Bachelor's Thesis

# FEATURE PERFORMANCE ANALYSIS: DIFFERENCES BETWEEN BLACK-BOX AND WHITE-BOX MODELS IN CONFIGURABLE SYSTEMS

MANUEL MESSERIG

April 9, 2023

Advisor:

| | |
|---|---|
| Florian Sattler | Chair of Software Engineering |
| Christian Kaltenecker | Chair of Software Engineering |

Examiners:

| | |
|---|---|
| Prof. Dr. Sven Apel | Chair of Software Engineering |
| Prof. Dr. Jan Reineke | Real-Time and Embedded Systems Lab |

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

## 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

Nearly all modern software systems are configurable. To give the end-user flexibility each system include various configuration options. However, it is not clear to the end-user how these configuration options might interact and influence properties such as performance. For this reason, we investigate this by using two different analyses, a white-box analysis and a black-box analysis. To present the influence of each configuration option and interaction between them using performance-influcence model, which we use to compare and evaluate both analyses.

In this thesis we analyze, 5 different configurable software systems we engineered and the compression tool XZ using both white-box analysis and black-box analysis.

TODO: INSERT FINDINGS

# CONTENTS

# LIST OF FIGURES



# LIST OF TABLES



# LISTINGS

## ACRONYMS

# 1 INTRODUCTION

Modern software systems are designed to be configurable; they offer users flexibility by providing them to turn functionality on and off. This flexibility allows a configurable software system to satisfy the demand of multiple users by offering a single software system that contains multiple features [1].

An example of such a configurable software system would be the Linux kernel, whose code base itself contains over 6'000'000 lines of code consist of more than 10'000 features [10]. All these features implement functionality, which allows the user to select the features he wants to create an operating system that meets the user's needs.

All these features affect the system in different ways. To keep track of all features and their interactions, we use a feature model, that describes how features interact with another. Inside the feature model, there can be different kinds of constraints to visualize the relationships between features in a configurable system [1].

Now that we have an overview of the system we are interested in what capacity each feature or feature interaction influences the system, To identify these influences we present two different analyses: a white-box analysis and a black-box analysis. However, before this, we analyze the system's configuration space and select the crucial features of the system. Then, we create configurations from the selected features that contain the interactions we want to observe.

In the black-box analysis, we run the system with each configuration as input, and during execution, we can measure various metrics. Whereas we will focus on measuring the execution time. We will use the measurement we collect to learn each feature's influence on the system using multiple linear regression.

In our white-box analysis, we have more information because we have access to the system's source code. We use an analysis that helps us determine which parts of the code are influenced by which feature. During execution, we can measure the time spent inside these features.

Now both our analyses generate different types of data that are different. Therefore, we need to transform this data into a model that we can then use to evaluate both analyses by comparing these models; for this, we use performance-influcence model, which are machine learning models that have been used successfully in previous works. We build these models by using the data generated by the white-box analysis and black-box analysis.

To show the validity of our models, we establish a ground truth. To do this, we design a small configurable system to test both of our analyses. This system will contain several features, some of which interact with each other in different ways. Since we developed this system ourselves, we know how each feature should impact the runtime of our system, so we create a baseline performance-influence model to compare our models against.

## 1.1 GOALS OF THIS THESIS

In this thesis, we aim to determine if white-box and black-box analyzes can identify the influence of the different features and feature interactions of a configurable system. Furthermore, we are interested in the differences of the performance-influcence models built with data generated by each analysis.

We are interested in whether both models can correctly identify the influence of each feature and feature interactions and if they reach the same results. We are particularly interested in the differences between them and examine whether one model performs particularly poorly in some instances and why this is so.

We collect all these interest points to form the following research questions that we answer during this thesis:

RQ1 : How accurately does white-box and black-box models detect feature and feature interactions?

RQ2 : Do performance models created by our white-box and black-box attribute the same influence to each feature?

RQ3 : What are the reasons for similarities or differences between performance models?

# 2 BACKGROUND

This chapter introduces the general concepts on which we build on in this thesis. We explain the concepts behind *configurable software systems* and afterwards, how we model these configurable software systems with feature models. We introduce *performance-influcence models*, a way to model and predict the influcence of features on the configurable software system. In Section 2.4 and Section 2.5, we introduce *black-box analysis* and *white-box analysis*, two different analysis approaches to analyze the performance of configurable software systems.

## 2.1 CONFIGURABLE SYSTEMS

In this section, we explain the general concepts behind configurable software systems and the benefits and challenges when using them.

We explain what a feature and configuration is in the context of configurable software systems in Section 2.1.2. We introduce functional and non-functional properties in Section 2.1.3 and highlight differences between them.

### 2.1.1 *General Concepts*

We call a software system configurable if it offers options that allow developers to select functionality to change the behavior of the system. By this, the system can satisfy the demand of multiple user groups. While we provide only a single software system that include various features for users to choose from [18].

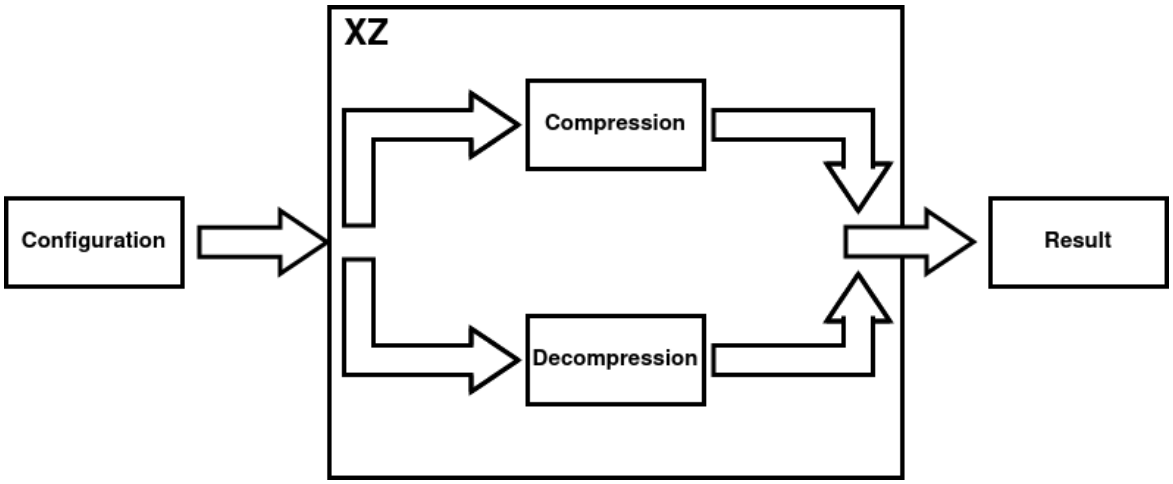


Figure 2.1: Simplified version of XZ.

As an example, let us inspect the compression tool XZ[1]. Figure 2.1 depicts a simplified version of XZ, which contains two main functions: compression and decompression. It is up to the user to decide what he needs, for when he wants to compress a file he would select a configuration that enables this functionality, but regardless the choice, the software system contains both functions.

### 2.1.2 *Features and Configurations*

During the years there have been many definitions of what a *feature* is, on one side, features are used as a means of communication between the different stakeholder of a system, where on the other hand, a feature is defined as an implementation-level concept. To unify both usages Apel et.al. introduced the following definition [1]:

**Definition 2.1.1.** "A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option."

Thus, a feature is both an abstract concept that refers to particular functionality of a system and the implementation of that functionality. In our example Figure 2.1 both, *Compression* and *Decompression*, are unique features that refer to a piece of functionality of XZ and the implementation of the functionality.

Inside a configurable software system, features are not independent of one another. Most of the time, features influence the behavior of different features. When this happens, we say that these features interact with each other. Due to this, we are interested in the degree to which a feature interaction influences the system.

We differentiate between *binary* features and *numeric* features. A binary feature can be either selected or deselected. Commonly, when we select a binary feature we represent it with 1 and 0 otherwise. A numeric feature is a feature which, if selected, requires a numerical value that specifies a different behavior of that feature. These can have various meanings depending on the feature it implements. For Figure 2.1, *Decompression* could be modeled as a binary feature, since we have the option to decrypt a file or not. On the contrary, *Compression* could be implemented as a numeric feature, where the numeric value represents the quality of compression we want.

A configuration option is a predefined way for developers to change the functionality of the configurable system. These options allow us to select features we want to include or exclude. Therefore, a configuration is a set of configuration options, whereas we call a configuration valid as long as the selection of configuration option is allowed by the system.

The configuration space refers to the set of all configurations. Some of these configurations can be invalid. As we cannot execute systems with invalid configurations in practice, our work focuses on valid configuration, hence, when we refer to a configuration space we always mean the space of valid configurations, except otherwise noted.

1 Visited at 21.03.2023 https://tukaani.org/xz/

### 2.1.3 *Functional and Non-functional Properties*

When analyzing a configurable system, we distinguish between what a system can do and how the system archives that goal. The former refers to the functional properties, while the latter describes the non-functional properties of the system [14]. When we talk about functional properties, we mean everything a system can do, including which problem it solves and what features the system provides us, the user. For example, in Figure 2.1, we can see the features of XZ, *Compression*, and *Decompression*; they refer to the functionality XZ provides.

While functional properties refer to what a system can do, non-functional properties refer to how or in which circumstance that functionality is achieved [14]. Examples of non-functional properties are performance, memory consumption, CPU usage and energy consumption. However, not all non-functional properties are quantifiable. Some relate to the quality of a system that we can not easily measure, such as how secure a system is or how high the code quality is in regard to code readability or documentation [14].

## 2.2 MODELLING CONFIGURABLE SYSTEM

In this section, we explain how we model configurable systems using *feature models*. Furthermore, we introduce *feature diagrams*, a visual representation of feature models.

### 2.2.1 *Feature Models*

A configurable system often contains numerous features and different dependencies. However, not all features are freely combinable, some of them can only be selected when another is deselected. The large the system the harder it gets to keep all these constraints in mind [1], therefore we use *Feature Models* to describe the relation between features by that and define which feature selections are valid [1].

