Noname manuscript No.
(will be inserted by the editor)

On the Relation of Control-flow and Performance Feature Interactions: A Case Study

Sergiy Kolesnikov · Norbert Siegmund · Christian Kästner · Sven Apel

Received: date / Accepted: date

Abstract Detecting feature interactions is imperative for accurately predicting performance of highly-configurable systems. State-of-the-art performance prediction techniques rely on supervised machine learning for detecting feature interactions, which, in turn, relies on time-consuming performance measurements to obtain training data. By providing information about potentially interacting features, we can reduce the number of required performance measurements and make the overall performance prediction process more time efficient. We expect that information about potentially interacting features can be obtained by analyzing the source code of a highly-configurable system, which is computationally cheaper than performing multiple performance measurements. To this end, we conducted an in-depth qualitative case study on two real-world systems (MBEDTLS and SQLITE), in which we explored the relation between internal (precisely control-flow) feature interactions, detected through static program analysis, and external (precisely performance) feature interactions, detected by performance-prediction techniques using performance measurements. We found that a relation exists that can potentially be exploited to predict performance interactions.

1 Introduction

A *feature* is an end-user-visible behavior or characteristic of a (software) product that satisfies a stakeholder's requirement (Kang et al, 1990). Features are used to guide structure, reuse, and variation through the development of highly-configurable software systems (Apel et al, 2013a). While facilitating the development of highly-configurable software, reducing development costs, and improving product quality, combining features in a plug-and-play manner

S. Kolesnikov, University of Passau, Germany · N. Siegmund, Bauhaus-University Weimar, Germany · C. Kästner, Carnegie Mellon University, USA · S. Apel, Saarland University, Germany

introduces new challenges, such as the feature interaction problem (Bruns, 2005) or optional feature problem (Kästner et al, 2009). A *feature interaction* occurs when the functionality of a feature or its non-functional properties (e.g., performance) are influenced by the presence or absence of one or more other features (Zave, 2009). The presence of feature interactions hinders program comprehension and compositional reasoning about functional and non-functional properties: That is, we cannot reason about the properties of a system configuration (i.e., a valid feature combination) in terms of a straightforward combination of the *individual* influences of the involved features on these properties. This is because we also have to consider the influences of possible interactions among the involved features. A common practical scenario is searching for the best configuration of a system with respect to performance. To identify this configuration for a given operational environment, we need to know not only the individual influences of the involved features on performance, but also which interactions among these features exist and what influence on performance these interactions have.

The problem of detecting feature interactions and quantifying their influence on performance has been addressed in the past by employing machine learning (Siegmund et al, 2012; Guo et al, 2013; Zhang et al, 2015; Siegmund et al, 2015). For building a training dataset and identifying interactions, these techniques rely on selecting a representative subset from all system configurations (i.e., sampling) and on measuring the performance of each configuration in this sample (Sec. 3.3). The time needed to perform the measurements often makes up a substantial part of the overall time required by machine learning (Siegmund et al, 2013a; Kolesnikov et al, 2018). Therefore, reducing the measurement effort—by concentrating on system configurations that potentially reveal relevant feature interactions—can make these techniques more time efficient and accurate (Medeiros et al, 2016).

The main question that we address in this article is whether we can efficiently extract information about potentially existing feature interactions, which then can be used in performance prediction. In our previous work (Apel et al, 2013b), we described two types of interactions: (1) *external feature interactions*, which can be identified by observing the external behavior of a system, such as performance; and (2) *internal feature interactions*, which can be identified by analyzing or interpreting the source code of a system, for example, using control-flow analysis. A key hypothesis is that there is a relation between internal and external interactions, and that we can make use of this relation to automatically identify external interactions by identifying internal interactions in a fast and efficient way. For example, multiple function calls from one feature to another (internal feature interactions) can result in a performance overhead. This performance overhead arises only if the caller and the callee features are *both* present in a configuration (external feature interaction). This way, the internal interaction is related to its external counterpart. This relation, if present, would give us hints about the existence of external feature interactions based on the internal ones. In this work, we follow up on this idea and report on an exploratory case study in which we investigated the

control flow among features and its relation to *performance* feature interactions. We conjecture that by supplying the performance-prediction procedure with hints about which feature combinations are more likely or less likely to exhibit external feature interactions, the procedure can be made more focused on finding actual interactions.

Taking the *exploratory nature* of our study into account, the *qualitative character* of the expected results, as well as substantial technical challenges, we chose a *case-study approach* (Shull et al, 2007, p. 285) as our research method (see Sec. 3.1) with two systems—the MBEDTLS encryption library and the SQLITE database engine—as non-trivial, real-world subject systems.[1] MBEDTLS and SQLITE are highly-configurable systems used by several large projects, such as OPENVPN and FIREFOX,[2] which makes our case study practice-oriented.

Technically, using a state-of-the-art machine learning technique (Sec. 3.3), we learned *performance influence models* (Sec. 2.2), which we used in turn to identify potential external (performance) feature interactions among the features for the two subject systems. Furthermore, we manually inspected the code of the systems and checked whether the identified performance interactions actually exist and whether they are actually caused by the interplay of the corresponding features, and not just misinterpreted artefacts of measurement bias or environment noise.

Furthermore, using a variability-aware control-flow analysis, augmented by manual code inspection (Sec. 3.2), we identified control-flow interactions among the features of MBEDTLS and SQLITE. That is, we identified the code locations where the features pass the control to one another. Comparing the set of internal (control-flow) interactions with the set of external (performance) interactions revealed that those features that interact internally also interact externally (Sec. 4.3), which is in line with our expectation. Using the identified relation, we were able to substantially shrink the search space of performance feature interactions (Sec. 5). Furthermore, we made first steps towards developing a predictor for identifying features that are likely to interact externally based on the set of internal interactions, although, with mostly negative results (Sec. 3.5).

