instruction whose execution depends on a configuration variable, is part of a feature region. However, not every such instruction also implements the functionality of a feature, as can be seen in listing 2. This means that feature regions overapproximate the amount of instructions responsible for a feature's functionality. Since feature regions are used for computing both structural and dataflow-based commit-feature interactions, they are overapproximated to some extent as well. Thus, it's possible that commits of structural interactions don't actually implement functionality of a feature and commits of dataflow interactions don't actually affect instructions implementing a feature's functionality through dataflow. Furthermore Sattler et al. [6] explains that our deployed VaRA taint analysis does not necessarily detect all dataflows occuring in a program. This results in taints being under-approximated, meaning that some instructions are not tainted when they correctly should be. Thus, some dataflow interactions could be missed by our deployed commit-feature interaction

detection.
Concerning the external validity of our findings, most dangers come from the selection of which projects we investigate. Our pool of investigated projects is limited and it's likely that the way commits and features are used in them is different to other projects to some extent. In previous chapters we have discussed that the chosen projects are rather small and many of them are from the same domain. This could mean that our findings might not be applicable for projects of larger size or of different domains. As we already factor in the size of a feature in the analysis of our data, we are able mitigate some doubts about the applicability of our results onto larger projects, as we can scale them accordingly.

```
1. if FeatureEncryption:                          ▷ FeatureEncryption
2.     sendEncryptedMessage(message)              ▷ FeatureEncryption
3. else:                                          ▷ FeatureEncryption
4.     sendMessage(message)                       ▷ FeatureEncryption
```

Listing 2: Feature Region Overapproximation

The function of `FeatureEncryption` is to send the message encrypted. According to our definition of feature regions all instructions stemming from the shown lines of code belong to a region of `FeatureEncryption`, as their execution depends on the configuration variable of `FeatureEncryption`. However only instructions stemming from the lines 1-2 implement the actual functionality of the feature. Thus our analysis overapproximates the lines 3-4 to also belong to the feature.

CONCLUSION

In this work we research the main properties of structural and dataflow-based commit-feature interactions. Using high-repository information about commits and their authors as well as a combination of both types of interactions allows us to gain additional knowledge on their properties. To investigate these properties we examine several small software projects revolving around compression, such as xz. For this, we create reports containing all found structural and dataflow-based commit-feature interactions of the to be examined projects. The collected data is then processed to facilitate performing calculations with it and displaying it graphically. Structural interactions and the injection of author information within them can be utilized to provide insights into feature development and usage of commits therin. This includes showing how often commits implement more than one feature and how strongly correlated the size of a feature and the number of commits used to implement it are. Furthermore we calculate the average number of authors that implement a feature in a project. Dataflow interactions are used to unveil interactions between features and commits that cannot be discovered through a purely structural analysis. Seeing how common they really are could encourage programmers to be more aware of them. To gain some insight on this, we measure what fraction of commits contributing code to the project affect features through dataflow. This can also improve our understanding on which impact new commits have on features, as it gives us an estimate of how likely new commits are to influence the data of a feature.

BIBLIOGRAPHY

[1] 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).

[2] 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.

[3] Luis Matos. *Git Tips and Git Commit Best Practices*. https://gist.github.com/luismts/495d982e8c5b1a0ced4a57cf3d93cf60 [Accessed: (24.07.2023)]. 2023.

[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.