Bachelor's Thesis

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

MANUEL MESSERIG

April 25, 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 includes various configuration options. However, it is not clear to the end-user how these configuration options might interact and influence properties such as performance.

We are interested in whether a change in behavior is the result of single independent configuration options or due to these options interacting with each other. In previous research, two different methods have emerged for identifying these influences: black-box analysis and white-box analysis. We are interested in how accurately these two analysis methods can identify the influences of each configuration option and their interactions. To present these influences, we use performance-influence models and build a model for each analysis out of the data they produce. We use these performance-influence models as the foundation for comparing and identifying the differences between white-box and black-box analyses.

In this thesis, we analyze 5 different configurable software systems designed by us and the compression tool XZ using both white-box analysis and black-box analysis. We present a structured approach to comparing white-box and black-box analyses. We could confirm that systems containing multicollinearity leads to inaccurate results for both analyses. In addition, the white-box analysis could not identify most configuration options for XZ.

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to both of my advisors Florian Sattler and Christian Kaltenecker. Both supported me in their best capacity and gave me helpful criticism and good feedback. I would especially like to thank them for being patient and responding quickly to my questions, even after regular working hours. I couldn't have found better advisors to accompany me through this thesis.

Second, I would like to thank my friends Lukas Abelt, Nils Alznauer, Alexander Rogovskyy, and Karl Schrader, who proofread this thesis and provided suggestions and feedback.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

TEF trace event format

VIF variance inflation factor

1 INTRODUCTION

Nowadays, all modern software systems are configurable and offer a large variety of functionality to satisfy multiple interest groups with a single configurable software system. These functionalities are represented as features that the user chooses by selecting the configuration options corresponding to that feature. However, with the increasing complexity of modern configurable software systems, the number of features the system contains also increases [1].

An example of such a configurable software system is the Linux kernel, whose code base itself contains over 6'000'000 lines of code with more than 10'000 features [9]. All these features implement functionality, which allows the user to select the desired features he wants to create an operating system that meets his needs.

All these features make it increasingly difficult to understand how much they influence the qualitative and quantitative aspects of a configurable software system. Furthermore, all these features might interact with one another, which changes the software systems behavior and runtime. To identify to which degree these features and their interactions influence the system, previous research introduced two different analyses techniques: white-box [14, 15] and black-box [4, 12].

For black-box analysis, it is not necessary to have access to the source code, one such example for this is that we only measure the time spent when executing the system using different features. We repeat this measurement process for all the features and feature interactions we want to measure. Afterwards, we need to infer the time spent for each feature. For this purpose, machine-learning methods already have been established in literature such as multiple linear regression, classification and regression tree, and random forest [4, 6, 12].

However, if we have access to the system's source code, we can use a white-box analysis. White-box analysis uses this information of the source code to achieve a more fine-grained view of the system. Different approaches for the white-box analysis exist to identify the time spent inside the different features and interactions [14, 15]. We use the analysis framework VaRA to identify different code regions affected by each feature and feature interaction and then measure the time spent in these regions.

The goal of this thesis is to compare both analyses to identify advantages and disadvantages of using one over the other. However, the output data of both analyses differ and are not directly comparable with each other. Therefore, we need to find a model which makes the data comparable. Thus, we introduce performance-influence models an established way to model configurable software systems by assigning the expected influence to each feature and feature interaction [12]. Afterwards, we build one model for each of our analyses using the data they produce, which we then use to compare the influences of each feature and feature interaction with each other.

1.1 GOAL OF THIS THESIS

In this work, we compare white-box analysis and black-box analysis with each other. We investigate how accurately both analyses can identify the influence of features and their interactions. We are interested in if an analysis loses accuracy due to specific reasons. Additionally, we are interested in how similar the performance-influence models of both analyses are. Since they are built for the same system, we want to know if the models agree because we can use both analysis methods interchangeably depending on how similar they are. Last, we identify potential advantages and disadvantages of each analysis, so that the user can take these into concern when deciding on which one to use.

In this thesis we answer the following research questions:

RQ1 : How accurately do 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?

To answer these questions, we introduce an approach to compare white-box and black-box performance-influence models that can be used in further studies. In addition, we implement five different configurable software systems that can serve as a baseline for future evaluations. Last, we use both white-box and black-box analyses on the real-world compression tool XZ.

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 a 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 users 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 [17].

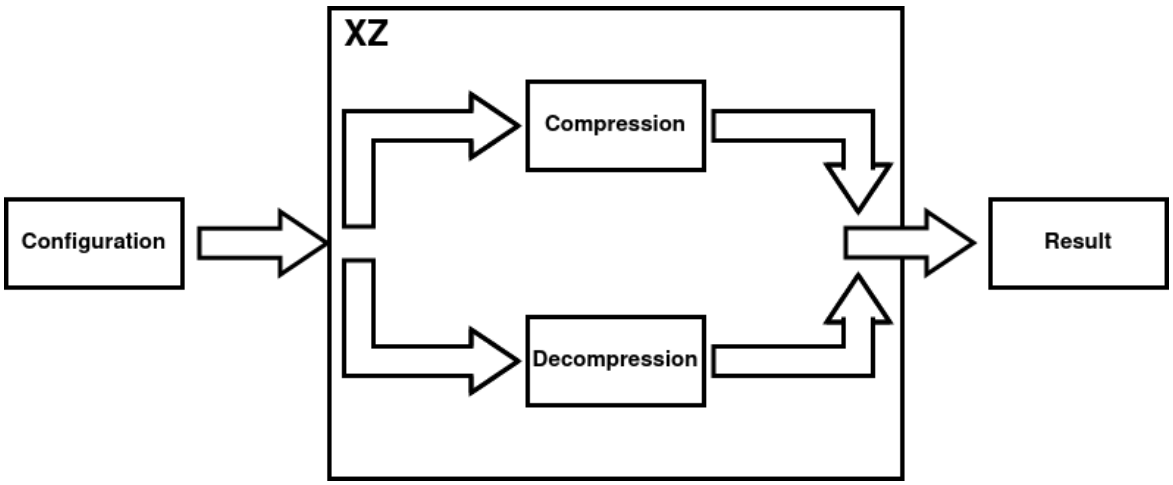

Figure 2.1: Simplified version of XZ.

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 they need, for when they want to compress a file they would select the configuration optional 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. 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 possible 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 [13]. 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 [13]. 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 [13].