The key difference of this study to previous work is that we do not pick one type of interaction and study its properties, but rather we take two different types of interactions (external and internal) and study their relation. More specifically, we investigate the relation among control-flow (Sec. 2.1) and performance (Sec. 2.2) interactions, and discuss the implications of our findings (Sec. 5). To the best of our knowledge, this is the first case study that analyzed both the external and the internal feature interactions for the same systems, investigated possible relations between these two types of interactions, and as a result provided the following contributions:

[1] `https://tls.mbed.org/` `https://www.sqlite.org/`
[2] `https://openvpn.net/` `https://www.mozilla.org/`

- We define a relation between control-flow and performance interactions based on the features these interactions concern, and we discuss the plausibility of this relation.
- We define a conceptual framework for exploring the relation between internal and external interactions.
- As a first case study of its kind, we explore and confirm the relation between control-flow and performance feature interactions, based on two real-world highly-configurable subject systems.
- We discuss the implications of our findings for performance prediction of highly-configurable systems.

2 Internal and External Feature Interactions

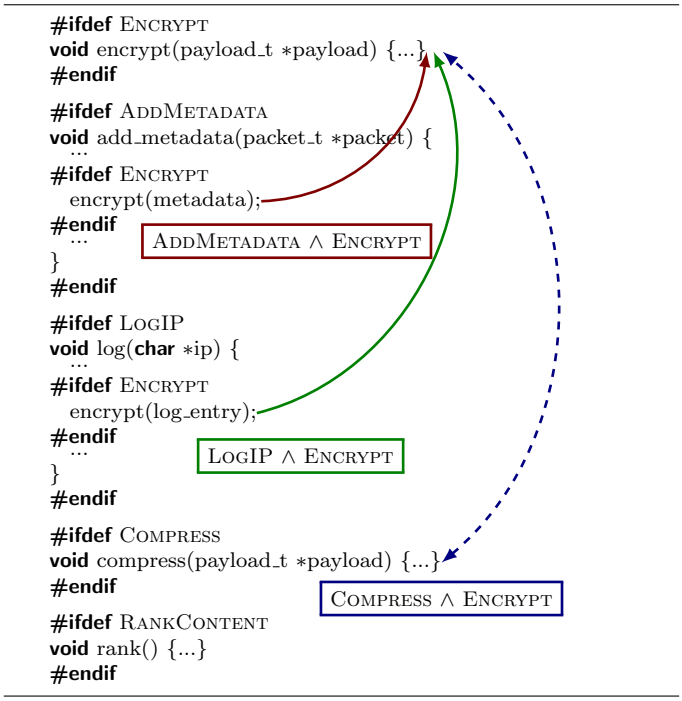
To illustrate how features may interact internally and externally and how these interactions can be related, we use a simple example of an audio streaming system with five optional features: Compress compresses the audio stream; Encrypt encrypts data; AddMetadata adds data about the stream quality, description of the audio content, information about its authors, etc., to the stream; LogIP logs IPs of the users receiving the stream; RankContent ranks the audio content according to its popularity. The performance of the system is measured by the maximum number of users that can simultaneously receive an audio stream without the system becoming overloaded.

2.1 Control-Flow Interactions (Internal)

In Figure 1a, we illustrate an excerpt of the implementation of the audio streaming system. The code of each feature is delimited using C preprocessor #ifdef annotations. We denote internal interactions among features with arrows. The boxes on the arrows contain *presence conditions* for the corresponding interactions (von Rhein et al, 2015), that is, which features must be enabled (or disabled) for the interaction to take place. For example, if both features AddMetadata and Encrypt are enabled, then metadata are encrypted along with the audio data. For this purpose, AddMetadata calls the encryption function of feature Encrypt (denoted by the solid red arrow). Consequently, there is a control-flow interaction between these two features.

Likewise, there is a control-flow interaction between features LogIP and Encrypt (denoted by the dashed green arrow), since the log entries are encrypted if both features are enabled.

Finally, an internal interaction exists between features Compress and Encrypt (denoted by the dotted blue arrow). This is a data-flow interaction, because both features operate on the same resource (i.e., the audio stream).



(a) Control-flow (solid line) and data-flow (dashed line) interactions interactions in the audio streaming system.

$$100 - 15\cdot\text{Compress} - 15\cdot\text{Encrypt} - 5\cdot\text{AddMetadata} - 5\cdot\text{RankContent} - 5\cdot\text{LogIP} - 5\cdot\boxed{\text{AddMetadata}\cdot\text{Encrypt}} + 10\cdot\boxed{\text{Compress}\cdot\text{Encrypt}}$$

feature interactions

(b) A performance influence model with performance interactions.

Fig. 1: Interactions in the audio streaming system.

2.2 Performance Interactions (External)

In Figure 1b, we show a *performance influence model* (Sec. 3.3) of the audio stream system. For a given system configuration, the model can predict the maximum number of users that can simultaneously receive an audio stream without overloading the system (here, we assume that the model is 100 % accurate). To calculate the predicted value, we substitute 1 for the names of the enabled features and 0 for the names of all disabled features. Then, we evaluate the arithmetic expression. For example, for the configuration with

feature Compress enabled and the rest of the features disabled the system can reliably serve $100 - 15 \cdot 1 - 15 \cdot 0 + \cdots + 10 \cdot 0 = 85$ users.

The individual terms of the model (i.e., the summands) describe the influence of individual features *as well as* of their interactions on the performance of the system. The first term (100) describes the performance of the base configuration (with all features disabled). The second term ($-15 \cdot$Compress) describes the influence of feature Compress on the performance *relative* to the performance of the base system. Thus, the computationally expensive feature Compress reduces the base performance by 15.

The terms containing more than one feature (denoted by boxes in Figure 1b) describe the influences of the interactions among the involved features on performance. For example, enabling both features AddMetadata and Encrypt makes the system encrypt not only the audio stream, but also the metadata that are added to the stream. This results in a computational overhead, which reduces the system's performance by 5 users that can be served.

In our example, we assume that encrypting a small string containing an IP address is so fast that this has no measurable effect on the performance of the system. Therefore, there is no a performance interaction between features LogIP and Encrypt. Consequently, there is no a corresponding term in our performance influence model.