Due to the fact that large configurable systems tends to have numerous feature we use feature diagrams which are a common visual representation of feature models.

### 2.2.2 *Feature Diagrams*

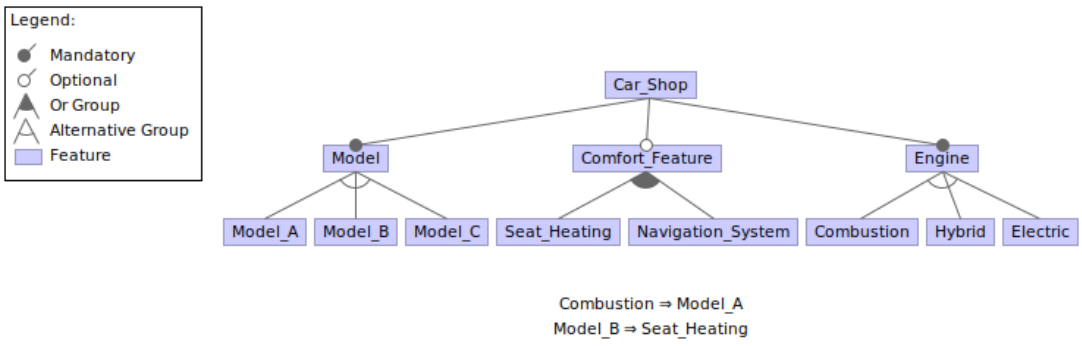


Figure 2.2: A feature diagram of a car dealership.

A feature diagram is conceptually built as a tree where each node of the tree represents a feature, for which edge between a parent node and a child usually implies that the parent is a more general concept and the child a specialization. So, the further we descend in the tree, the more specialized the feature gets, with regard to its subtree.

Figure 2.2 is an example for a feature diagram that depicts the structure of a car dealership. We see that the root node *car_shop* represents the overall system. As we descend we look at the child nodes of the *Engine* feature, we can see concrete engine types, such as *Combustion*, *Hybrid*, and *Electric*.

We differentiate between two types of feature nodes: *abstract* and *concrete* feature. Abstract features are used to structure the diagram, as such they not correspond to a concrete implementation. In contrast, concrete features reflect variability inside the system, where a concrete feature is bound to an implementation artifact [1].

In Figure 2.2, the feature *Engine* is considered an abstract feature since it does not implement anything and is only used to give the feature diagram more structure by grouping all kinds of engines a car can possess and signaling the user that he must select a specific engine. The specific engines, like *Combustion* is a concrete feature since it implements a specific type of functionality.

Each feature contains a graphical notation indicating whether the feature is mandatory or optional. If the feature is mandatory, it is indicated with a black bubble any if it is optional, it is displayed with an empty bubble [2]. In Figure 2.2, we can see that *Engine* and *Model* are mandatory features, which make sense the first two are necessary for any car but *Comfort_Feature* is optional and not necessary for a car to function.

In addition to mandatory and optional features, there are alternative and choice groups. A parent can have one of the groups the alternative group is marked with an empty half circle and the choice group with a filled circle. When we use a choice group, one feature must be selected, but others can also be selected. Therefore, the choice group corresponds to the logical or operator. In an alternative group, we can select only one feature; the configuration is invalid if more than one feature is selected [2]. In Figure 2.2, we see that *Engine* has an alternative group, which makes sense since each car can only contain one *Engine*, whereas it makes sense that *Comfort_Feature* contains a choice group, you can have a navigation system and seat heating in a car without conflict.

A feature diagram may contain various constraints that need to be satisfied, defined as boolean algebra. In Figure 2.2, we see two constraints, $Combustion \implies Model\_A$ and $Model\_B \implies Seat\_Heating$. The reason for such constraints could be that *Model_B* is a luxury model that only gets shipped with seat heating.

We use a feature model to check whether a configuration is valid, which is particularly useful if we want to sample or enumerate all valid configurations [2].

## 2.3 PERFORMANCE-INFLUENCE MODELS

In the previous Section, we introduced a way to represent variability by using feature models. However, while doing so we ignored the non-functional properties which are as important for configurable software systems. To model the measurable non-functional properties and to which degree each feature influences the configurable system, we introduce performance-influence models. A performance-influence model is a polynomial consisting of several

terms, each representing either a feature or an interaction between features, whereas the coefficient in each term represents the degree to which these features influence the system. The sum of all terms represents the time the performance-influence model predicts given a configuration of features [13].

Formally, let $\mathcal{O}$ be the set of all configuration options and $\mathcal{C}$ the set of all configurations, then let $c \in \mathcal{C}$ be a function $c : \mathcal{O} \implies \{0, 1\}$ that assigns either 0 or 1 to each binary option. If we select a feature $o$, then $c(o) = 1$ holds, otherwise $c(o) = 0$. In general, a performance-influence model is a function $\Pi$ that maps configurations $\mathcal{C}$ to an estimated prediction, therefore $\Pi : \mathcal{C} \implies \mathbb{R}$ [13].

We encode all our features as binary features and distinguish between single features $o$ denoted as $\phi_o$ and feature interactions $i...j$ denoted as $\Phi_{i...j}$. Based on these definitions, we define a performance-influence model formally as [13]:

$$\Pi = \beta_0 + \sum_{i \in \mathcal{O}} \phi_i(c(i)) + \sum_{i...j \in \mathcal{O}} \Phi_{i...j}(c(i)...c(j)) \tag{2.1}$$

While $\beta_0$ denotes the base performance, which refers to the time taken by the system regardless of configuration, $\sum_{i \in \mathcal{O}} \phi_i(c(i))$ is the sum of each feature and $\sum_{i...j \in \mathcal{O}} \Phi_{i...j}(c(i)...c(j))$ is the sum of each feature interaction [13].

Listing 2.1: Example code of a simple configurable software system that contains 4 features

```
1  void foo() {
2      bool A, B, C, D;
3      assign_feature(A, B, C, D); //Assigns true to each selected feature
4
5      fpcsc::sleep_for_secs(2); //Spending time in base feature
6      if(A)
7          fpcsc::sleep_for_secs(1);
8      if(B)
9          fpcsc::sleep_for_secs(2);
10     if(C)
11         fpcsc::sleep_for_secs(1);
12     if(D)
13         fpcsc::sleep_for_secs(2);
14     if(A && B)
15         fpcsc::sleep_for_secs(2);
16     if(C && D)
17         fpcsc::sleep_for_secs(0);
18 }
```

In Listing 2.1 we see a simple code snippet with some features that affect the performance in different ways. In Line 3, four features $A$, $B$, $C$, and $D$, are declared, each of which can either be *true* or *false* depending on the configuration chosen. Line 7, 9, 11, 13 will only be executed if the corresponding features are active. If this is the case, the system sleeps for the specified time. In Line 14, we have a feature interaction where $\{A, B\}$ must be active in order for Line 15 to be executed, we would attribute the time spent in Line 15 to the feature

interaction $\{A, B\}$ and not to either feature alone. The performance-influence model for our system would look as follows:

$$\Pi = 2 + 1 \cdot c(A) + 2 \cdot c(B) + 1 \cdot c(C) + 2 \cdot c(D) + 2 \cdot c(A) \cdot c(B) + 0 \cdot c(C) \cdot c(D) \quad (2.2)$$

For simplicity, let us assume that the execution of the code takes no time at all, and we spend no time in any feature except the time specified in the *sleep_for_seconds* function. The constant 2 here refers to $\beta_0$, the time we spend in our base feature in Line 5. If we decide on the configuration $\{A, B, C, D\}$ the model would evaluate like this:

$$\begin{aligned}
\Pi &= 2 + 1 \cdot c(A) + 2 \cdot c(B) + 1 \cdot c(C) + 2 \cdot c(D) + 2 \cdot c(A) \cdot c(B) + 0 \cdot c(C) \cdot c(D) \\
\Pi &= 2 + 1 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 + 2 \cdot 0 + 2 \cdot 1 \cdot 1 + 0 \cdot 1 \cdot 0 \\
\Pi &= 2 + 1 + 2 + 1 + 2 + 2 \\
\Pi &= 10
\end{aligned}$$

Thus, for the configuration containing $\{A, B, C, D\}$ our performance-influence model would predict an expected time of 10 seconds.

## 2.4 BLACK-BOX ANALYSIS

In this section, we introduce black-box analysis and how we use it to analyze configurable software systems. In Section 2.4.1, we explain the general concepts of a black-box and black-box analysis. Afterwards, we highlight the challenges we encounter when using a black-box analysis. After using the black-box analysis, we use the obtained data, to build a performance-influence model using multiple linear regression in Section 2.4.2.

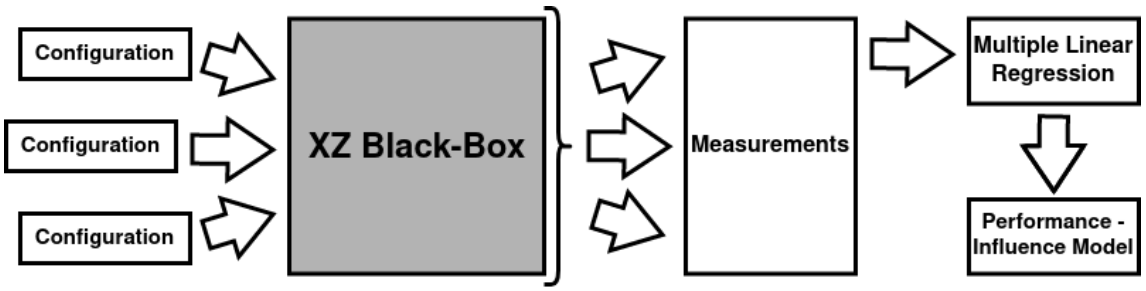


Figure 2.3: Process of using a black-box analysis to build a performance-influence model for *XZ*.