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 larger 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 and define which feature selections are valid [1].

As 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*

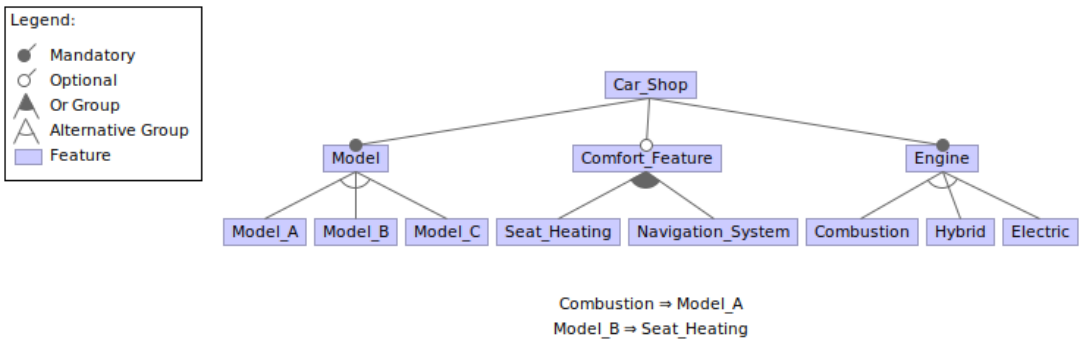

Figure 2.2: A feature diagram of a car dealership.

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* are concrete features that implement specific types 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 [1]. In Figure 2.2, we can see that *Engine* and *Model* are mandatory features, which make sense given that they are 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 [1]. In Figure 2.2, we see that *Engine* has an alternative group, since each car can only contain one *Engine*, whereas it makes sense that *Comfort_Feature* contains a choice group: one 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 [1].

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. 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 [12].

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 Π that maps configurations $\mathcal{C}$ to a prediction, therefore $\Pi : \mathcal{C} \implies \mathbb{R}$ [12].

We encode all our features as binary features and distinguish between single features o denoted as ϕ_o and feature interactions $i...j$ denoted as $\Phi_{i...j}$. Based on these definitions, we define a performance-influence model formally as [12]:

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

While β_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 [12].

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 the user-specified value to each 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 selected. 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 selected 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 β_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.

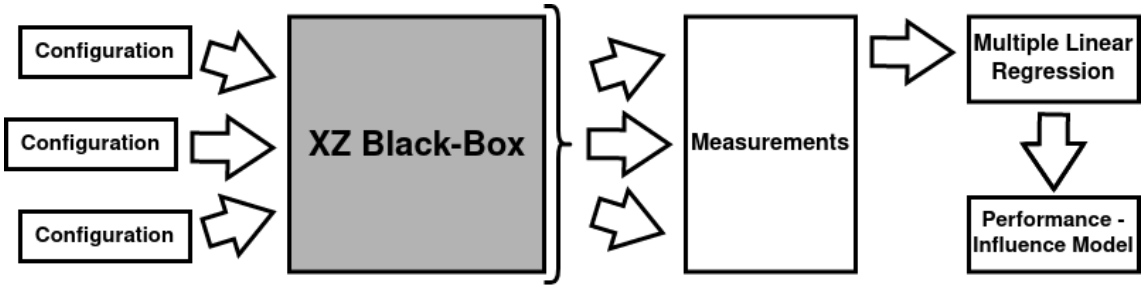

Figure 2.3: Process of using a black-box analysis to build a performance-influence model for *XZ*.

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-influence 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-influence 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-influence 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 [1].

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 [9], 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. [17], 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, Π_1 and Π_2 predict for any configuration the same amount of time, however, if we take a close look at how Π_1 assigns the influence of each feature, we can see that this is not interpretable. Here Π_1 assigns to the *Base* feature an influence of -1000, which does not make sense since the time spent in the *Base* feature can not be negative. This also leads to Π_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 [4]:

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

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

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 β, since they quantify the influence of each feature or feature interaction on the whole system. In addition, β_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 ϵ [5].

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 β 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 [5].

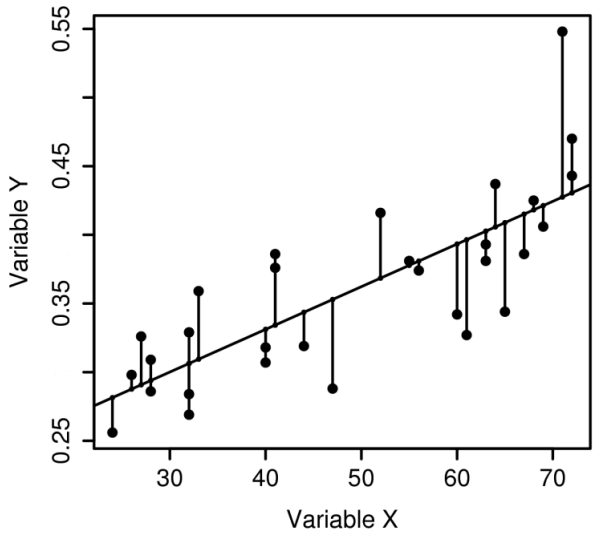

Figure 2.4: Ordinary Least Squares regression model residuals [2].

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 [5]:

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

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

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 and as we see 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-influence 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 [5]. 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-influence 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. [3]

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-influence 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.

Π	Base	A	B	C	$\Pi(\{Base\})$
Π_0	0	5	10	20	**0**
Π_1	5	10	5	20	**5**
Π_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 [3].

In Table 2.3, we see the predictions each of the 3 performance-influence models make for the configurations $\{Base\}$. Now Π_0, Π_1, and Π_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 software 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 [3].

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 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 $\overline{c}(o_j)$ [3].

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 on which terms to use with the 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, we introduce the general concepts behind white-box analysis. 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 5.1.

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 uses these measurements of how much time we spent in each feature and not just in the overall system to learn the influence of each feature to then build the performance-influence model.