The last term in the model describes an interaction between features Compress and Encrypt with a positive influence of the performance. Each of the two features individually has a negative influence of -15 on the system performance, but encryption is faster if the data were compressed before. Therefore, the combined influence of both features on performance is less than the sum of their individual influences: $-15 - 15 + 10 = -20$ and not -30.

Finally, feature RankContent as well as all other possible feature combinations have no measurable influence on performance and, therefore, they are not in the performance influence model.

2.3 Relating Control-Flow and Performance Interactions

Table 1 summarizes the control-flow and the corresponding performance interactions from our example (Fig. 1). The feature combinations (AddMetadata, Encrypt) and (Compress, Encrypt) give rise to both control-flow and performance interactions. Based on our knowledge about the implementation, we can explain the causal relation between the control-flow and performance interactions captured by these feature combinations: The call to the computationally expensive encryption functionality (a control-flow interaction) leads to the performance decrease in the configurations containing the features that implement and use the encryption functionality (i.e., a performance interaction between these features occurs). Notice that the related control-flow and performance interaction involve exactly the same features, so we can also relate them based on the features they involve. However, the mere presence of control flow among features does not always indicate the presence of a perfor-

Table 1: A list of interacting features from Figure 1. It illustrates which of the features interact internally (control-flow interaction), externally (performance interaction), or both.

Interacting Features	Control flow	Performance
AddMetadata, Encrypt	✓	✓
Compress, Encrypt	✓	✓
LogIp, Encrypt	✓	–

mance interaction. For example, the control-flow interaction between features LogIp and Encrypt has no corresponding performance interaction. So, it is an open question to what extent a presence of a control-flow interaction can be used as an *indicator* for a potentially existing performance interaction.

Also note that from 26^3 feature combinations possible in the audio streaming system only three combinations give rise to feature interactions. All remaining feature combinations can be ignored by an interaction detection technique, because features in these combinations do not interact.

In what follows, we investigate *to what extent* a relation between control-flow and performance interactions exists in a real-world setting. Furthermore, we define and evaluate a predictor that uses control-flow interactions to predict potential performance interactions. With such a predictor in place, we could make interaction detection more efficient and accurate, which would be a valuable contribution to research fields, such as optimization of non-functional properties, combinatorial testing, and sampling techniques.

3 Research Questions and Conceptual Framework

In our study, we address the following research questions:

- **RQ1:** Do control-flow feature interactions and performance feature interactions relate (in terms of the definition of Section 3.4)?
- **RQ2:** If a relation exists, can it be effectively leveraged to improve existing techniques for detecting external feature interactions or even to predict external feature interactions based on internal ones?

Before we can answer these questions, we have to decide on methods and tools that we will use in our study and how to combine them in a conceptual framework. Using this conceptual framework, we will then study relations among internal and external interactions and answer the research questions. Particularly, we have to choose a suitable research method, specify how we identify control-flow and performance interactions, define what a relation between these two types of interactions exactly is, and describe how we want to leverage it. Next, we describe this conceptual framework.

[3] 10 combinations with 2 features, 10 with 3, 5 with 4, and 1 with 5.

3.1 Research Method

We use the case study research method to explore a relation between control-flow and performance interactions. Shull et al (2007) defines a case study as an "initial investigation of some phenomena". The relation, which we explore in our work, is novel and it is studied for the first time. Therefore, our study qualifies as an initial investigation of a phenomena.

Yin (2003) has a broader definition of a case study as "an empirical inquiry that investigates a contemporary phenomenon within its real-life context." It is easy to construct an artificial example with a clearly existing relation between control-flow and performance interactions (cf. Sec. 2.3), but this would say nothing about the existence of this phenomenon in real-world configurable systems. The goal of our study is to investigate whether there is or may be such a relation between control-flow and performance interactions in real-world configurable systems, which matches exactly the definition by Yin.

Finally, Flyvbjerg (2006) states that "case studies offer in-depth understanding of how and why certain phenomena occur, and can reveal the mechanisms by which cause–effect relationships occur". In our study, we want to obtain deep insights into the nature of the relation between control-flow and performance feature interactions and not only report statistics. By focusing on two systems, we aim at increasing the internal validity of the study, because, this way, we can better identify and control confounding effects that may vary from one subject system to another (e.g., architecture, size of features). Moreover, our study involves bleeding-edge techniques for detecting control-flow and performance interactions in highly configurable systems that are technically challenging and cannot be easily applied to a large number of non-trivial real-world systems. For example, information about the variability in the configurable system that is provided in its documentation is often outdated, so compiling all configurations of the system becomes a tedious try-and-error process.

Based on all these considerations we have chosen the case study as our research method. In our case study, we go through the following steps: We take two real-world highly configurable systems and identify control-flow and performance interactions in these systems; then, we examine if the identified control-flow and performance interactions can be related based on the features that occur in them; finally, we evaluate predictors for performance interactions based on these relations. Next, we describe these steps in more detail and give an overview of the subject systems.

3.2 Identifying Control-Flow Interactions

To identify control-flow interactions, we use a variability-aware call-graph analysis (Ferreira et al, 2015) implemented in TypeChef[4] that identifies function calls among features implemented with preprocessor annotations (Fig. 1a).

[4] `http://fosd.net/TypeChef/`

The central idea of a variability-aware analysis is to achieve efficiency by analyzing code parts that are shared by multiple system configurations only once. This is achieved by analyzing the source code of the system that still contains variability (e.g., the code with preprocessor annotations in Figure 1a), as opposed to analyzing the source code of individual configurations, which may be exponentially many in the number of features. A variability-aware call-graph analysis provides an efficient way to identify function calls among features of a highly-configurable system and makes the detection of internal interactions feasible.