In Figure 2.3, we use a black-box analysis for *XZ* to build a performance-influence model. We start by focusing on finding the features that we are interested in. Then we build ourselves multiple configurations that hold different interactions between these features. Afterwards, we run *XZ* as a black-box on each configuration; during the execution, we analyze the system by measuring different non-functional properties. In our case, we are interested in the performance of each feature. We repeat this process for each configuration during the black-box analysis and collect the measurements. Next, we use these measurements together with multiple linear regression to build a performance-influence model.

### 2.4.1 *General Concepts*

We have introduced performance-influcence model to represent each feature and feature interaction's influences. In this section, we expand on this topic and introduce the *black-box analysis* a method to collect data to build performance-influcence model.

A black-box of a configurable system is conceptually simple, we execute a given system with a configuration, and after finishing, we receive an output. However, the critical part is that we are unaware of how the black-box produces the output. Since we cannot see inside the system, we need an approach that does not require this. Therefore, in a black-box analysis, we solve this issue by observing the machine on which the system is executed and collect measurements for the non-functional property we are interested in.

However, before we start analyzing the system, we first have to select the features we are interested in, since for most configurable systems it is not feasible to use the whole configuration space due to its size. This issue is called *combinatorial explosion* in Section 2.4.1.1 we explain how we deal with this problem.

After deciding which features are of interest to us, we can now turn to the question of how we analyze the system and collect the data we need to build a performance-influcence model.

As shown in Figure 2.3, we cannot analyze how the system produces the output; therefore, we are limited to the non-functional properties we can observe from the outside. For this reason, we execute the system with each configuration and measure the property we are interested in, such as energy consumption, memory usage, and computational resources used.

#### 2.4.1.1 *Challenges*

One of the larger problems we face when using black-box analysis is the issue of combinatorial explosion, which refers to the effect that when features increase linearly, the number of possible configuration increase exponentially [3].

Suppose we have a configurable system where each feature is a binary option. We also define that in this system, each feature is entirely independent of another (i.e., the system has no constraints, and selecting or deselecting one feature has no effect on other features). The number of unique configurations this system can produce is $2^n$, where 2 refers to the type of feature options allowed, binary in our case, and $n$ denotes the number of features.

The problem, is that all these different features can interact with each other in different ways, and for very small systems we can certainly brute force our way by benchmarking all possible configuration, however this does not scale. So, the brute force-method is not feasible for larger systems.

To illustrate the problem, the Linux kernel contains 10'000 different features [10], thus there are $2^{10000}$ possible configurations. It is estimated that the universe contains about $10^{79}$ atoms, which is still less than the number of unique configurations a system with 263 features produces. Such a system is already impossible to analyze using brute force, let alone the Linux kernel.

Hence, we cannot fully explore the entire configuration space and must select a subset representing the system with high accuracy. One way to solve this issue is to select different configurations from the configuration space using a sampling strategy. Whereas in this

work, we take advantage of the findings of Xu et al. [18], where they have shown that not all features are equally important and that up to 54,1% of features are rarely set by users. We use this information together with our domain knowledge to extract the most important features we are interested in.

### 2.4.2 *Multiple Linear Regression*

After we used the black-box analysis to collect measurements of the all the configuration are interested in, we use them to build our performance-influence model of the system. This section explains the reasoning behind using *multiple linear regression*. Afterwards, in Section 2.4.2.1, we explain *ordinary least squares*, an estimator to calculate the coefficient of each term inside the performance-influence model. When using *ordinary least squares* as an estimator, we have to handle *multicollinear features*. We explain why multicollinear features are a problem in Section 2.4.2.2 and how to reduce the influence of them by using *Variance Inflation Factor* in Section 2.4.2.3.

When building a performance-influence model from our black-box data we have would like our model to be interpretable. While multiple methods to predict performance have been introduced, such as neural networks, they lack the interpretability of the model, which on contrary *multiple linear regression* provides.

To illustrate why interpretability is important lets inspect the following performance-influence model:

$$\Pi_1 = -1000 + 1001 \cdot \mathit{Feature\_A} + 1002 \cdot \mathit{feature\_B} - 1000 \cdot \mathit{Feature\_A} \cdot \mathit{feature\_B}$$
$$\Pi_2 = 0 + 1 \cdot \mathit{Feature\_A} + 2 \cdot \mathit{feature\_B} + 0 \cdot \mathit{Feature\_A} \cdot \mathit{feature\_B}$$

Under the condition that either *Feature_A* or *Feature_B* needs to be selected, $\Pi_1$ and $\Pi_2$ predict for any configuration the same amount of time, however, if we take a close look at how $\Pi_1$ assigns the influence of each feature, we can see that this is not interpretable. Here $\Pi_1$ assigns to the *Base* feature an influence of -1000 thousand, which does not make sense since the time spent in the *Base* feature can not be negative. This also leads to $\Pi_1$ assigning unrealistic amounts of time to features or feature interactions.

To build the performance-influence model using the measurements from the black-box analysis we use the following formula for multiple linear regression for matrices [6]:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 ... \beta_n x_n + \epsilon \tag{2.3}$$
$$Y = X\beta + \epsilon$$

$Y = \mathit{Dependent\ variable}$
$X = \mathit{Independent\ variable}$
$\beta = \mathit{Regression\ coefficient}$
$\epsilon = \mathit{Error}$

In our case, $Y$ is a vector with $n$ elements containing the output of our black-box model, i.e., the measurements for each configuration in our set of configurations $\mathcal{C}$.

Our independent variable $X$ is an $n \times m$ matrix, where $n$ is the number of configurations used, and $m$ is the number of features and feature interactions across all configurations. To accommodate feature interactions in this linear model, we add a term for each interaction we want to include. For example, if we consider the interaction between features $x_i$ and $x_j$ we add the term $\beta_k x_i x_j$.

We are interested in the values of the coefficients $\beta$, since they quantify the influence of each feature or feature interaction on the whole system. In addition, $\beta_0$ denotes the intercept, representing the influence of the base code, meaning the part of the code executed regardless of the chosen configuration.

The value of each feature in the matrix is 1 if the feature is selected or 0 if it is not selected. If we have numerical features with $l$ different options, we split these features into $l$ binary features and encode them as an alternative group in our feature model.

All our measurements have a possible error represented by $\epsilon$ [7].

#### 2.4.2.1 *Ordinary Least Squares*

Now that we have seen the general formula of multiple linear regression and know what the different components stand for, we still need to figure out how to calculate the regression coefficient $\beta$ and the values that tell us the influence of each feature.

For this purpose, we use the ordinary least squares estimator, which is optimal for the class of linear unbiased estimators, but is unreliable when the independent variables $X$ contains a high degree of multicollinearity. The problem of multicollinearity is later explained in detail in Section 2.4.2.2. Ordinary least squares minimizes the sum of the squared residuals, where the residual is the difference between the predicted value of the estimator and the actual value [7].

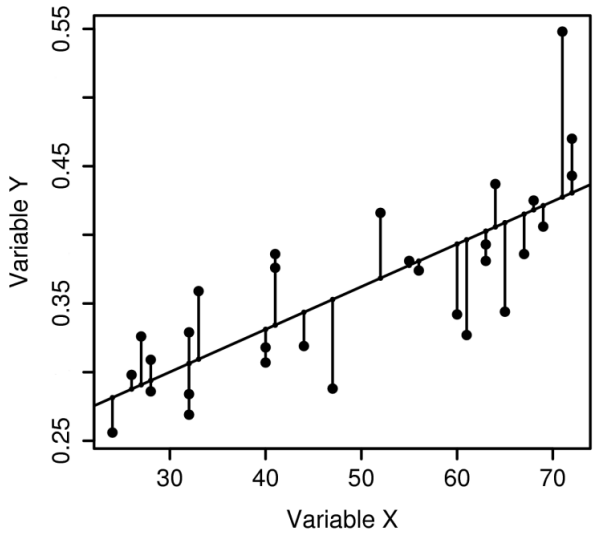


Figure 2.4: Ordinary Least Squares regression model residuals [2].

2 Visited at 06.03.2023, https://datajobs.com/data-science-repo/OLS-Regression-[GD-Hutcheson].pdf

We see an illustration of an ordinary least squares estimator for a linear regression model in Figure 2.4. In which we have only one variable X and the corresponding measurements Y, allowing us to compute the single regressor for this linear regression model.

To compute the regression coefficients using ordinary least squares, the following formula is used [7]:

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y \tag{2.4}$$

$\hat{\beta} = \textit{Ordinary Least Squares Estimator}$
$\top = \textit{Matrix Transposed}$

For us $\hat{\beta}$ contains all the regression coefficient we are interested in, i.e. the predicted time spent for each feature or feature interaction.

We proceed with an example of using ordinary least squares to build a performance-influence model. We select some configurations and use the black-box analysis to measure the time spend using these configurations. The results are in Table 2.1.

| Base | A | B | C | A ∧ B | A ∧ C | *measured* |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | **1** |
| 1 | 1 | 0 | 0 | 0 | 0 | **2** |
| 1 | 0 | 1 | 0 | 0 | 0 | **3** |
| 1 | 0 | 0 | 1 | 0 | 0 | **2** |
| 1 | 1 | 1 | 0 | 1 | 0 | **6** |
| 1 | 1 | 0 | 1 | 0 | 1 | **3** |
| 1 | 0 | 1 | 1 | 0 | 0 | **4** |
| 1 | 1 | 1 | 1 | 1 | 1 | **7** |

Table 2.1: Configuration samples of Listing 2.1, where *measured* is the time we measure using the selected features in that row.

Using the configuration samples from Table 2.2 we can now determine $X$ and $Y$:

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, Y = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 6 \\ 3 \\ 4 \\ 7 \end{bmatrix} \tag{2.5}$$

Using the ordinary least squares Equation 2.4.2.1 we obtain the following results:

$$\hat{\beta} = 1 + \begin{bmatrix} 0,00 \\ 1,00 \\ 2,00 \\ 1,00 \\ 2,00 \\ 0,00 \end{bmatrix} \tag{2.6}$$

All values have been rounded to 2 decimal places. We can see that all values have been assigned correctly.