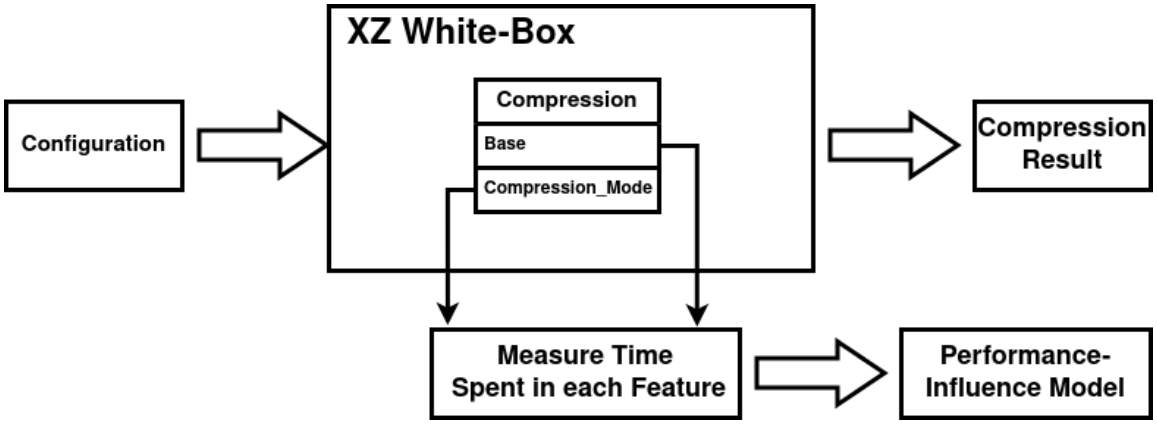

Figure 2.5: Process of using a white-box analysis to build a performance-influence model for *XZ*.

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 [10].

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* [11].

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 [8].

DOMINATOR APPROACH To detect these feature regions VaRA uses 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 [18].

2.5.2.3 *Locating feature variables*

Unfortunately, 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-influence 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*.

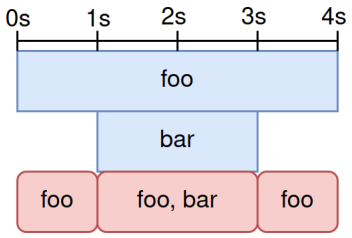

Figure 2.6: Example of two features interaction. Features are highlighted in blue and all the current active features are highlighted in red.

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 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.

To build up the performance-influence model, we must combine the knowledge of which features are selected for that configuration and which feature regions we measured. Since, when we measure the time spent inside a region, it does not imply that the feature that influences the region is selected. An example where this is the case is in Listing 2.2 in Line 8. Here we need to combine these measurements with the selected features because otherwise, we would attribute the time spent in *bar()* in not encryption to the feature *Encryption*.

To build the performance-influence model for the white-box, we again use multiple linear regression. However, instead of only using multiple linear regression once as we did for the black-box performance-influence model. We build a multiple linear regression model for each feature and feature interaction for which we collected the measurements during the white-box analysis. Thus, for the independent variable Y, we use the time measured for that feature region and combine it with the knowledge of which features are selected in that configuration. For each configuration we measured, we add 1 to the regression coefficient x_1 if all the features of the feature region are selected and if not, we add 0. This means that we only have one regression coefficient for each multiple linear regression. Suppose we have an intercept β_0 when learning a feature. In that case, we add that influence to the *Base* feature

because this is the influence independent of that feature being selected, which matches the description of our *Base* feature.

3

METHODOLOGY

In this chapter, we introduce our research questions. Furthermore, we formalize how we evaluate the research questions and explain how we measure and collect data for our evaluation. We then proceed in Section 3.4 to establish a ground truth, which tests both white-box analysis and black-box analysis in different scenarios to identify weaknesses. Last, in Section 3.3, we explain the experiment setup we choose to evaluate both analyses.

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, we want to answer the following research question:

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

Due to the number of features, it is hard to know the influence of each feature or feature interaction on a configurable software system. Therefore, when we use white-box or black-box analyses to quantify the influence of these features and feature interaction, we are interested in the accuracy of both analyses. Another point of interest is if we can identify a group of features for which either analysis performs worse than its counterpart. To answer this, we research how accurately they can identify the influence of features and feature interactions.

Thus, we quantify the difference between the predicted and true influence of each feature or feature interaction f in the set of features and feature interactions F by calculating $error_f$ and afterwards its mean $\overline{error}$, following the approach outlined in [7].

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

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

Note that $actual\_true_f$ is the true influence of the feature or feature interaction f that we obtain from the baseline of our ground truth. Whereas $predicted_f$ 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.

To investigate this, we use a qualitative measurement using the ground truth systems we design. We build a performance-influence model with each analysis method, and use Equation 3.4, for which we know the actual values for each feature and feature interaction.

Another, point of interest is how similar the performance-influence models are. We answer 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?

This is of interest because, because if both analyses produce similar results we can use them both interchangeable with each other and therefore choose the analysis we prefer without compromising its accuracy.

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 use the following formulas:

$$similarity_f = \frac{|predicted_f^{WB} - predicted_f^{BB}|}{\Pi_{BB}} \tag{3.3}$$

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

Here WB stands for white-box and BB for black-box. How similar two features are is calculated by $similarity_f$, where we normalize the absolute difference of the time prediction for each feature by the overall time for the black-box performance-influence models. Ideally, the overall time for both performance-influence models should be the same, with only the time distribution between features being different. However, the measurement code creates an overhead, which is minimal for the black-box analysis since we only measure when the system starts and when it ends. Afterward, we calculate the mean similarity $\overline{similarity_f}$ by dividing through the number of features and feature interactions F. A $\overline{similarity_f}$ of 0 indicates that both performance-influence models are perfectly similar.

To evaluate this question, we use both qualitative and quantitative measurements. For the qualitative measurements, we use the ground truth systems we design to identify why the performance-influence models differ. For the quantitative measurements, we use experimental results to investigate the reasons for similarities and differences.

3.2 COLLECTING DATA

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

The non-functional property we analyze is runtime, because it is 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 [2] any measure each configuration 30 times. Afterwards, we take the mean of the time spent as values for that configuration 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 log the time spent in a different report.

All measurements are performed on the same server node. To minimize background noise, we ensure 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. The server nodes operating system is Debian 11.

During the evaluation, we used different libraries and will mention the noteworthy ones here. To build the performance-influence model we use the implementation of multiple linear regression by scikit-learn[1]. In addition to that we use the implementation of the variance inflation factor implemented in the statsmodels[2] library.

3.3 EXPERIMENT SETUP

This section describes the main experiment conducted to evaluate both analyses.

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[3]. The primary compression algorithm that XZ currently uses to perform lossless data compression is LZMA2. We use XZ (XZ Utils) version 5.5.0[4].