The underlying data structure for the analysis is *the variable abstract syntax tree*. Similar to an abstract syntax tree (AST), a variable AST provides an abstraction of the source code that can be efficiently analyzed, but it also provides information on which part of the code belongs to which features (in the form of presence conditions). Using this information, a call-graph analysis can identify, for each function call, which feature is the caller and which feature is the callee. Furthermore, the analysis can identify a presence condition for each call, that is, which features must be enabled (or disabled) for the call to take place at runtime. For example, in Figure 1a, the call from feature AddMetadata to feature Encrypt (solid red arrow) occurs only if both features AddMetadata and Encrypt are enabled (denoted by the presence condition in the box under the arrow). Due to the static nature of the technique, the collected information about the calls may be an overapproximation, but this is a problem with any static analysis approach. The current implementation of the analysis also uses pointer analysis to increase the accuracy of the call graph (Ferreira et al, 2015).

3.3 Identifying Performance Interactions

For detecting performance feature interactions, we learn performance influence models (Fig. 1b). As discussed in Section 2.2, a performance influence model captures the influences of individual features and their interactions on performance of a configurable system. We learn performance influence models using the tool SPL Conqueror,[5] which implements a state-of-the-art machine learning algorithm based on multivariable regression and forward feature selection (Siegmund et al, 2015). The algorithm takes as input a sample of system configurations and corresponding performance measurements. The accuracy of the learned performance influence model depends, among other factors, on how representative the sampled configurations are for the entire configuration space. To get a performance influence model of the highest possible accuracy, and, consequently, to detect feature interactions as precise as possible (i.e., to obtain the ground truth), we measured not a sample but all configurations of the subject system and used these measurements as the algorithm input. The performance measurements were done using a standard benchmark.

[5] `http://fosd.net/SPLConqueror/`

To build a performance influence model, SPL Conqueror starts with calculating a set of features and their combinations that can be included in the model to reduce the model's prediction error. Compress · Encrypt in Figure 1b, for example, is a feature combination that has been eventually included in the model during the learning process. The algorithm iterates over the set of features and their combinations and selects one element of the set (a candidate) that explains variations in the performance of the system best; that is, the element yielding the model's lowest prediction error, when incorporated into the model. The coefficients in the model (e.g., 10 for the candidate Compress · Encrypt in Figure 1b) are learned using multivariable regression by treating candidates as independent variables and the measured performance as dependent variable. The selection of candidates continues until either a predefined accuracy is reached or all features and feature combinations that could reduce the prediction error of the model have been considered. For a more in-depth description of the algorithm, we refer the reader to previous work (Siegmund et al, 2015).

3.4 Relating Control-Flow and Performance Interactions

After we have identified the internal (control-flow) interactions, the question is what we can learn from them regarding external (performance) interactions. To answer this question, we relate the control-flow interactions and performance interactions based on the features involved in them, as we explained it in our example in Section 2.3. The goal is to find out if the features involved in performance interactions also occur in one or more internal interactions and vice versa. This is a feasibility check to see if the interactions can be related based on the features' occurrence at all. That is, if we find no interactions that can be related in this way, this would mean that it is impossible to define any relation between interactions based on the corresponding feature occurrences in these interactions.

We define a performance interaction i_p and a control-flow interaction i_c as related if $\mathit{features}(i_p) \subseteq \mathit{features}(i_c)$ or if $\mathit{features}(i_p) \supseteq \mathit{features}(i_c)$, where $\mathit{features}(i)$ is the set of features that contribute to the interaction i.

Furthermore, for each related pair of interactions, we determine how similar the interactions are (i.e., if they contain exactly the same features or if they also contain features that are present only in one of them). The similarity of the related interactions can be interpreted as the strength of their relation: the higher the similarity, the higher the strength of the relation. We calculate the similarity of interactions using the Jaccard index J (Jaccard, 1912):

$$J(i_p, i_c) = \frac{\mathit{features}(i_p) \cap \mathit{features}(i_c)}{\mathit{features}(i_p) \cup \mathit{features}(i_c)}$$

where $\mathit{features}(i)$ is the set of features involved in the interaction i. The Jaccard index equals 1 if both interactions involve exactly the same features and is less than 1 otherwise.

3.5 Predicting Performance Interactions

If we find a relation between control-flow and performance feature interactions as defined in Section 3.4, the question is whether we can use this relation to predict performance feature interactions.

One method is to build on our argumentation in Section 2.3 and to assume that every control-flow interaction corresponds to an existing performance interactions. Of course, we already know that there may be control-flow interactions without corresponding performance interactions. Nevertheless, it is an open question *how* accurate this simple method can be if applied to a real-world system.

We can also use a more advanced method based on reoccurring feature combinations in control-flow interactions: We argue that, if a set of features occurs in multiple control-flow feature interactions, then this set of features is also likely to give rise to one or more external interactions. The rationale behind this argument is that, if a set of features is involved in many control-flow feature interactions, then chances are high that it is also involved in performance interactions, because the accumulated influence of the control-flow interactions on performance have a measurable effect.

We use *frequent item set mining* (Borgelt, 2012) as a method to identify such frequent feature sets. This method was successfully used as a general pattern mining method (Maqbool and Babri, 2007; Qiao et al, 2013). In terms of frequent item set mining, we refer to a feature as an *item*. For example, features such as AddMetadata and Encrypt in the running example in Fig. 1 are items. The set of all items (all features) is the *item base* B (e.g, the item base of the running example contains all its features). A subset of the item base $I \subseteq B$ is an *item set* that corresponds to a feature combination. An item set (i.e., a feature combination) that denotes an internal interaction in a system is a *transaction* $t \in T$, where T is a set of transactions. In the running example, a set of features {AddMetadata, Encrypt} is an item set and it is also a transaction, because these two features interact at the control-flow level (Fig. 1a). Based on these definitions, we define the *support* (a.k.a. absolute frequency) s of an item set I: $s = |\{t : t \in T \wedge I \subseteq t\}|$. In Fig. 1a, the item set {Encrypt} has a support value of three, because it is a subset of every transaction (i.e., control-flow feature interaction) in the running example. Item set {AddMetadata, Encrypt} has a support value of 1, because there is only one control-flow feature interaction involving these features. The support value and a threshold $E \in [0, \infty)$ is used to decide which of the item sets are considered frequent: All item sets with the support value $s \geq E$ are *frequent* item sets. Based on our hypothesis, frequent item sets predict external feature interactions. In our analysis, we also ignore item sets with only one item (feature), because a feature interaction requires at least two different features. We use an implementation of the Apriori algorithm from the Orange library[6] to calculate the support value.