In this example, we choose the configurations we want to analyze and measure the time spent for each configuration using the black-box analysis. We collect the results in Table 2.1. Then as our independent variable $Y$ we use the time *measured* for each configuration and for $X$ we use each feature or feature interaction we included in the configurations. Afterwards, we use the ordinary least squares formula of Equation 2.4.2.1 with $X$ and $Y$ to calculate the regression coefficients $\hat{\beta}$. Now we can use the values to create the performance-influcence model:

$$\Pi = 1 + 1 \cdot c(A) + 2 \cdot c(B) + 1 \cdot c(C) + 2 \cdot c(A) \cdot B + 0 \cdot c(A) \cdot c(C) \tag{2.7}$$

#### 2.4.2.2 *Multicollinear Features*

We already mentioned that ordinary least squares is optimal as long as our configurations do not contain multicollinearity. The reason for that is in presence of multicollinearity the variance of the estimator inflates, which in result hurts the interpretability of the model. We call features multicollinear when there exist a near linear dependency between these features, meaning we can nearly represent one feature as a combination of different features and the feature does only provide a small amount of new information to the system [7]. If a feature does not provide any new information to the system, then we speak of perfect multicollinearity. As an example take a look at a performance-influcence model for some the monthly expenses of a student, where *take_out* is included in the cost of *food*:

$$monthly\_expenses = food + take\_out$$

Now this shows multicollinearity between the features *food* and *take_out*, since *take_out* is already present in the cost of *food*, furthermore it is a case of perfect multicollinearity, since the feature does not provide any new information.

One way multicollinearity is introduced into a system is by using alternative groups, since the selection of a feature in the alternative group can be expressed by the combination of all other feature. [5]

| Base | A | B | C | $\Pi(*)$ |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | **5** |
| 1 | 0 | 1 | 0 | **10** |
| 1 | 0 | 0 | 1 | **15** |

Table 2.2: Configuration example illustrating multicollinearity in an alternative group, where $\Pi(*)$ is the predicted time for the selected feature inside the row.

Now consider the example presented in Table 2.2, where we see a configuration example that contains multicollinear features $A$, $B$, and $C$ due to an alternative group. The example contains a mandatory *Base* feature and 3 features that are in an alternative to each other. Now we can always model the presence of a feature in an alternative group due to the absence of other feature, here for feature $C$ to be selected, $B$ and $A$ needs to be deselected. This results in the following 3 performance-influcence models:

$$\Pi_0(c) = 0 + 5 \cdot c(A) + 10 \cdot c(B) + 20 \cdot c(C)$$
$$\Pi_1(c) = 5 + 10 \cdot c(A) + 5 \cdot c(B) + 20 \cdot c(C)$$
$$\Pi_2(c) = 8 + 20 \cdot c(A) + 10 \cdot c(B) + 7 \cdot c(C)$$

This leads to multiple performance-influcence models that are accurate with respect to the individual measurement, but make completely different statements when compared.

| $\Pi$ | Base | A | B | C | $\Pi(\{Base\})$ |
|---|---|---|---|---|---|
| $\Pi_0$ | 0 | 5 | 10 | 20 | **0** |
| $\Pi_1$ | 5 | 10 | 5 | 20 | **5** |
| $\Pi_2$ | 8 | 20 | 10 | 7 | **8** |

Table 2.3: Performance predictions of Table 2.2

The examples illustrated how multicollinearity is introduced when we use alternative groups. Therefore, when choosing the configuration space, this needs to be considered.

Another way multicollinearity is introduced into the system is to have features that are mandatory or connected by a condition. If we have features that are mandatory, we cannot distinguish these features with our black-box analysis because they are always selected together, and we cannot determine the extent to which each feature influences the system [5].

In Table 2.3, we see the predictions each of the 3 performance-influcence models make for the configurations $\{Base\}$. Now $\Pi_0$, $\Pi_1$, and $\Pi_2$, assign completely different values to the *Base* feature, which makes it impossible for, the user, to infer the correct value. The reason is that both *Base* and a feature of the alternative group are mandatory. Therefore, we cannot measure one without the presence of the other. Hence, the values of *Base* or the alternative group feature can be set in any ratio as long as the sum of the two values equals the measured time.

In this section, we learned about multicollinearity and how alternative groups and mandatory features introduce multicollinearity into a system. When we decide which features to use in our configuration set, we use our domain knowledge of the system to reduce multicollinear features to a minimum. To measure the degree of multicolinearity inside a system, we introduce the variance inflation factor in the following Section.

#### 2.4.2.3 *Variance Inflation Factor*

In reality, multicollinearity is often unavoidable in configurable systems, when we want to model the influence of a feature interaction between features $A$ and $B$ we introduce a term $A \cdot B$, this feature interaction is only selected when feature $A$ and $B$ are selected. Due to that we can not remove terms that introduce multicollinearity, however we can remove perfect multicollinear terms since they do not provide our system with new information.

Thus, we need a method to identify perfect multicollinear features to remove them. To check for perfect multicollinearity, we use the variance inflation factor (VIF), where a VIF factor of inf indicates that there is perfect multicollinearity in our system [5].

We compute the VIF score using the following equation:

$$VIF_j = \frac{1}{1 - R_j^2} \tag{2.8}$$

$$R_j^2 = 1 - \frac{\sum\limits_{\forall c \in \mathcal{T}} (c(o_j) - \overline{c}(o_j))^2}{\sum\limits_{\forall c \in \mathcal{T}} (c(o_j) - f_j(c \setminus o_j))^2} \tag{2.9}$$

Where $\mathcal{T}$ is the trainings set containing $j$ features $o_j$. The $\text{VIF}_j$ can be calculated for each feature by using the coefficient of determination $R^2$. To do this, we need to calculate $R_j^2$ for each feature $o_j$, fitting a linear regression function $f_j$ to predict whether $o_j$ is selected in the configuration $c \setminus o_j$, using all other features as predictors and the overall mean $\overline{c}(o_j)$ [5].

Using the VIF we can determine which features introduce multicollinearity into the system and to which degree they do. In the following Section we explain how we use the VIF score to identify perfect multicollinear feature in our system.

#### 2.4.2.4 *Deciding which terms by using VIF*

Multicollinearity is present in nearly every configurable system. Therefore, we cannot simply remove every term that introduces multicollinearity into the performance-influence model. So to handle multicolinearity, we need a strategy to determine the terms we must remove. For this, we use the VIF to identify the terms that add no new information, meaning the features that introduce perfect multicollinearity. To accomplish this, we use the following algorithm:

The Algorithm 1 takes as an input a *Model_to_check* with all the terms we want to check for multicollinearity. Afterwards, the algorithm for each *Term* in *Model_to_check*, add *Term* to our *Current_model* and check in Line 7 if the term introduced multicollinearity, which happens if one VIF value is inf, if this happens we remove *Term* from our *Current_model*. In the end, we return *Current_model*, containing all terms that do not introduce multicollinearity.

**Algorithm 1** Iterative VIF to check for perfect multicollinearity

```
1: Input: Model_to_check, list containing all the terms we want to check
2: Output: Current_model, list containing no terms that introduce perfect multicollinearity
3: Current_model ← [ ] \\Initialize empty list
4: Current_model.add(Model_to_check.pop())
5: for Term in Model_to_check do
6:     Current_model.add(Term)
7:     if ∞ in VIF(Current_model) then
8:       Current_model.remove(Terms)
9: return Current_model
```

In the end, we have a model that contains no perfect multicollinearity, meaning that each feature or feature interaction adds new information to our model and that none of our features is redundant. This increases the accuracy when using ordinary least squares as an estimator to build the performance-influence model.

## 2.5 WHITE-BOX ANALYSIS

A black-box analysis is very useful for systems where we do not have access to the source code, but has drawbacks. So, in the case where we have source code access, we should take advantage of this additional information to overcome some of the drawbacks. Analyses that incorporate source code information are usually referred to as white-box analyses.

In Section 2.1.1, introduce the general concepts behind white-box analysis and highlight current state-of-the-art strategies that employ white-box analysis in tho context of configurable software systems. Subsequently, we present VaRA, a white-box analysis framework that focuses on the analysis of configurable software systems. In Section 2.5.3, we explain how to build the performance-influence model using the white-box data.

### 2.5.1 *General Concept*

While a black-box analysis measures the time we spend executing the system from start to finish, a white-box analysis archives a more fine grained view by using different strategies to determine how much time we spend in each feature, some of which we further explain in Section 6.1.

For example, let us take a look at a white-box analysis that analyses *XZ* in Figure 2.5 and compare this to how a black-box analysis would work. Both analyses use different configurations containing the features and feature interaction we want to measure. The black-box analysis now would measure the time spent while the system is executed with the specific configuration. However, the white-box analysis instead uses a strategy to measure the time spent in specific features. Afterwards, the black-box analysis would use a model, like multiple linear regression, to infer the time spent in each feature. However, the white-box analysis does not need this since we already measured the time spent in each feature, which we then use to build the performance-influence model.

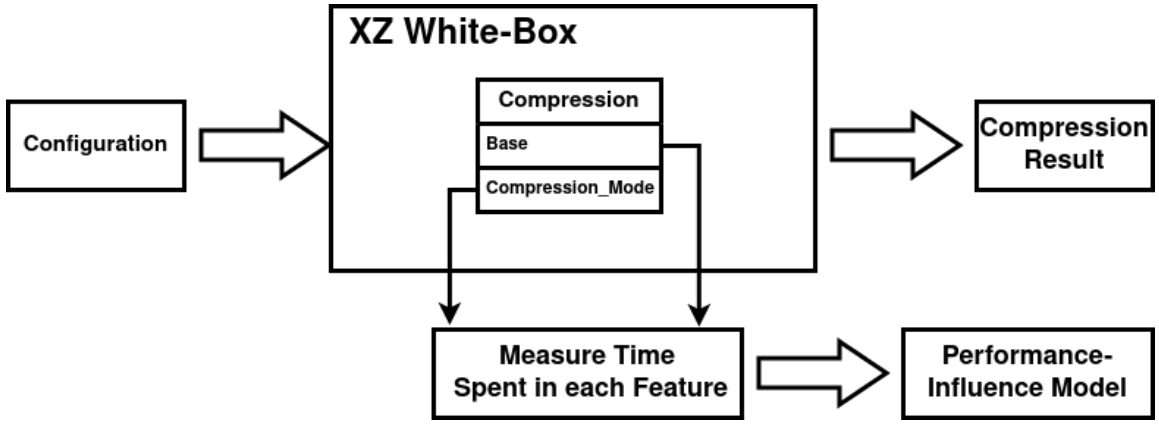


Figure 2.5: Process of using a white-box analysis to build a performance-influence model for *XZ*.