The file we compress is a geographic map encoded as a JSON file of size 203 MB[5], which we choose because XZ achieves a compression rate of at least 70%. This ensures that for a file of 203 MB, a significant amount of time is spent in each feature and feature interaction we want to measure.

1 Visited at 03.04.2023
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
2 Visited at 03.04.2023
https://www.statsmodels.org/stable/generated/statsmodels.stats.outliers_influence.variance_inflation_factor.html
3 Visited at 02.04.2023
https://tukaani.org/xz/
4 Commit hash "610dde15a88f12cc540424eb3eb3ed61f3876f74"
5 Visited at 03.04.2023
https://github.com/simonepri/geo-maps/releases/latest/download/countries-land-1m.geo.json

3.3.1 *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 measure all possible configurations and ensure the accuracy of the black-box analysis.

We selected three core features of XZ that the documentation says to have a significant influence on 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 option 0 makes XZ use as many threads as available cores. Multithreading allows *XZ* to split the input into different blocks and compress them independently of one another. This feature is optional, and encoded as an alternative group.

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

$$\begin{aligned} |C| &= |\textit{compression level}| \cdot |\textit{extreme}| \cdot |\textit{threads}| \\ 100 &= 10 \cdot 2 \cdot 5 \end{aligned} \tag{3.5}$$

Here $|\textit{compression level}|$, $|\textit{extreme}|$, and $|\textit{threads}|$ are the number of configuration options for the respective feature.

To identify the *Base* feature for XZ, we need to identify the time spent in compression independent of which features are selected. However, XZ does not allow us to select the mode compression without choosing a compression level. Therefore, as the *Base* feature, we choose the configuration with minimal time spent compressing, which is *compression level 0* and *threads 0*.

3.4 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 software systems.

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 from which we manually build the performance-influence models that we use for comparison. We design all these systems with a different focus 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 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.

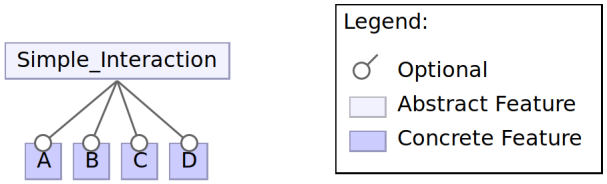

Figure 3.1: Feature model of Listing 2.1.

ELSE CLAUSE The first modification is the addition of 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 clause* to feature *A*, if *A* deselected. In contrast, the black-box analysis cannot differentiate between the *else clause* that is executed when feature *A* is deselected and the time spent in *Base*. We are interested in whether the white-box performance-influence model still identifies the influence of feature *A* correctly, even though the analysis attributed time to *A* when it is deselected.