[6] `http://orange.biolab.si/`

3.6 Subject Systems

The case study was conducted using two real-world highly-configurable software systems: the MBEDTLS library implementing the transport security network protocol TLS/SSL and a SQL database engine SQLITE. The initial use case for the systems was the embedded domain, but now they are also used in non-embedded projects, such as OPENVPN and FIREFOX.

Similar to a large number of other real-world highly configurable systems, the subject systems are written in C using C-preprocessor directives to implement compile-time variable features. MBEDTLS has 97 and SQLITE has 12 features, which results in 1921 and 1533 configurations respectively. The configurations are obtained using a SAT solver (built into SPL CONQUEROR) by computing all feature combinations that satisfy the constraints in the feature models (see the following subsections) of the subject systems. MBEDTLS comprises 50 K and SQLITE 195 K lines of code. Both systems have a highly modular architecture, which is thoroughly documented along with the corresponding preprocessor macro names allowing relatively easy matching of code to the corresponding modules and submodules.

The manageable number of features and configurations makes these systems especially suitable for an in-depth qualitative case study: For example, it allows us to measure performance of all configurations and use these measurements in turn to identify a baseline of performance feature interactions in reasonable time. The feasible number of resulting feature interactions allows us to verify that every one of them actually exists in the system. Furthermore, the size of the subject systems allows us to manually inspect and understand the structure of the systems and the interplay of their features (Sec. 4, 5). Altogether, the manageable size of the subject systems is a prerequisite for the internal validity of our qualitative case study.

Features and Feature Model of MBEDTLS

At the top level, MBEDTLS consists of modules, such as *Cipher*, *Public Key*, *Hashing*. Each module implements the corresponding algorithms and protocols. For example, the *Cipher* module includes submodules that implement cipher algorithms, such as AES, DES, and ARC4. Submodules implement the features of the system. The cipher-algorithm features can be combined with other features, such as hash algorithms and public-key implementations, to provide an encryption protocol. We used the original documentation and manual code inspection to construct a feature model for MBEDTLS version 2.2.1.

Features and Feature Model of SQLITE

SQLITE consists of a *Core* providing a C-language interface and being responsible for executing compiled SQL code, an *SQL Compiler*, and a *Backend* providing the low-level implementation of the database. A user can configure

the operation of these modules by enabling or disabling their features through compile-time options. For example, *Core* can be configured to operate safely in a multithreaded environment by enabling the SQLITE_THREADSAFE feature. We studied the documentation and the source code of version 3.16.2 to construct a feature model.

Performance Measurements of MBEDTLS

The primary application of MBEDTLS is the encryption of data transmitted over a TCP/IP network. Ensuring fast and secure data transfer is commonly considered an important property of communication networks, such as the Internet. So, the time required to encrypt data and transfer them over the network is an important non-functional property of MBEDTLS. Measuring the time required by encryption alone is not representative, because different configurations may produce different amounts of payload (e.g., due to data compression and different amounts of generated metadata) influencing the transmission time. Therefore, we defined the performance measure for a configuration of MBEDTLS as the amount of time (in seconds) required to encrypt and successfully transmit a fixed amount of input data.

To detect performance feature interactions reliably based on performance benchmarks, it must be ensured that every feature included in a configuration is invoked during the benchmark of this configuration. Otherwise, the influence of features and their interactions on performance cannot be deduced from the benchmark results. The original automated test framework of MBEDTLS includes tests that check the library's functionality in a client-server environment and is suitable to serve as a typical benchmark suite. During the tests, the functionality of every feature in the configuration is tested, that is, every feature is actually invoked.

We used 2 GB of random data as input to ensure that the fastest configuration requires, at least, five seconds for transmission and to mitigate the influence of warm-up effects on the result. We repeated the benchmark 30 times to further reduce the influence of measurement bias. To exclude the influence of network latencies, we ran the benchmark locally using the local network interface.

Performance Measurements of SQLITE

The developers of SQLITE provide a performance benchmark that measures time required by the database to execute a set of queries.[7] The original benchmark is not compatible with the latest version of the system that we use, so we used it as guidance to create a new compatible benchmark. While constructing the benchmark we made sure that the features of SQLITE are actually invoked during the benchmarking process. Our benchmark measures the execution time

[7] http://sqlite.org/speed.html

in seconds. To reduce the influence of warm-up effects and measurement bias, the benchmark runs, at least, 25 seconds and every run is repeated 30 times.

The benchmarks for both systems were conducted on an Intel i5-4590, 16 GB RAM, 256 GB SSD, Ubuntu 16.04.

4 Results

In this section, we describe the results of applying of our conceptual framework (Sec. 3) to subject systems. To increase internal validity, we report in Sections 4.2 and 4.1 in detail how we identified performance and control-flow interactions. Based on these data, we report the identified relation between performance and control-flow interactions (Sec. 4.3), which we use to answer RQ1 in Section 5, and how this relation can be leveraged (Sec. 4.4), which we use to answer RQ2 in Section 5.

4.1 Control-Flow Interactions

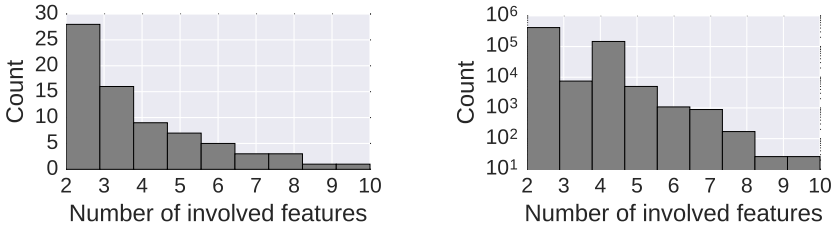
In this section, we report on control-flow interactions that we identified in MBEDTLS and SQLITE using variability-aware call-graph analysis implemented in TYPECHEF (Sec. 3.2). Furthermore, we explain the limitations of TYPECHEF that prevent it from detecting all control-flow interactions (e.g., in cases where one feature uses function pointers to call another). We discuss how we addressed these limitations to increase the internal validity of the study by manually identifying control-flow interactions missed by TYPECHEF.