### 2.5.2 *VaRA*

To analyze the configurable software system we are interested in, we use VaRA, a framework that is built on LLVM. In addition, we use the VaRA Tool Suite[3], which provides us with a framework that supports us when analyzing configurable software systems with VaRA.

The purpose of VaRA is to provide various analyses for systems where the user only needs to focus on the high-level conceptual information of the system, while VaRA handles the low-level-details. Since VaRA is built on top of LLVM, it is able to analyze systems written in languages that can be compiled by LLVM, such as C, C++ or Rust [11].

We use VaRA as the core of our white-box analysis to measure the time spent inside each feature and feature interaction, from which we then build the performance-influence model.

#### 2.5.2.1 *Feature Region*

To analyze configurable systems, VaRA identifies code regions associated with a feature or feature interaction. VaRA refers to such code regions that are influenced by feature decisions as *feature region*. The first step for VaRA to be able to detect these regions is to find the feature variable that represents the features inside the code and mark them as feature variables. A feature region is, therefore, a part of the code that is executed depending on the value of the feature variable. Whenever we detect a feature region, we inject code into the system to measure the time spent in these regions. Therefore, after detection every feature region, the whole code is instrumentalized by VaRA to measure the time spent in each feature region.

3 Visited at 14.03.2022 https://vara.readthedocs.io

```
1 void encrypt() {
2     bool Encryption; //Feature Variable
3     assign_feature(Encryption); //Assigns true if Encryption is selected
4
5     if(Encryption)
6         foo();
7     else
8         bar();
9 }
```

Listing 2.2: Feature region example. The feature variable is highlighted in orange and the feature region is highlighted in red.

To illustrate this, we design a small function that encrypt files depending on if the encryption feature is selected. In Listing 2.2, we can see the structure of the *Encryption* feature region. The feature variable in Line 2 represents whether *Encryption* is selected or deselected and so, depending on *Encryption*, either the *then* or *else* branch is executed. Together, these two branches form an *Encryption* feature region.

#### 2.5.2.2 *Taint Analysis*

Before explaining VaRA feature region detection, we explain the concept behind a taint analysis since both approaches build upon this analysis.

The common usage of a taint analysis is in cybersecurity, where we trace the data flow of data that originates from an outsider or an untrusted source. We track how the data is propagated through the system until it reaches a point where it is again accessible to the outside. We call the access where the data is injected *sources*, and the points where the data is exposed to the outside *sinks* [12].

VaRA uses taint analysis, too however, in contrast to the common usage of a taint analysis, we are not interested in finding the sinks where data is leaked to the outside. However, instead, we are interested whenever instructions access our feature. For the taint analysis, VaRA uses feature variables as sources and instruction as sinks. Whenever an instruction accesses a feature variable, this instruction is tainted by that feature [9].

DOMINATOR APPROACH We call the second approach the *Dominator Approach*, in here, VaRA uses domination relationships to identify feature regions.

To do this, VaRA works with basic blocks of the control flow graph, whereas a basic block is an instruction sequence that contains an entry label, which is the entry point for the code, and a terminator at the end, which determine the control flow of the block. An example of a terminator is an if condition that uses a feature variable.

To discover these domination relationships, VaRA checks out which basic block dominates other basic blocks with dependent terminator instructions. A basic block $BB_1$ dominates a different basic block $BB_2$ when the terminator of $BB_1$ decides if $BB_2$ is executed. Now instructions in $BB_2$ would depend on the terminator instruction of $BB_1$. The feature for the feature region of $BB_2$ is the feature that corresponds to the feature variable used by the terminator of $BB_1$ [19].

#### 2.5.2.3 *Locating feature variables*

VaRA is not able to automatically detect which variables represent features. Therefore, we need to provide VaRA with information about the location of feature variables. One way to do this is by giving VaRA a feature model as a XML file containing every feature's location inside the code.

```
1  <locations>
2      <sourceRange category="necessary">
3          <path>src/my_encryption.c</path>
4          <start>
5              <line>2</line>
6              <column>10</column>
7          </start>
8          <end>
9              <line>2</line>
10             <column>19</column>
11         </end>
12         </sourceRange>
13 </locations>
```

Listing 2.3: Feature model of Listing 2.2 in XML. The start of a feature variable is highlighted in red and the end is highlighted in green.

In Listing 2.3 we show how we encode the *Encryption* feature from Listing 2.2 as a feature variable. To do so we specify different tags inside the *XML*. The location tag contains, at least one <*sourceRange*> tag, in which we specify the feature variable that is associated with the feature, however a location can contain multiple source ranges, whereas the feature is implemented by multiple feature variables. Each <*sourceRange*> tag contains a <*path*> tag that specifies the location of the file containing the feature variable. After specifying the path, we need even further to specify the location of the feature variable inside the file. For this, we see in Line 4 the <*start*> tag and in Line 8 the <*end*> tag, inside both, we specify the <*line*> and <*column*> for where the feature variable starts end ends.

For Listing 2.2, we would specify that the feature variable *Encryption* is in Line 2 and begins in *column 10* and ends in *column 19*.

#### 2.5.2.4 *VaRA Tool Suite*

We use VaRA in combination with the VaRA Tool Suite, a framework written in python that assists us when analyzing configurable software systems using VaRA by specifying an *experiment* to analyze a particular *project*. The result our analysis produces is written into a *report*. We emphasize that both experiments and projects are independent, allowing us to use different experiments to analyze a project in different ways. To start the analysis, we use the command-line tool *vara-run*, which specifies which experiments we want to run for which project.

### 2.5.3 *Trace Event Format*

When we use VaRA the code of the configurable software system gets instrumented to measure the time spent inside each feature region. All these measurements are collected in a trace event format (TEF) [4] report, which we use to calculate the overall time spent in each feature and feature interaction to build the performance-influcence model.

Every time we enter or leave a feature region, a trace event is triggered that contains the following information:

```
"name": "Base",
"cat": "Feature",
"ph": "B",
"ts": 0,
"pid": 726119,
"tid": 726119,
"args": {
    "ID": 0
}
```

Listing 2.4: Example of a feature region trace entry in the TEF file

In Listing 2.4, we see how a trace event is structured. Each trace event contains a *name* that refers to the names of the features that affect this region, *ph* represents the event type, while *B* signals the beginning and *E* the end of an event. All events contain a timestamp *ts* that refers to the time in milliseconds when the event began or ended. *Args* contain an *ID* that refers to each event, so the event that initiates the beginning *E* of a region has the same *ID* as the event that signals when we leave the region with *E*.

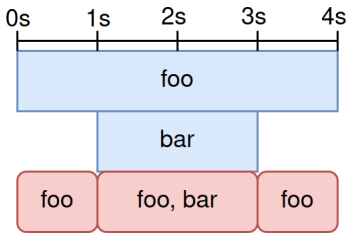


Figure 2.6: Example of two features interaction. Features are highlighted in blue and all the current active features are highlighted in red.

When a trace event for a feature region begins before a trace event for a different feature ends, we say these features interact. As an example in Figure 2.6, we have two features, *foo* and *bar*, where the trace event of *foo* starts at *0* and ends at *4* seconds and the trace event of *bar* starts at *1* and ends at *3* seconds. We spent *2* seconds in feature *foo*, from *0 to 1* and *3 to 4*, but since we did not leave the feature region *foo* before entering *bar* we have an interaction between these features. Therefore, we spent *2* seconds inside the feature interaction *(foo, bar)* instead of the feature *bar*.

4 Visited at 15.03.2022
https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU

Since we are only interested in which features interact, we ignore the order in which the interaction happens, which means the previous interaction of *(foo, bar)* is the same as *(bar, foo)*. This also makes sense in the context of performance-influence model since we defined the interaction of features as a product, where the time spent in that feature interaction is added if all features of that interaction are selected, in this case, $2 \cdot c(foo) \cdot c(bar)$. Since the product is commutative, we also ignore the order in which the features interact inside the performance-influence model.

When we have nested trace events of the same feature, we do not add a feature interaction between the same features, due to the reason that inside the performance-influence model the feature interaction $2 \cdot c(foo) \cdot c(foo)$ is the same as $2 \cdot c(foo)$.

After the system finishes its execution, we have to transform the TEF report into a performance-influence model by aggregating all events. To do so, we sum up all the time spent in each region and attribute this time to the feature that influenced that region.

We calculate the time spent on each feature as follows:

$$\text{time(feature)} = \sum_{\text{event} \in \text{feature}} \left( \sum_{(ts_E, ts_B) \in \text{event}} ts_E - ts_B \right) \tag{2.10}$$

$$\text{feature\_coefficients} = \sum_{\text{feature} \in \text{TEF Report}} \text{time(feature)} \tag{2.11}$$

In Equation 2.10, we calculate the time spent for the given feature. For this equation, we define that a *feature* as a list of *events*, whereas each event is a pair of trace events representing when the feature region is entered and when it is left. To calculate the time spent in the region, we compute $ts_E - ts_B$.

Equation 2.11 is a sequence of the total time spent in each feature or feature interaction we measured in the *TEF Report*. Now that we obtained all the coefficients that represent the influence of each feature and feature interaction, we use them to build the performance-influence model. By that, we build up a performance-influence model from feature specific white-box performance measurements.