FUNCTION Another modification of our *Simple Interaction* system is adding a function in which we spend time. In Line 7 we call the function *waste_time(1)* instead of *fpcsc::sleep_for_secs(1)*.

```
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 changes the feature model and introduces multicollinearity into the system. We see the modified in Figure 3.2. The white-box analysis should still be able to correctly identify the time spent in *Base* and feature *B*, while the black-box analysis should distribute the time spent between the features differently.

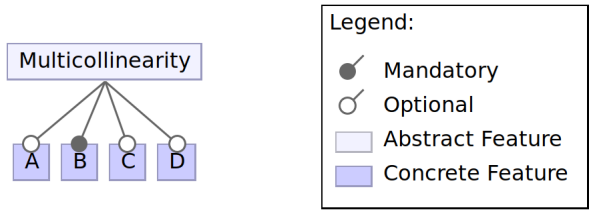

Figure 3.2: Feature model of the *Multicollinearity* system.

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 feature *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 EVALUATION

This chapter evaluates the thesis's core claims. We present the results for both the ground truth systems and XZ in Section 4.1. Afterwards, we discuss the results in Section 4.2. At the end in Section 4.3, we explain the threats to validity to our approach.

4.1 RESULTS

This Section presents the results of the ground truth systems and XZ. All values for the ground truth systems are rounded to 3 decimal places, whereas for the XZ experiment, we rounded the values to 6 decimal places. This should not influence the comparability of our models, since in comparison to the overall time spent inside the system this influence is insignificant. Due to that, for the white-box we also discard all features and feature interaction for which the summed up time spent in these regions is less than one millisecond. All results values for the ground truth and experiment results are represented in seconds.

4.1.1 *Ground Truth Results*

We now evaluate the ground truth systems *Simple Interactions*, *Else Clause*, *Function*, *Multicollinearity* and *Shared Feature Variable*, which we introduced in Section 3.4 and present the resulting performance-influence models.

Simple Interaction

Π	*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 4.1: Direct comparison between the baseline, black-box and white-box performance-influence models for *Simple Interaction*

The *Simple Interaction* system tests if both analyses can identify interactions between features, as we can see in Table 4.1 since all the performance-influence models are identical means that both analyses were able to identify these interactions.

Else Clause

Π	*Base*	*A*	*B*	*C*	*D*	$A \wedge B$	$C \wedge D$
Baseline	4	-1	2	1	2	2	0
Black-box	4	-1	2	1	2	2	0
White-box	4	-1	2	1	2	2	0

Table 4.2: Direct comparison between the baseline, black-box and white-box performance-influence models for *Else Clause*

The *Else Clause* system checks how both analyses attribute the time spent in the else clause. We see that the performance-influence models in Table 4.2 are identical, which means that the analyses were able to attribute the time correctly.

Function

Π	*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 4.3: Direct comparison between the baseline, black-box and white-box performance-influence models for *Function*

In the *Function* system, we test how the analyses handle it when we spend time in a different function than the main function. We see in Table 4.3 that the performance-influence models are the same which means that both analyses attribute the time correctly.

Multicollinearity

Π	*Base*	*A*	*B*	*C*	*D*	$A \wedge B$	$C \wedge D$
Baseline	2	1	2	1	2	2	0
Black-box	4	3	0	1	2	0	0
White-box	4	1	0	1	2	2	0

Table 4.4: Direct comparison between the baseline, black-box and white-box performance-influence models for *Multicollinearity*

For the *Multicollinearity* system, we change feature *B* from an optional to a mandatory feature, for which we expect the black-box analysis to be unable to attribute the time for the

affected features accurately. As we can see when we compare the performance-influence models in Table 4.4, the black-box is inaccurate for the features *Base*, *A*, *B* and the interaction $A \wedge B$, whereas the white-box in only inaccurate for *Base* and *A*.

Shared Feature Variables

Π	*Base*	*A*	*B*	*C*	*D*	*E*	$A \wedge B$	$C \wedge D$
Baseline	2.0	1.0	2.0	1.0	2.0	1.0	2.0	0.0
Black-box	2.0	1.0	2.0	1.0	2.0	1.0	2.0	0.0
White-box	6.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 4.5: Direct comparison between the baseline, black-box and white-box performance-influence models for *Shared Feature Variables*

For the *Shared Feature Variables*, we encoded two features *D*, and *E* as a single string. We expected that the white-box analysis is unable to distinguish between these features and instead encode them as one interaction. However, we see in Table 4.5 when inspecting the white-box performance-influence model and compare it to the baseline that it did not identify any features besides *Base* during the analysis. In addition, the white-box performance-influence model wrongly attributed the time for the *Base* feature.

4.1.2 *Experiment Results*

We now present the performance-influence models for XZ. The performance-influence model built from the black-box analysis can be seen in the Appendix A, where we split the table into two tables, where in Table A.3 the feature *extreme* is deselected and in Table A.4 *extreme* is selected. The performance-influence model built from the white-box analysis data can be seen in Table A.5 for which *extreme* is deselected and in Table A.6 *extreme* is selected. We see that VaRA was unable to measure the influences for all features and interactions besides *Base* and *threads 1,2,4*, and *8*.

4.1.3 *Results Research Questions*

We now present our results for the research questions. We discard feature interactions that have an influence of 0 across all performance-influence models for the same system.

Results RQ1

We now present the results for RQ 1, in which we investigate the error for the performance-influence models by evaluating the ground truth systems using the performance-influence model previously shown.

error	*Base*	*A*	*B*	*C*	*D*	$A \wedge B$	*E*	$\overline{error}$
Simple Interaction								
Black-box	0	0	0	0	0	0		0
White-box	0	0	0	0	0	0		0
Else Clause								
Black-box	0	0	0	0	0	0		0
White-box	0	0	0	0	0	0		0
Function								
Black-box	0	0	0	0	0	0		0
White-box	0	0	0	0	0	0		0
Multicollinearity								
Black-box	1	2	1	0	0	1		0.833
White-box	1	0	1	0	0	0		0.333
Shared Feature Variable								
Black-box	0	0	0	0	0	0	0	0
White-box	2	1	1	1	1	1	1	1.143

Table 4.6: Respective *error* scores and $\overline{error}$ score for white-box and black-box performance-influence models for the *Ground Truth* systems. We discarded all values below one millisecond since their impact is comparably insignificant.

In Table 4.6 we see the *error* and $\overline{error}$ scores for each analysis per system. For the first three systems *Simple Interaction*, *Else Clause*, and *Function*, we have already seen that the performance-influence models of each analysis are identical to the baseline model. Therefore, the *error* scores are all 0 and by that the $\overline{error}$ too.

Fer the *Multicollinearity* system, both analyses contain *error* scores that indicate a difference between the analysis and the baseline performance-influence models. We first inspect the black-box analysis performance-influence model where we have an *error* score higher then 0 for the features *Base*, *A*, *B*, and the interaction $A \wedge B$ and an $\overline{error}$ of 0.833. For the white-box we only have an *error* score for the features *Base* and *B*, whereas the $\overline{error}$ score is 0.333.

Last, we inspect the results for the *Shared Feature Variable*, we have no *error* for the black-box analysis and therefore an $\overline{error}$ of 0. For the white-box we have an *error* for every feature and feature interaction, this means that for no feature we were able to correctly predict the time correctly, which results in an $\overline{error}$ of 1.143.

4.1.4 *Results RQ2*

We now present the results for RQ 2 by evaluating the ground truth systems and the experiment using the performance-influence model previously shown. We begin with presenting the ground truth systems results.

similarity	*Base*	*A*	*B*	*C*	*D*	$A \wedge B$	*E*	$\overline{similarity}$
Simple Interaction	0.0	0.0	0.0	0.0	0.0	0.0		0.0
Else Clause	0.0	0.0	0.0	0.0	0.0	0.0		0.0
Function	0.0	0.0	0.0	0.0	0.0	0.0		0.0
Multicollinearity	0.0	0.2	0.0	0.0	0.0	0.2		0.033