MBEDTLS

From 761 992 function calls in the system, we detected 575 560 control-flow feature interactions. This number of interactions includes duplicate interactions that appear if the corresponding function call between features occurs in multiple locations in the code. The number of unique control-flow interactions is 73.

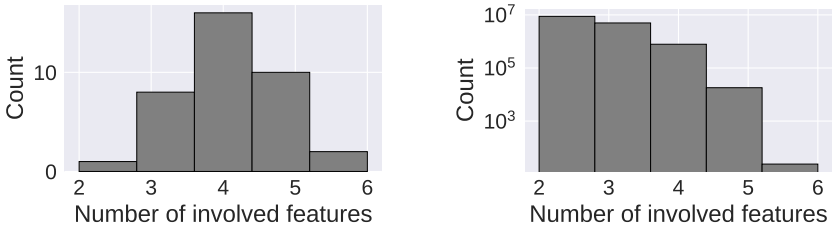
Notably, among the unique control-flow interactions, there are interactions with up to 10 features, but most unique interactions involve only two features (Fig. 2a). If we also consider the duplicates (Fig. 2b), the overall picture stays largely the same: Only the number of interactions involving four features becomes larger than the number of those involving three features.

While manually exploring the source code of MBEDTLS, we found that cipher, mode, and hash algorithms call each other indirectly, using function pointers. This indirection was introduced by the designers of the library to decouple the algorithms and to make their concrete implementations interchangeable. TYPECHEF would need to be extended with a variability-aware, inter-procedural data-flow analysis to identify which features interact using indirect function calls. Being aware of this technical limitation of TYPECHEF,



(a) Unique interactions only (b) All interactions

Fig. 2: mbedTLS: counts of features in control-flow interactions.



(a) Unique interactions only (b) All interactions

Fig. 3: SQLite: counts of features in control-flow interactions.

we added 11 indirect control-flow interactions that we collected while manually exploring the code to the set of interactions. In our manual exploration of the source code, we relied on our understanding of the subject systems' structure and interplay of the features. For example, based on the knowledge, we knew that cipher, mode, and hash algorithms should closely work together. So, we looked for control flow among all features implementing these algorithms. The total number of the identified unique control-flow interactions is 84 (73 were found using TypeChef and 11 manually). It would be infeasible to find manually all instances of indirect control-flow interactions, so their exact number (including duplicates) is unknown. We discuss the corresponding threats to validity in Section 6.

SQLite

From over 14 587 337 function calls in the system, we detected 14 587 335 control-flow feature interactions. That is, all but two function calls involved more than one feature. The number of unique control-flow interactions is 37.

In contrast to mbedTLS, most unique interactions involve 4 features, and there are interactions with up to 6 features (Fig. 3a). Although, if we also consider duplicates (Fig. 3b), the picture becomes similar to that in mbedTLS:

Interactions among 2 features prevail and the count of interactions decreases with the increasing number of involved features.

While manually inspecting the code of SQLite, we found that the option SQLITE_DEFAULT_MEMSTATUS (which is used by TypeChef to identify the code belonging to the feature DEFAULT_MEMSTATUS) is used to set a Boolean variable at compile-time. This variable is then used at runtime to check if feature DEFAULT_MEMSTATUS is enabled or disabled. This way, the feature can be enabled or disabled at runtime. Again, TypeChef would need a data-flow analysis to trace the connection the preprocessor macro to the corresponding Boolean variable to detect control-flow interactions in which feature DEFAULT_MEMSTATUS is involved. By further exploring the code, we identified two control-flow interactions of this kind and added them to the set of automatically detected interactions. Therefore, the total number of the identified unique control-flow interactions is 39.

Summary. Overall, we identified 575 571 control-flow interactions in mbedTLS among which 84 were unique. Some interactions involve up to 10 features, but most interactions are between 2 features. For SQLite, we identified 14 587 335 control-flow interactions, with 39 unique. Due to technical limitations of TypeChef, indirect control-flow interactions in mbedTLS and interactions induced by runtime variability in SQLite could not be detected by TypeChef. We manually inspected the source code to collect these interactions.

4.2 Performance Interactions

In this section, we report on performance interactions that we identified in mbedTLS and SQLite. Using domain knowledge and manual inspection of the source code, we confirm that the identified interactions actually exist and thereby increasing internal validity of our study.

We used the performance benchmark results obtained using SPL Conqueror (cf. Sec. 3.6) as input data to identify performance interactions in mbedTLS and SQLite, as described in Section 3.3. Table 2 lists for both systems the performance interactions and their influences on performance of the systems in seconds. The *negative values* in the influence column denote *positive influences* of the corresponding interactions on performance. That is, they denote how much less time a configuration that includes them would need to execute the benchmark.

The mean standard deviation for the performance measurements of mbedTLS is 0.42 s. Therefore, we classified all interactions with the absolute influences less than this value as noise and discarded them. From the remaining 16 interactions, 11 are interactions between two features; and five are interactions among three features. The mean standard deviation for the performance measurements of SQLite is 0.09 s. The influences of the three identified interactions for the system are much higher and, therefore, are unlikely to be

Table 2: Performance interactions and their influences on performance of the systems in seconds.