# 3

# METHODOLOGY

In this chapter, we introduce our research questions. Furthermore, we formalize how we evaluate these questions and explain how we measure and collect data for our evaluation. We then proceed in Section 3.3 to establish a ground truth, which tests both white-box analysis and black-box analysis in different scenarios to test both analyses and identify weaknesses.

## 3.1 RESEARCH QUESTIONS AND OPERATIONALIZATION

In the following, we introduce our research questions and formalize how we evaluate them. The goal of this work is to work out the differences between white-box and black-box models when analyzing the feature performance of configurable software systems. To quantify these differences, ask:

**RQ1**: How accurately do white-box and black-box models detect features and feature interactions?

Due to the sheer number of features, it is hard to know the influence each feature or feature interaction has on a configurable system. Therefore, when we use white-box or black-box analysis to quantify the influence of these features and feature interaction, we are interested in the accuracy of both analyses. To answer this, we research how accurately they can identify the influence of features and feature interactions.

To investigate this we design multiple small configurable systems in the ground truth section. We use the ground truth systems for this because, in these systems, we are fully aware of how much time we should spend in each feature or feature interaction. Thus, we quantify the difference between the predicted and true influence of each feature $f$ in the set of features and feature interactions $F$ by calculating the *error rate* and afterwards its mean, following the approach outlined in [8].

$$error_f = \frac{|true_f - predicted_f|}{true_f} \tag{3.1}$$

$$\overline{error} = \frac{\sum_{f \in F} error_f}{|F|} \tag{3.2}$$

Note that $true_f$ is the true influence of the feature or feature interaction $f$ that we obtain from the baseline of our ground truth and $predicted_f$ is the influence predicted by the performance-influence models we build. The closer $\overline{error}$ is to 0, the better the prediction, with an $\overline{error}$ of 0 indicating a perfectly accurate performance-influence model.

Another, point of interest is how similar the performance-influence models are, we anwser this in the following research question.

**RQ2**: Do performance models created by our white-box and black-box attribute the same influence to each feature?

Another point we are interested in is to know if the performance-influence models of both analyses are similar. This is of interest because, similar analyses that we can use both analyses interchangeably for one another.

To answer this question, we investigate whether the performance-influence models build-out of the white-box and black-box analyses' data agree, i.e. they predict the same influence for each feature. Compared to RQ1, we do not measure if the predictions are accurate regarding the true influence but if both analyses produce similar results. To quantify the difference, we again use the mean error rate. However, we adjust the formulas as follows:

$$similarity_f = \frac{|predicted_{f,Model_1} - predicted_{f,Model_2}|}{predicted_{f,Model_1}} \tag{3.3}$$

$$\overline{similarity_f} = \frac{\sum_{f \in F} error_f}{|F|} \tag{3.4}$$

Here we quantify the similarity between the predictions of models for each feature $f$. Since the similarity between white-box and black-box model is not the same as the similarity between black-box and white-box model, we have to calculate it for both models.

## 3.2 COLLECTING DATA

Now that we have defined how we answer the research questions, we explain on how we collect the data using both analyses.

The non-functional property that we decide to analyze is runtime, due to it being a quantifiable metric that reflects the changes to the system if a feature with a significant influence is selected.

When measuring with either white-box or black-box analysis, each measurement is subject to noise, which influences the performance during the execution of the system. To reduce this noise, we follow the suggestions made by Arcuri et.al [4], measure each configuration 30 times, and take the mean of the time spent as values to build the performance-influence model.

We collect the data for each analysis using the VaRA Tool Suite, in which we define an experiment for each project. For our white-box analysis, we write an experiment that specifies that we want to use VaRA to instrument our code so that during execution, all trace events can be tracked and written into the TEF report file. For our black-box analysis, we wrap the command we want to measure using the Linux `time` command and write the time spent into a different report.

All the measurements are performed on the same server node. To minimize background noise, we ensured that no other job is executed during the measurement. The server node contains an AMD EPYC 72F3 8-Core Processor with 16 threads and 256 GB RAM.

## 3.3 GROUND TRUTH

In this section, we introduce our ground truth to establish that both black-box analysis and white-box analysis can analyze simple, configurable systems and identify weaknesses.

To do so, we design multiple configurable systems that test both analyses in different scenarios. Afterward, we evaluate as described in Section 3.1. Since we design these systems ourselves, we have a baseline that we use to build a performance-influence model for each system manually. We design all the systems with different focuses in mind, such as a system that includes multicollinearity, to see to what extent they influence the analyses.

Each system uses *Simple Interaction* as the base system, however, they extend or change the system in different ways.

**SIMPLE INTERACTION** For our first system, we use the code from our previous example in Listing 2.1 and provide an additional feature model of the system in Figure 3.1. Since we are going to use this as our base system, extending as needed to fit each particular scenario, it is kept intentionally simple, only containing some interactions and no constraints.

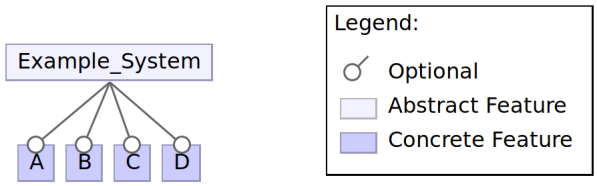


Figure 3.1: Feature model of Listing 2.1.

**ELSE CLAUSE** The first modification we do is adding an *else clause* to the if statement of feature *A* in Line 6. We encode this as follows:

```
1 if(A)
2     fpcsc::sleep_for_secs(1);
3 else
4     fpcsc::sleep_for_secs(2);
5
```

We expect the white-box analysis to attribute the time spent in the else case to feature *A* since it is a part of the feature region of feature *A*. In contrast, the black-box analysis cannot differentiate between the else clause that is executed when feature *A* is unselected and the time spent in *Base*.

**FUNCTION** The next modification of our basic system is adding a function to the system in which we spent time. In Line 7 we call the function *waste_time(1)* instead of *fpcsc::sleep_for_secs(1)*, in which we spent time a second.

```
1 void waste_time(int duration){
2     fpcsc::sleep_for_secs(duration);
3 }
```

Both white-box and black-box analysis should be able to still correctly attribute the time spent in each feature.

MULTICOLLINEARITY Instead of modifying the system's code, we add a restriction to the feature model of Figure 3.1. We change feature *B* from optional to mandatory. This introduces multicollinearity into the system. The white-box analysis should still be able to identify correctly the time spent in *Base* and feature *B*, while the black-box analysis should distribute the time spent between *Base* and *B* differently.

SHARED FEATURE VARIABLE We now add an optional feature *E* to the system. However, instead of encoding it like the previous features by specifying a separate variable, we modify the feature variable *D* in Line 2 to a `string de = "00"` for which the first character represents feature *D* and the second character *E*. If either is selected, we assign the character for that feature *"1"*. We add another if statement to spend time if *E* is selected and encode this as follows:

```
1 std::string de = "00";
2
3 if(de[0] == '1')
4     fpcsc::sleep_for_secs(2);
5
6 if(de[1] == '1')
7     fpcsc::sleep_for_secs(1);
```

We expect the black-box analysis to still work as intended, but the white-box analysis to be inaccurate since both features *D* and *E* share the same feature variable. Each feature region of *D* or *E* should now be influenced by the feature interaction *{D, E}*.

# 4 EXPERIMENT

This chapter describes the main experiment that is conducted to evaluate both analyses.

## 4.1 RESEARCH SETUP

The configurable system we analyze is the open source software system XZ, a command line tool written in C and a component of XZ UTILS[1]. The primary compression algorithm that XZ currently uses to perform lossless data compression is LZMA2. We use XZ (XZ UTILS) version 5.5.0 with the commit hash "610dde15a88f12cc540424eb3eb3ed61f3876f74".

The file we compress is a geographic map encoded as a JSON file of size 203 MB[2], which ensures that we spend a significant amount of time so that all features can be measured.

### 4.1.1 *Configuration Space*

Due to the challenge of combinatorial explosion explained in Section 2.4.1.1. We have to select a subset of features XZ offers to still measure all possible configurations and ensure the accuracy of the black-box analysis.

We selected three core features of XZ that should significantly influence the system's performance: two numeric features, *compression level* and *threads*, and one binary feature, *extreme*. The value of *compression level* ranges between *0* and *9* whereas *threads* can either be *0*, *1*, *2*, *4* or *8*. The feature *compression level* sets the compression preset level, which influences the compression ratio. This increases the necessary memory needed during compression and decompression, as well as the compression speed. This feature is encoded as an alternative group. When *extreme* is enabled LZMA uses a slower variant of the selected compression preset to achieve a potentially better compression ratio. This feature is optional. The feature *threads* specifies the number of worker threads to use, whereas the option 0 makes XZ use as many threads as there are cores available. Multithreading allows *XZ* to split the input into different blocks and compress them independently of another. This feature is optional, encoded as an alternative group.

The number of configurations we can build using these three features is calculated as follows:

$$|C| = |\textit{compression level}| \cdot |\textit{extreme}| \cdot |\textit{threads}| \tag{4.1}$$
$$100 = 10 \cdot 2 \cdot 5$$

We select the features *compression level 1* and *threads 0* as our base feature since it represents the least amount of time XZ spends for compression.

1 Visited at 02.04.2023 https://tukaani.org/xz/
2 Visited at 03.04.2023 https://github.com/simonepri/geo-maps/releases/latest/download/countries-land-1m.geo.json

# 5 EVALUATION

This chapter evaluates the thesis core claims. We begin with presenting the results for both the ground truth systems and XZ in Section 5.1. Afterward in Section 5.2 we discuss the results. At the end in Section 5.3, we explain the threads of validity for the results.

## 5.1 RESULTS

In this section, we present the results of the ground truth and of XZ.