Shared Feature Variable	0.364	0.091	0.182	0.091	0.182	0.182	0.091	0.143

Table 4.7: Respective *similarity* scores and $\overline{similarity}$ score for each ground truth system.

We see the ground truth system results for research question 2 in Table 4.7. For the systems *Simple Interaction*, *Else Case*, and *Function* the *similarity* score is 0 and thus $\overline{similarity}$ is 0 too.

The *similarity* scores for the *Multicollinearity* system are 0 except for the feature A and the interaction $A \wedge B$ where they have a score of 0.2 and a $\overline{similarity}$ of 0.033 for the whole system.

The previous systems do not have a feature E and therefore no *similarity* score for this feature. However, we added this feature for our last system *Shared Feature Variable*. In addition, each feature and interaction have a different *similarity* score and for this system, the $\overline{similarity}$ score is 0.143.

XZ Results

We see the results for our experiment in Table A.7 and Table A.8, with the similarity scores for each feature and feature interactions. For Table A.7 the feature *extreme* is deselected and in Table A.8 it is selected. The time for Π_{BB} is 3123.275 seconds, which we used to calculate the $\overline{similarity}$ score of 0.003898.

4.2 DISCUSSION

In this Section, we discuss the results of both analyses. First, we discuss the resulting performance-influence models for both the ground truth systems and XZ, and then the results of the research questions.

We first look at the performance-influence models created by both analyses, these cover basic functionalities which every modern configurable software system contains. Thus, we are interested if both analyses were able to correctly identify the time spent in each feature and feature interactions in these basic systems, to ensure that both analyses can analyze simple systems. We start by inspecting the respective models for the ground truth systems *Simple Interactions*, *Else Clause*, and *Function*. As expected, the performance-influence models for all three systems are identical. This suggests that both analyses accurately identified the influence of all features and feature interactions for simple systems.

For both analyses, we have an $\overline{error}$ score of 0. This confirms that our performance-influence models are identical to the baseline and therefore accurate. Hence, if both analyses' performance-influence models are identical to the baseline, they are also identical to each

other, which the $\overline{similarity}$ score of 0 confirms. For the these simple systems, both models are able to correctly attribute the runtime.

For the *Multicollinearity* system, we expected the black-box analysis to be inaccurate due to the introduction of multicollinearity and when we inspect the performance-influence model built out of the black-box data, we see noticeable changes compared to our baseline.

The first change is that, due to feature B being mandatory, feature A and the interaction $A \wedge B$ are perfectly multicollinear since B is always selected in conjunction with A. This is detected when we decide which terms to add to our multiple linear regression model using Algorithm 1 and thus $A \wedge B$ is not added to the model. In addition, due to multicollinearity, the time distribution between features in the performance-influence model changed. While features C and D are still correctly attributed, we have a change for features *Base* and A with an increase of 2 seconds, and feature B with a decrease of 2 seconds. We assume that the time spent in feature B is attributed to feature *Base* since they are indistinguishable for the black-box. The 2 seconds spent in the interaction $A \wedge B$ is attributed to A due to the perfect multicollinearity between these features.

The performance-influence model for the white-box analysis is affected by this as well, which shows in the model too. However, compared to the black-box, the white-box still correctly identifies the influence of feature A and the interaction $A \wedge B$. In our opinion this is because, when we built the performance-influence model out of the white-box data, the multiple linear regression algorithm we use assigned the influence to feature *Base* instead of feature B. However, since we know the specific influence of each feature and interaction, the white-box can differentiate between feature A and interaction $A \wedge B$.

We inspect the *error* scores to answer RQ1. In this system we have an *error* for the features affected by multicollinearity, which results in an $\overline{error}$ of 0.833. This indicates a high average error when using a black-box model for multicollinear systems. The white-box analysis has a smaller $\overline{error}$ of 0.333 because it detects the influence of feature A and the interaction $A \wedge B$.

For RQ2 we have a $\overline{similarity}$ score of 0.033 for this system, which means that each feature an average difference of 0.033 of the overall time for the system. Our interpretation of this score is that multicollinearity does influence both of our models.

Next, we inspect the results for the *Shared Feature Variables* system. As expected, the black-box analysis identified the influence of all features correctly. As for the white-box analysis, something unexpected happened: We expected that the white-box analysis would not be able to correctly identify the influence of features D and E since they share one feature variable. What happened is that although we did not change the feature variables for A, B, and C, VaRA was unable to identify any features at all. This led to VaRA assigning the influence of every feature to the feature *Base*. Table A.2 shows all the measurements.

To answer RQ1, we inspect the $\overline{error}$. Since the black-box analysis is identical to the baseline, we have an $\overline{error}$ score of 0. Since the measurements of the white-box analysis were wrong the performance-influence model built is also faulty. This results in large *error* scores for each feature and feature interaction and leads to an $\overline{error}$ score of 1.143, which says that on average the predicted time is of by a factor of 1.143 compared to the actual value. This indicates that for this system the white-box analysis is unable to predict the influence of features and feature interaction.

For research question two, we inspect the $\overline{similarity}$ score, which for this system is ~ 0.143. This means that the difference between each feature is, on average, 0.143 of the overall time for the system, which indicates that the analyses predict severely different values for most features and feature interactions.

Last, we discuss the results of our experiment. Compared to our ground truth systems, we do not know the time spent in each feature or interaction in XZ. Therefore, we cannot accurately say that the assigned time influences of our performance-influence models are correct. Nevertheless, we shall state our assessments here.

The performance-influence model built out of the black-box analysis data appears accurate. When inspecting the values assigned to all single features, they estimate a similar time compared to the time XZ uses when compressing data. When using different compression levels with different threads, the predictions align with our expectations. Furthermore, the values in Table A.4 agree with the description of the *extreme* feature because all feature interactions where *extreme* is selected together with the influence of the feature *extreme* itself are positive, indicating that they all slow down the system.