	ID	Influence (sec)	Performance Interaction (features involved)
MBEDTLS	1	10.73	CIPHER_MODE_CBC, SHA256_C
	2	-9.71	AES_C, AESNI_C
	3	8.53	AESNI_C, SSL_CBC_RECORD_SPLITTING
	4	6.93	CIPHER_MODE_STREAM, AESNI_C
	5	6.08	SHA256_C, CIPHER_MODE_STREAM
	6	5.75	AES_C, AESNI_C, GCM_C
	7	3.49	CIPHER_MODE_CBC, SHA256_C, SHA256_SMALLER
	8	3.45	SHA256_C, CIPHER_MODE_STREAM, SHA256_SMALLER
	9	3.44	SHA256_C, AESNI_C, CIPHER_MODE_STREAM
	10	3.14	CIPHER_MODE_CBC, RIPEMD160_C
	11	-2.97	AES_C, GCM_C
	12	-2.84	CIPHER_MODE_STREAM, MD5_C
	13	1.93	AESNI_C, CAMELLIA_C
	14	1.68	CIPHER_MODE_CBC, SHA1_C
	15	1.60	CIPHER_MODE_STREAM, AESNI_C, MD5_C
	16	1.51	RIPEMD160_C, CIPHER_MODE_STREAM
SQLITE	1	1.50	DEFAULT_MEMSTATUS, THREADSAFE
	2	1.47	MEMDEBUG, THREADSAFE
	3	1.41	DEFAULT_MEMSTATUS, MEMDEBUG, THREADSAFE

To relate the influences to configuration run times, note that the fastest MBEDTLS configuration completed its benchmark in 6.7 seconds and the fastest SQLITE configuration completed its benchmark in 26.7 seconds.

noise. Two of the interactions are interactions between two features and one is an interaction between three features.

MBEDTLS

All identified interactions in MBEDTLS are among features implementing different ciphers, block cipher modes of operation (simply "modes"), and cryptographic hash functions. This is plausible, because these three types of algorithms work tightly together to implement an encryption protocol. Ciphers (e.g., AES) are used to encrypt data, modes (e.g., CBC) are used in combination with block ciphers to encrypt amounts of data larger than a block (i.e., a fixed amount of data a block cipher can operate on; 128 bit for AES), and cryptographic hash functions (e.g., SHA) are used with modes to implement authentication and to ensure data integrity.

To confirm that the identified performance interactions actually result from the interplay of the corresponding features, we manually inspected the source code of MBEDTLS. Next, we present the results of this code inspection.

Interaction 1 in Table 2 arises between a mode (CBC) and a hash function (SHA256). CBC uses hashing extensively to calculate keyed-hash message authentication code (HMAC). SHA256 is computationally more expensive than, for example, MD5; therefore, this combination with the mode has a negative

influence of 10.73 seconds on performance. Interactions 5, 7, 8, 10, 12, 13, 14, and 16 have a similar cause and explanation. In addition to a mode and a hash function, interactions 7 and 8 also include the feature SHA256_SMALLER, which denotes that an implementation of SHA256 with smaller binary footprint was used. However, this implementation also has a lower performance, which leads to the negative influence of this interaction on performance. Interaction 12 has a positive influence on performance of using a mode (stream mode, in this case) with a less computationally complex (but also less secure) MD5 hash function. In interaction 13, the AES cipher is used as a hash function in combination with the Camellia cipher.

Interaction 2 arises from the usage of the AES cipher for encryption in combination with a native implementation of the AES algorithm in assembler (AESNI). The native implementation makes encryption faster, so this interaction has a positive influence of 9.71 seconds on performance.

Interaction 3 arises from the usage of the AES cipher for encryption in combination with an implementation of the CBC mode that includes a record splitting algorithm. This algorithm is a countermeasure against the BEAST attack on the SSL algorithm (Duong and Rizzo, 2011). The way record splitting is implemented increases the number of packets to be transmitted (compared to the number of packets without this countermeasure). The increased number of packets results, in turn, in a negative influence on performance.

Interactions 4, 6, 9, 11, and 15 arise from the influence of further combinations of ciphers, modes, and hash functions on performance, similar to the first interaction.

SQLite

All performance interactions in SQLite include the feature THREADSAFE. This is plausible, because THREADSAFE is a crosscutting feature that adds the mutex and thread-safety logic to all unsafe regions in the code. This additional thread-safety code imposes a runtime overhead and makes the benchmarks for the configurations containing it run longer. We inspected the code of SQLite and confirmed that both features DEFAULT_MEMSTATUS and MEMDEBUG retrieve a mutex (i.e., use THREADSAFE feature) at a certain stage of operation that results in interaction among THREADSAFE and these features.

Summary. Overall, we identified 16 performance interactions in mbedTLS. 11 of them occur between 2 features and 5 among 3 features. In SQLite, we identified 3 performance interactions. 2 interactions between 2 features and 1 among 3 features. Using domain knowledge and manual inspection of the source code, we identified the cause of all interactions and thereby confirmed that they actually exist in the systems and are caused by the interplay of the corresponding features.

4.3 Relating Interactions

In this section, we describe how we identified relations among performance and control-flow interactions that we described in Sections 4.2 and 4.1

Performance Interactions → Control-Flow Interactions

Using the relation definition $features(i_p) \subseteq features(i_c)$ (Section 3.4), for each performance interaction, we identified all unique related control-flow interactions (i.e., all control-flow interactions involving exactly the same features as the performance interaction). Furthermore, for each pair of related interactions, we calculated the Jaccard index (Section 3.4), which denotes how similar the interactions are (the index equals 1 if both interactions involve exactly the same features and is less than 1 otherwise).

Table 3 summarizes the results. For each performance interaction, it shows the number of the related control-flow interactions and the mean of all Jaccard indexes calculated for these relations. The numbers show that there is a relation between every performance interaction and, at least, one control-flow interaction. The Jaccard indexes show that the related control-flow interactions that were automatically detected by TypeChef (those are the same as the interactions with the number of relations greater than 1 in Table 3), in-

Table 3: Performance interactions, the number of the control-flow interactions related to them, and the mean value of the corresponding Jaccard indexes.