### 5.1.1 *Ground Truth Results*

We now proceed with evaluating the different systems, SIMPLE INTERACTIONS, ELSE CLAUSE, FUNCTION, MULTICOLLINEARITY and SHARED FEATURE VARIABLE, that we introduce in Section 3.3

*Simple Interaction Results*

We start by constructing the baseline performance-influence model for this system as shown in Equation 2.2 and build the performance-influence model for the black-box analysis:

| $\Pi_{BB}$ | Base | A | B | C | D | A ∧ B | C ∧ D |
|---|---|---|---|---|---|---|---|
| Simple Interaction | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 2.0 | 0 |

Table 5.1: performance-influence model for simple interaction using black-box analysis

For the white-box analysis out with Listing 2.1 and Figure 3.1 we build Listing A.1 in which we declare all the feature variables of SIMPLE INTERACTIONS. We analyse Listing 2.1 using our white-box analysis, this produces a TEF report file in Listing A.2, afterward we use this report to calculate the time spend in each feature region described as in Equation 2.11 and Equation 2.10. This produces the following file in Listing A.3, from which we build the following performance-influence model:

| $\Pi_{WB}$ | Base | A | B | C | D | A ∧ B | C ∧ D |
|---|---|---|---|---|---|---|---|
| Simple Interaction | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 2.0 | 0 |

Table 5.2: performance-influence model for simple interaction using white-box analysis

We group our results in Table 5.3 to compare them with each other.

| $\Pi$ | Base | A | B | C | D | A $\wedge$ B | C $\wedge$ D |
|---|---|---|---|---|---|---|---|
| Baseline | 2 | 1 | 2 | 1 | 2 | 2 | 0 |
| Black-box | 2 | 1 | 2 | 1 | 2 | 2 | 0 |
| White-box | 2 | 1 | 2 | 1 | 2 | 2 | 0 |

Table 5.3: Direct comparison between the baseline, black-box and white-box performance-influence model for Listing 2.1

Out of Table 5.3 we now calculate $\overline{error}$ and for both white-box and black-box:

| RQ1 | $\overline{error}$ |
|---|---|
| Black-box | 0 |
| White-box | 0 |

Table 5.4: Results RQ1

| RQ2 | *similarity* |
|---|---|
| Black-box$_{WB}$ | 0 |
| White-box$_{BB}$ | 0 |

Table 5.5: Results RQ2

*Else Clause*

The results for the Else Clause system are as follows

## 5.2 DISCUSSION

In this section, discuss your results.

## 5.3 THREATS TO VALIDITY

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

# 6 RELATED WORK

## 6.1 STRATEGIES

When analyzing systems using a white-box approach, different strategies have been introduced. In this chapter, we explain the strategies the tools, ConfigCrusher and Comprex use, both model configurability on a feature level, whereas Weber et al. introduce a strategy that models configurability on a method level.

Velez et al. introduced us to ConfigCrusher [15], a white-box analysis that uses static data-flow analysis to see how features influence variables and the control flow of the system. In addition, ConfigCrusher leverages three insights about configurable systems from previous works, namely irrelevance, orthogonality, and low interaction degree. They use irrelevance to identify features relevant to the system's data flow, reducing the number of configurations required to analyze the system. They use orthogonality to identify features that do not interact with each other and, therefore, can be measured together. Since only a few features interact, ConfigCrusher focuses on the configurations with interacting features to reduce the number of configurations to be analyzed. From these findings, two techniques are developed, namely compression and composition. They use compression to reduce the number of configurations required to analyze the system by simultaneously analyzing regions that are independent of each other so that they can use a single configuration to analyze different features. Whereas composition takes advantage of the fact that performance-influence model can be built compositionally by building a performance-influence model for each region separately and then assembling all local performance-influence model into one model for the entire system. After using the data-flow analysis to generate a control flow graph and a statement influence map, which maps statements to the configuration options that influence that statement. Afterward, they use both the control flow graph and statement influence map to instrument the regions in the system that correspond to features and execute the instrumented system to track execution time of each feature. From these measurements, they build the performance-influence model for the system.

Velez et al. introduced Comprex [16], an approach that builds on ConfigCrusher but uses an iterative dynamic taint analysis instead of static analysis to determine how and to what extent features affect the control flow of the given system. By doing so, they identify which code regions are influenced by which configurations and, during execution measure the time spent in these regions to then build the performance-influence model.

Compared to ConfigCrusher and Comprex, Weber et al. [17] uses a profiling approach to generate performance-influence models that analyze configurability on a method level. To achieve this, they first used JProflier, a coarse-grained profiler, to learn a performance-influence model for every method that has been learned successfully. To identify the hard-to-learn methods, they use filtering techniques and then KIEKER, a fine-grained profiler, to

learn these methods. At the end, for each method, they obtain a performance-influcence model that shows how strong each feature influences the performance of that method.

# 7 CONCLUDING REMARKS

## 7.1 CONCLUSION

## 7.2 FUTURE WORK

# APPENDIX

Listing A.1: Feature model XML of Figure 3.1 for Listing 2.1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vm SYSTEM "vm.dtd">
<vm name="SimpleSystem" root="root">
    <binaryOptions>
        <configurationOption>
            <name>Example_System</name>
            <parent></parent>
            <optional>False</optional>
            <locations>
                <sourceRange category="necessary">
                    <path>src/simpleSystem.cpp</path>
                    <start>
                        <line>2</line>
                        <column>10</column>
                    </start>
                    <end>
                        <line>2</line>
                        <column>10</column>
                    </end>
                    </sourceRange>
            </locations>
        </configurationOption>
        <configurationOption>
            <name>B</name>
            <parent>Example_System</parent>
            <optional>True</optional>
            <locations>
                <sourceRange category="necessary">
                    <path>src/simpleSystem.cpp</path>
                    <start>
                        <line>2</line>
                        <column>13</column>
                    </start>
                    <end>
                        <line>2</line>
                        <column>13</column>
                    </end>
                    </sourceRange>
            </locations>
        </configurationOption>
        <configurationOption>
            <name>C</name>
```

```
            <parent>Example_System</parent>
            <optional>True</optional>
            <locations>
                <sourceRange category="necessary">
                    <path>src/simpleSystem.cpp</path>
                    <start>
                        <line>2</line>
                        <column>16</column>
                    </start>
                    <end>
                        <line>2</line>
                        <column>16</column>
                    </end>
                    </sourceRange>
            </locations>
        </configurationOption>
        <configurationOption>
            <name>D</name>
            <parent>Example_System</parent>
            <optional>True</optional>
            <locations>
                <sourceRange category="necessary">
                    <path>src/simpleSystem.cpp</path>
                    <start>
                        <line>2</line>
                        <column>19</column>
                    </start>
                    <end>
                        <line>2</line>
                        <column>19</column>
                    </end>
                    </sourceRange>
            </locations>
        </configurationOption>
    </binaryOptions>
    <numericOptions></numericOptions>
    <booleanConstraints/>
</vm>
```

Listing A.2: Trace event format report generated by VaRA of Listing 2.1


```
{
    "traceEvents": [
        {
            "name": "Base",
            "cat": "Feature",
            "ph": "B",
            "ts": 0,
            "pid": 343917,
            "tid": 343917,
            "args": {
                "ID": 0
            }
        },
        {
            "name": "FR(FeatureA)",
            "cat": "Feature",
            "ph": "B",
            "ts": 2000120,
            "pid": 343917,
            "tid": 343917,
            "args": {
                "ID": 1729382256910270467
            }
        },
        {
            "name": "FR(FeatureA)",
            "cat": "Feature",
            "ph": "E",
            "ts": 3000216,
            "pid": 343917,
            "tid": 343917,
            "args": {
                "ID": 1729382256910270467
            }
        },
        {
            "name": "FR(FeatureB)",
            "cat": "Feature",
            "ph": "B",
            "ts": 3000217,
            "pid": 343917,
            "tid": 343917,
            "args": {
                "ID": 1729382256910286851
            }
        },
        {
            "name": "FR(FeatureB)",
            "cat": "Feature",
```

```
                "ph": "E",
                "ts": 5000300,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910286851
                }
            },
            {
                "name": "FR(FeatureC)",
                "cat": "Feature",
                "ph": "B",
                "ts": 5000301,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910303235
                }
            },
            {
                "name": "FR(FeatureC)",
                "cat": "Feature",
                "ph": "E",
                "ts": 6000407,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910303235
                }
            },
            {
                "name": "FR(FeatureD)",
                "cat": "Feature",
                "ph": "B",
                "ts": 6000407,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910319619
                }
            },
            {
                "name": "FR(FeatureD)",
                "cat": "Feature",
                "ph": "E",
                "ts": 8000517,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910319619
                }
```

```
            },
            {
                "name": "FR(FeatureA)",
                "cat": "Feature",
                "ph": "B",
                "ts": 8000518,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910336003
                }
            },
            {
                "name": "FR(FeatureB)",
                "cat": "Feature",
                "ph": "B",
                "ts": 8000518,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910352387
                }
            },
            {
                "name": "FR(FeatureA)",
                "cat": "Feature",
                "ph": "E",
                "ts": 10000619,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910336003
                }
            },
            {
                "name": "FR(FeatureB)",
                "cat": "Feature",
                "ph": "E",
                "ts": 10000620,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910352387
                }
            },
            {
                "name": "FR(FeatureC)",
                "cat": "Feature",
                "ph": "B",
                "ts": 10000620,
                "pid": 343917,
```

```
                "tid": 343917,
                "args": {
                    "ID": 1729382256910368771
                }
            },
            {
                "name": "FR(FeatureD)",
                "cat": "Feature",
                "ph": "B",
                "ts": 10000621,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910385155
                }
            },
            {
                "name": "FR(FeatureC)",
                "cat": "Feature",
                "ph": "E",
                "ts": 10000641,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910368771
                }
            },
            {
                "name": "FR(FeatureD)",
                "cat": "Feature",
                "ph": "E",
                "ts": 10000641,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 1729382256910385155
                }
            },
            {
                "name": "Base",
                "cat": "Feature",
                "ph": "E",
                "ts": 10000642,
                "pid": 343917,
                "tid": 343917,
                "args": {
                    "ID": 0
                }
            }
        ],
        "stackFrames": {},
```