However, we have a severely different result for the performance-influence model built from the white-box analysis. When inspecting the measurement results from VaRA, we notice that no other feature regions besides *threads* were found. This led to VaRA assigning the time spent during compression largely to *Base*. Therefore, besides the four *threads* features and the *Base* feature, we have no other measurements, which leads to Table A.5 and Table A.6 being populated predominantly with zeros. To investigate this, we manually analyzed the source code of XZ to find the reasons which might have led to VaRA not being able to correctly identify the corresponding feature regions. First, we ensured that we identified the correct variables that corresponded to the features we wanted to analyze. For this reason, we found the enums `MODE_COMPRESS` and `opt_mode` in the `coder.h`[1] file, which decide whether XZ uses compression or decompression. Next, we identified where XZ sets its compression level. This happens in `coder.c`[2] in the `coder_set_preset` function, which overrides the default value of *preset_number* set in the same file. This is where one issue surfaces that VaRA can not handle. Both features *compression level* and *extreme* are implemented by the same feature variable *preset_number*. When *compression level* and *extreme* are selected, XZ uses different bit operators to calculate the preset that determines the degree of compression. We think these bit shifts are why VaRA cannot identify the feature regions of that feature. However, even if that was not the issue, the fact that XZ uses the same variable to implement the two different features *compression level* and *extreme* probably leads to VaRA being unable to differentiate between these two.

This faulty white-box performance-influence model also leads to significant differences between each feature. However, we obtain small *similarity* scores due to normalizing these values with the summed-up influence of all features and feature interactions. When interpreting them, we need to regard them in relation to the summed-up influence Π_{BB}, which is 3123.275. When inspecting the $\overline{similarity}$ score of 0.003898, this gives us an average

1 Visited at 23.04.2023
https://github.com/xz-mirror/xz/blob/master/src/xz/coder.h
2 Visited at 23.04.2023
https://github.com/xz-mirror/xz/blob/master/src/xz/coder.c

difference of 12.175 seconds. Therefore, we conclude that the two performance-influence models are not similar.

After discussing our results we conclude this Section with answering our research questions:

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

The performance-influence models built from the white-box and black-box data produced an $\overline{error}$ score of 0 for the *Simple Interaction*, *Else Clause*, and *Function* systems. Therefore, they successfully detected the influence of each feature and feature interaction. However, for the *Multicollinearity* system, both did produce inaccurate results: The black-box produced an $\overline{error}$ score of 0.833 and the white-box an $\overline{error}$ score of 0.333. For the *Shared Feature Variable* system, the black-box detected all features and interactions successfully while the white-box failed, which resulted in an $\overline{error}$ score of 1.143.

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

For the ground truth *Simple Interaction*, *Else Clause*, and *Function* systems, both performance-influence models produced the same results. Therefore, the $\overline{similarity}$ score is 0. However, for the *Multicollinearity* system, the $\overline{similarity}$ score is 0.033 and for the *Shared Feature Variable* system, the $\overline{similarity}$ score is 0.143, both of which indicating that the models were not similar. For the experiment, we obtained a $\overline{similarity}$ score of 0.003898, indicating that the models are not similar.

4.3 THREATS TO VALIDITY

In this section, we shall discuss potential threats to internal and external validity of our evaluation.

Internal Validity

One threat to the internal validity is that during the execution of the system, external influences can affect the performance of the server node and therefore our measurements. To minimize this, we ensured that the only thing being executed on the server node during the measurements is our system. In addition, we repeated every measurement 30 times to reduce the impact of outliers.

Another threat to internal validity is the risk of an implementation error when evaluating the black-box and white-box data. We use multiple linear regression for both. In addition, for the black-box evaluation, we calculate the VIF. To minimize the chance of an implementation error, we use well-known libraries such as SKLEARN for multiple linear regression and STATSMODELS to calculate the VIF.

External Validity

A threat to the external validity is that the experiment we choose favors one analysis over the other. To minimize this threat, we choose a stable release version of a real-world configurable software system that is open source and analyzable by white-box and black-box.

5 RELATED WORK

In this chapter, we introduce different the white-box analysis strategies and tools.

5.1 STRATEGIES

When analyzing systems using a white-box approach, different strategies have been introduced. In this section, 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 [14], 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-influcence 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 influcence map, which maps statements to the configuration options that influence that statement. Afterward, they use both the control flow graph and statement influcence 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 [15], 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. [16] 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-influence model that shows how strong each feature influences the performance of that method.

6 CONCLUDING REMARKS

6.1 CONCLUSION

In this work, we compared white-box and black-box analyses on whether they can identify different features and feature interactions that influence configurable software systems. To investigate this, we first implemented five different systems, each testing a different scenario, which we analyzed with the presented white-box and black-box methods. Additionally, we designed an experiment in which we used the analyses on the real-world compression tool XZ.

We first investigated how accurate performance-influence models learned by the analyses' data are. To evaluate this question, we built a performance-influence model for each ground truth system with the correct influences functioning as a baseline. We found out that for the systems *Simple Interactions*, *Else Clause*, and *Function*, both analyses produced performance-influence models identical to the baseline model. However, both analyses had difficulties with the system that introduced multicollinearity. For the last system *Shared Feature Variable*, the white-box analysis using VaRA failed to identify any features, whereas the black-box analysis produced a reasonable performance-influence model.

Afterwards, we researched how similar the performance-influence models produced by both analyses are. We found differences between the models for the *Multicollinearity* system. However, we think the difference is in an acceptable range. In contrast, for the *Shared Feature Variable* system, the difference was too significant for us to call these models similar. We proceeded with evaluating our XZ experiment using the same methods. However, when inspecting the performance-influence models, we noticed that the white-box analysis could not identify 95 of the 100 features and feature interactions we measured. Therefore, both performance-influence models were completely different.

6.2 FUTURE WORK

Regarding future work, there are many areas to expand on.

We used multiple linear regression to build our performance-influence models. It is of interest to test other methods to build these models and research if they perform better.

Additionally, we only looked at XZ as a real-world configurable software system, for which the white-box analysis could not identify the majority of features and feature interactions. However, there are many other systems for which the results might differ.

Next, we used VaRA for the white-box analysis. However, we also introduced other analyzing tools and it is worth investigating whether these tools produce similar results or if one outperforms the other.

Last, to deal with the issue of combinatorial explosion, we only investigated a limited configuration space. It is interesting to see how well both analyses perform if they use sampling strategies instead and research if this changes the accuracy of either analysis.

A

APPENDIX

We provide all data and evaluation scripts in the Zip file of this work. In addition, we upload them into the following GitHub repository: https://github.com/Aufrichtig98/BachelorThesisEvaluation.

	Base	A	B	C	D	A ∧ B	C ∧ D
{}	2.0	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	0.0
{A, B}	2.0	1.0	2.0	0.0	0.0	2.0	0.0