```
203         "timestampUnit":  "us"
204     }
```

Listing A.3: Aggregated time spent in each feature region of Listing A.2


```
{
    "Base": {
        "Occurrences": 8.0,
        "Overall Time": 2000.12,
        "Mean": 250.015,
        "Variance": 437546001.2135,
        "Standard Deviation": 661.472600500958
    },
    "Base,FeatureC": {
        "Occurrences": 4.0,
        "Overall Time": 1000.115,
        "Mean": 250.02875,
        "Variance": 187542127.3661875,
        "Standard Deviation": 433.061343652591
    },
    "Base,FeatureA": {
        "Occurrences": 2.0,
        "Overall Time": 1000.114,
        "Mean": 500.057,
        "Variance": 250056003.136,
        "Standard Deviation": 500.056
    },
    "Base,FeatureB": {
        "Occurrences": 2.0,
        "Overall Time": 2000.109,
        "Mean": 1000.0545,
        "Variance": 1000107002.86225,
        "Standard Deviation": 1000.0535
    },
    "Base,FeatureD": {
        "Occurrences": 1.0,
        "Overall Time": 2000.13,
        "Mean": 2000.13,
        "Variance": 0.0,
        "Standard Deviation": 0.0
    },
    "Base,FeatureA,FeatureB": {
        "Occurrences": 1.0,
        "Overall Time": 2000.135,
        "Mean": 2000.135,
        "Variance": 0.0,
        "Standard Deviation": 0.0
    },
    "Base,FeatureC,FeatureD": {
        "Occurrences": 1.0,
        "Overall Time": 0.018,
        "Mean": 0.018,
        "Variance": 0.0,
        "Standard Deviation": 0.0
```

```
    },
    "Overall time for all features": {
        "Time Taken": 10000.741
    },
    "Number of Repetitions": {
        "Repetitions": 1
    }
}
```


| | base | a | b | c | d | a, b | c, d |
|---|---|---|---|---|---|---|---|
| GTBasic | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 2.0 | 3.108624e-15 |

| | base | a | b | c | d | a, b | c, d |
|---|---|---|---|---|---|---|---|
| GTFunction | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 2.0 | 3.108624e-15 |

| | base | a | b | c | d | a, b | c, d |
|---|---|---|---|---|---|---|---|
| GTElse | 4.00025 | -1.0 | 2.000333 | 0.999667 | 1.999667 | 1.999667 | 0.000333 |

| | base | a | b | c | d | c, d |
|---|---|---|---|---|---|---|
| GTMulticollinearity | 4.0 | 3.0 | 2.220446e-16 | 1.0 | 2.0 | 0.0 |

| | base | a | b | c | d | e | a, b | c, d |
|---|---|---|---|---|---|---|---|---|
| GTSharedFeature | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | 1.0 | 2.0 | -2.109424e-15 |

| | Base | threads_1 | threads_2 | threads_4 | threads_8 |
|---|---|---|---|---|---|
| Base | 0.759000 | 6.405667 | 2.844667 | 1.119333 | 0.250667 |
| compression_level_1 | 0.271000 | 2.441667 | 1.078667 | 0.390000 | 0.108333 |
| compression_level_2 | 1.066000 | 9.904000 | 4.148333 | 1.741000 | 0.531000 |
| compression_level_3 | 2.757333 | 20.859333 | 9.201333 | 3.803000 | 1.128333 |
| compression_level_4 | 4.440667 | 28.496333 | 12.312667 | 4.763667 | 0.983000 |
| compression_level_5 | 7.546333 | 47.278667 | 19.124667 | 7.207667 | 1.250333 |
| compression_level_6 | 7.811000 | 49.184333 | 19.872667 | 7.491333 | 1.352667 |
| compression_level_7 | 17.261333 | 53.346333 | 16.777000 | 0.878000 | -0.227333 |
| compression_level_8 | 41.542667 | 40.312333 | -2.765000 | -1.131667 | -0.308667 |
| compression_level_9 | 91.876667 | -0.393667 | -2.575667 | -1.177333 | -0.180667 |

| | Base | threads_1 | threads_2 | threads_4 | threads_8 |
|---|---|---|---|---|---|
| Base | 2.743667 | 28.790333 | 12.409667 | 4.855667 | 1.124333 |
| compression_level_1 | 0.510333 | 35.168667 | 15.168333 | 6.246333 | 1.914000 |
| compression_level_2 | 0.617667 | 37.456000 | 16.133333 | 6.649000 | 1.978000 |
| compression_level_3 | 1.029667 | 40.189333 | 17.567000 | 7.193000 | 2.171000 |
| compression_level_4 | -0.563667 | 39.905667 | 16.609333 | 5.804667 | 0.680333 |
| compression_level_5 | -2.245667 | 48.860333 | 18.286667 | 5.505000 | -0.863000 |
| compression_level_6 | -2.339000 | 50.476667 | 18.719667 | 5.551667 | -0.966667 |
| compression_level_7 | -1.485333 | 56.680000 | 16.838333 | -0.346667 | -1.753333 |
| compression_level_8 | 0.835667 | 45.689333 | -1.603333 | -0.290000 | 0.469667 |
| compression_level_9 | 8.915333 | 9.152667 | 5.483667 | 7.371000 | 8.297667 |

| | Base | FeatureA | FeatureC | FeatureD | FeatureB | FeatureA ∧ FeatureB |
|---|---|---|---|---|---|---|
| {} | 2.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| {A} | 2.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| {A, B} | 2.0 | 1.0 | 0.0 | 0.0 | 2.0 | 2.0 |
| {A, B, C} | 3.0 | 1.0 | 0.0 | 0.0 | 2.0 | 2.0 |
| {A, B, C, D} | 2.0 | 1.0 | 1.0 | 2.0 | 2.0 | 2.0 |
| {A, B, D} | 2.0 | 1.0 | 0.0 | 2.0 | 2.0 | 2.0 |
| {A, C} | 3.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| {A, C, D} | 2.0 | 1.0 | 1.0 | 2.0 | 0.0 | 0.0 |
| {A, D} | 2.0 | 1.0 | 0.0 | 2.0 | 0.0 | 0.0 |
| {B} | 2.0 | 0.0 | 0.0 | 0.0 | 2.0 | 0.0 |
| {B, C} | 3.0 | 0.0 | 0.0 | 0.0 | 2.0 | 0.0 |
| {B, C, D} | 2.0 | 0.0 | 1.0 | 2.0 | 2.0 | 0.0 |
| {B, D} | 2.0 | 0.0 | 0.0 | 2.0 | 2.0 | 0.0 |
| {C} | 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| {C, D} | 2.0 | 0.0 | 1.0 | 2.0 | 0.0 | 0.0 |
| {D} | 2.0 | 0.0 | 0.0 | 2.0 | 0.0 | 0.0 |

# BIBLIOGRAPHY


[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013, pp. 26–36. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013, pp. 243–244. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7.

[4] Andrea Arcuri and Lionel C. Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ACM, 2011, pp. 1–10. DOI: 10.1145/1985793.1985795. URL: https://doi.org/10.1145/1985793.1985795.

[5] Johannes Dorn, Sven Apel, and Norbert Siegmund. "Mastering Uncertainty in Performance Estimations of Configurable Software Systems." In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 684–696. DOI: 10.1145/3324884.3416620. URL: https://doi.org/10.1145/3324884.3416620.

[6] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "Predicting Performance of Software Configurations: There is no Silver Bullet." In: *CoRR* abs/1911.12643 (2019). arXiv: 1911.12643. URL: http://arxiv.org/abs/1911.12643.

[7] Jürgen Groß. *Linear Regression*. Springer Berlin Heidelberg, 2003. DOI: 10.1007/978-3-642-55864-1. URL: https://doi.org/10.1007/978-3-642-55864-1.

[8] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. "Distance-Based Sampling of Software Configuration Spaces." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1084–1094. DOI: 10.1109/ICSE.2019.00112.

[9] Janik Keller. ""Feature Taint Analysis: How Precise can VaRA Track the Influence of Feature Variables in Real-World Programs"." Master Thesis. Germany: University of Saarland, 2023.

[10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. "Evolution of the Linux Kernel Variability Model." In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 136–150. DOI: 10.1007/978-3-642-15579-6_10. URL: https://doi.org/10.1007/978-3-642-15579-6_10.

[11] Florian Sattler. ""A Variability-Aware Feature-Region Analyzer in LLVM."" Master Thesis. Germany: University of Passau, 2017.

[12] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)." In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 317–331. DOI: 10.1109/SP.2010.26. URL: https://doi.org/10.1109/SP.2010.26.

[13] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-influence models for highly configurable systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 284–294. DOI: 10.1145/2786805.2786845. URL: https://doi.org/10.1145/2786805.2786845.

[14] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. "Approaching Non-functional Properties of Software Product Lines: Learning from Products." In: *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. Ed. by Jun Han and Tran Dan Thu. IEEE Computer Society, 2010, pp. 147–155. DOI: 10.1109/APSEC.2010.26. URL: https://doi.org/10.1109/APSEC.2010.26.

[15] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "ConfigCrusher: towards white-box performance analysis for configurable systems." In: *Autom. Softw. Eng.* 27.3 (2020), pp. 265–300. DOI: 10.1007/s10515-020-00273-8. URL: https://doi.org/10.1007/s10515-020-00273-8.

[16] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems." In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1072–1084. DOI: 10.1109/ICSE43902.2021.00100. URL: https://doi.org/10.1109/ICSE43902.2021.00100.

[17] Max Weber, Sven Apel, and Norbert Siegmund. "White-Box Performance-Influence Models: A Profiling and Learning Approach (Replication Package)." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 232–233. DOI: 10.1109/ICSE-Companion52605.2021.00107.

[18] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 307–319. DOI: 10.1145/2786805.2786852. URL: https://doi.org/10.1145/2786805.2786852.

[19] Tom Zahlbach. ""Finding Feature-Dependent Code: A Study on Different Feature-Region Detection Approaches"." Master Thesis. Germany: University of Saarland, 2023.