{A, B, C}	2.0	1.0	2.0	1.0	0.0	2.0	0.0
{A, B, C, D}	2.0	1.0	2.0	1.0	2.0	2.0	0.0
{A, B, D}	2.0	1.0	2.0	0.0	2.0	2.0	0.0
{A, C}	2.0	1.0	0.0	1.0	0.0	0.0	0.0
{A, C, D}	2.0	1.0	0.0	1.0	2.0	0.0	0.0
{A, D}	2.0	1.0	0.0	0.0	2.0	0.0	0.0
{B}	2.0	2.0	2.0	0.0	0.0	0.0	0.0
{B, C}	2.0	2.0	2.0	1.0	0.0	0.0	0.0
{B, C, D}	2.0	2.0	2.0	1.0	2.0	0.0	0.0
{B, D}	2.0	2.0	2.0	0.0	2.0	0.0	0.0
{C}	2.0	2.0	0.0	1.0	0.0	0.0	0.0
{C, D}	2.0	2.0	0.0	1.0	2.0	0.0	0.0
{D}	2.0	2.0	0.0	0.0	2.0	0.0	0.0

Table A.1: White-box analysis results for the *Else Clause* system

In Table A.1 we see the white-box analysis measurements for the *Else Clause* system. In this system feature *A* got an influence of 2 seconds if it is deselected. This illustrates again why we cannot simply build the performance-influence model without including the information of the currently selected features inside the configuration.

	Base	A	B	C	D	E	A ∧ B	C ∧ D
{}	2.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A}	3.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B}	7.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, C}	8.007	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, C, D}	10.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, C, D, E}	11.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, C, E}	9.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, D}	9.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, D, E}	10.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, B, E}	8.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, C}	4.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, C, D}	6.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, C, D, E}	7.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, C, E}	5.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, D}	5.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, D, E}	6.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{A, E}	4.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B}	4.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, C}	5.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, C, D}	7.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, C, D, E}	8.001	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, C, E}	6.007	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, D}	6.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, D, E}	7.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{B, E}	5.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{C}	3.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{C, D}	5.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{C, D, E}	6.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{C, E}	4.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{D}	4.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{D, E}	5.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0
{E}	3.000	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table A.2: White-box analysis results for the *Shared Feature* system

Table A.2 shows the white-box analysis results for the *Shared Feature Variable* system. Here we see that VaRA cannot identify any feature region besides that for feature *Base*.

Black-box	Base	threads_1	threads_2	threads_4	threads_8
Base	0.759	6.406	2.845	1.119	0.251
compression_level_1	0.271	2.442	1.079	0.390	0.108
compression_level_2	1.066	9.904	4.148	1.741	0.531
compression_level_3	2.757	20.859	9.201	3.803	1.128
compression_level_4	4.441	28.496	12.313	4.764	0.983
compression_level_5	7.546	47.279	19.125	7.208	1.250
compression_level_6	7.811	49.184	19.873	7.491	1.353
compression_level_7	17.261	53.346	16.777	0.878	-0.227
compression_level_8	41.543	40.312	-2.765	-1.132	-0.309
compression_level_9	91.877	-0.394	-2.576	-1.177	-0.181

Table A.3: Black-box performance-influence model for XZ with the feature extreme being deselected

In Table A.3, we see the first part of the performance-influence model for XZ where feature *extreme* is deselected of black-box analysis.

Black-box	Base	threads_1	threads_2	threads_4	threads_8
Base	2.744	28.790	12.410	4.856	1.124
compression_level_1	0.510	35.169	15.168	6.246	1.914
compression_level_2	0.618	37.456	16.133	6.649	1.978
compression_level_3	1.030	40.189	17.567	7.193	2.171
compression_level_4	-0.564	39.906	16.609	5.805	0.680
compression_level_5	-2.246	48.860	18.287	5.505	-0.863
compression_level_6	-2.339	50.477	18.720	5.552	-0.967
compression_level_7	-1.485	56.680	16.838	-0.347	-1.753
compression_level_8	0.836	45.689	-1.603	-0.290	0.470
compression_level_9	8.915	9.153	5.484	7.371	8.298

Table A.4: Black-box performance-influence model for XZ with the feature extreme being selected

For Table A.4, we see the second part of the black-box performance-influence model for XZ where feature *extreme* is selected.

White-box	Base	threads_1	threads_2	threads_4	threads_8
Base	36.231	0.095	0.073	0.072	0.074
compression_level_1	0.0	0.0	0.0	0.0	0.0
compression_level_2	0.0	0.0	0.0	0.0	0.0
compression_level_3	0.0	0.0	0.0	0.0	0.0
compression_level_4	0.0	0.0	0.0	0.0	0.0
compression_level_5	0.0	0.0	0.0	0.0	0.0
compression_level_6	0.0	0.0	0.0	0.0	0.0
compression_level_7	0.0	0.0	0.0	0.0	0.0
compression_level_8	0.0	0.0	0.0	0.0	0.0
compression_level_9	0.0	0.0	0.0	0.0	0.0

Table A.5: White-box performance-influence model for XZ with feature *Extreme* being deselected

Table A.5 presents the first part of the performance-influence model for XZ built out of the white-box analysis measurements where feature *extreme* is deselected. As we can see, the only features the white-box analysis detects are *Base*, *threads_1*, *threads_2*, *threads_4*, and *threads_8*.

White-box	Base	threads_1	threads_2	threads_4	threads_8
Base	0.0	0.0	0.0	0.0	0.0
compression_level_1	0.0	0.0	0.0	0.0	0.0
compression_level_2	0.0	0.0	0.0	0.0	0.0
compression_level_3	0.0	0.0	0.0	0.0	0.0
compression_level_4	0.0	0.0	0.0	0.0	0.0
compression_level_5	0.0	0.0	0.0	0.0	0.0
compression_level_6	0.0	0.0	0.0	0.0	0.0
compression_level_7	0.0	0.0	0.0	0.0	0.0
compression_level_8	0.0	0.0	0.0	0.0	0.0
compression_level_9	0.0	0.0	0.0	0.0	0.0

Table A.6: White-box performance-influence model for XZ with feature *Extreme* being selected

In Table A.6 we see the second part of the performance-influence model for XZ built out of the white-box analysis measurements where feature *extreme* is selected. As we see none of the feature or feature interaction have been detected.

similarity	Base	threads_1	threads_2	threads_4	threads_8
Base	0.011357	0.002021	0.000888	0.000335	0.000057
compression_level_1	0.000087	0.000782	0.000345	0.000125	0.000035
compression_level_2	0.000341	0.003171	0.001328	0.000557	0.000170
compression_level_3	0.000883	0.006679	0.002946	0.001218	0.000361
compression_level_4	0.001422	0.009124	0.003942	0.001525	0.000315
compression_level_5	0.002416	0.015138	0.006123	0.002308	0.000400
compression_level_6	0.002501	0.015748	0.006363	0.002398	0.000433
compression_level_7	0.005527	0.017080	0.005372	0.000281	0.000073
compression_level_8	0.013301	0.012907	0.000885	0.000362	0.000099
compression_level_9	0.029417	0.000126	0.000825	0.000377	0.000058

Table A.7: *similarity* scores for the XZ experiment with feature *Extreme* being deselected. The value for Π_{BB} is 3123.275

Table A.7 presents the first part of the *similarity* scores for the XZ experiment.

similarity	Base	threads_1	threads_2	threads_4	threads_8
Base	0.000879	0.009218	0.003973	0.001555	0.000360
compression_level_1	0.000163	0.011260	0.004856	0.002000	0.000613
compression_level_2	0.000198	0.011993	0.005165	0.002129	0.000633
compression_level_3	0.000330	0.012868	0.005625	0.002303	0.000695
compression_level_4	0.000181	0.012777	0.005318	0.001859	0.000218
compression_level_5	0.000719	0.015644	0.005855	0.001763	0.000276
compression_level_6	0.000749	0.016162	0.005994	0.001778	0.000310
compression_level_7	0.000475	0.018148	0.005391	0.000111	0.000561
compression_level_8	0.000268	0.014629	0.000513	0.000093	0.000150
compression_level_9	0.002854	0.002931	0.001756	0.002360	0.002657

Table A.8: *similarity* scores for the XZ experiment with feature *Extreme* being selected. The value for Π_{BB} is 3123.275

In Table A.8 the second part of the *similarity* scores for the XZ experiment is shown.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction." In: *IEEE Softw.* 37.4 (2020), pp. 58–66. DOI: 10.1109/MS.2020.2987024. URL: https://doi.org/10.1109/MS.2020.2987024.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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