	ID	Performance Interaction (features involved)	Relations	Jaccard (mean)
MBEDTLS	1	CIPHER_MODE_CBC, SHA256_C	1	1.00
	2	AES_C, AESNI_C	10	0.53
	3	AESNI_C, SSL_CBC_RECORD_SPLITTING	2	0.38
	4	CIPHER_MODE_STREAM, AESNI_C	1	1.00
	5	SHA256_C, CIPHER_MODE_STREAM	1	1.00
	6	AES_C, AESNI_C, GCM_C	4	0.53
	7	CIPHER_MODE_CBC, SHA256_C, SHA256_SMALLER	1	1.00
	8	SHA256_C, CIPHER_MODE_STREAM, SHA256_SMALLER	1	1.00
	9	SHA256_C, AESNI_C, CIPHER_MODE_STREAM	1	1.00
	10	CIPHER_MODE_CBC, RIPEMD160_C	1	1.00
	11	AES_C, GCM_C	13	0.40
	12	CIPHER_MODE_STREAM, MD5_C	1	1.00
	13	AESNI_C, CAMELLIA_C	4	0.35
	14	CIPHER_MODE_CBC, SHA1_C	1	1.00
	15	CIPHER_MODE_STREAM, AESNI_C, MD5_C	1	1.00
	16	RIPEMD160_C, CIPHER_MODE_STREAM	1	1.00
SQLITE	1	DEFAULT_MEMSTATUS, THREADSAFE	1	1.00
	2	MEMDEBUG, THREADSAFE	16	0.45
	3	DEFAULT_MEMSTATUS, MEMDEBUG, THREADSAFE	1	1.00

volve, on average, twice as many or even more features than there are in the corresponding performance interactions.

Control-Flow Interactions → Performance Interactions

Using the relation definition $features(i_p) \supseteq features(i_c)$ (Section 3.4), for each control-flow interaction, we identified all related performance interactions (i.e., all performance interactions involving exactly the same features as the control-flow interaction).

Table 4 summarizes the results. For MBEDTLS, among the 84 unique control-flow interactions, we found 15 interactions that have one or more related performance interactions. For SQLITE, among the 39 unique control-flow interactions, we found 2 interactions that have one or more related performance interactions. The Jaccard indexes show that the related performance interactions that were automatically detected by TYPECHEF (interactions 1–4 for MBEDTLS) involve mostly the same features as the corresponding control-flow interactions. The manually added control-flow interactions (interactions 5–15 for MBEDTLS and all interactions for SQLITE) match exactly the related performance interactions.

Summary. We found a relation between every of the 16 identified performance interactions and one or more control-flow interactions. The Jaccard indexes show that the automatically detected control-flow interactions do not generally contain exactly the same features as the related performance interactions, that is, the automatically detected control-flow interactions involve, on average, twice as many features as the corresponding performance interactions. The manually added control-flow interactions involve exactly the same features as the corresponding performance interactions.

4.4 Predicting Performance Interactions

MBEDTLS: *Direct Matching*

As we describe in Section 3.5, one prediction method is to assume that every control-flow interaction induces a performance interaction that involves exactly the same features. In MBEDTLS, from the 73 automatically identified unique control-flow interactions there are three—interactions 1, 3, and 4 in Table 4—that have exactly the same features as the related performance interactions 2, 6, and 11 in Table 3. That is, three of the 16 performance interactions could be predicted by the direct matching. Therefore, the precision of the direct matching is 4.11 % and the recall is 18.75 %. If we also incorporate the 11 indirect control-flow interactions, which we identified by manually inspecting the code, the total number of matching control-flow interactions becomes 14. Including indirect control-flow interactions increases the precision and recall to 16.7 % and 51.85 % respectively.

SQLite: *Direct Matching*

In SQLite, there are no automatically identified unique control-flow interactions that match exactly any of the performance interactions. Including the manually added control-flow interactions gives the prediction precision of 5.13 % and the recall of 67 %.

mbedTLS: *Frequent Item Sets*

Using frequent item set analysis (cf. Sec. 3.5) on the set of control-flow interactions for mbedTLS, we found 44 item sets, of which we calculated the support values. The support values range from 11 % to 34 %, meaning that there are item sets occurring in 11 % to 34 % of all control-flow interactions.

Two of the found item sets match exactly the performance interactions 2 and 11 of Table 3. Notice that we ran the frequent item set analysis only on the automatically detected control-flow interactions. We were not able to run it on the indirect control-flow interactions, because then we would have to find every instance of such interaction manually, which is infeasible. Nevertheless, we incorporated the indirect control-flow interactions into further analysis by approximating their support values based on the distribution of support values for similar indirect interactions (see Sec. 6, for threats to validity). Among the 44 detected item sets, there are 33 item sets capturing interactions among ciphers, modes, and hash functions. We assigned support values to the indirect

Table 4: Control-flow interactions, the number of the related performance interactions, and the mean value of the corresponding Jaccard indexes. Control-flow interactions without related performance interactions are not listed.

	ID	Control-Flow Interaction (features involved)	Relations	Jaccard (mean)
mbedTLS	1	AES_C, AESNI_C	2	0.83
	2	GCM_C, AESNI_C	1	0.67
	3	GCM_C, AES_C	2	0.83
	4	GCM_C, AES_C, AESNI_C	1	1.00
	5	CIPHER_MODE_CBC, SHA256_C	1	1.00
	6	CIPHER_MODE_STREAM, AESNI_C	1	1.00
	7	SHA256_C, CIPHER_MODE_STREAM	1	1.00
	8	CIPHER_MODE_CBC, SHA256_C, SHA256_SMALLER	1	1.00
	9	SHA256_C, CIPHER_MODE_STREAM, SHA256_SMALLER	1	1.00
	10	SHA256_C, AESNI_C, CIPHER_MODE_STREAM	1	1.00
	11	CIPHER_MODE_CBC, RIPEMD160_C	1	1.00
	12	CIPHER_MODE_STREAM, MD5_C	1	1.00
	13	CIPHER_MODE_CBC, SHA1_C	1	1.00
	14	CIPHER_MODE_STREAM, AESNI_C, MD5_C	1	1.00
	15	RIPEMD160_C, CIPHER_MODE_STREAM	1	1.00
SQLite	1	DEFAULT_MEMSTATUS, THREADSAFE	1	1.00
	2	DEFAULT_MEMSTATUS, MEMDEBUG, THREADSAFE	1	1.00

control-flow interactions according to the distribution of the support values of these 33 item sets. That is, 6 % of the interactions were assigned a support value of 11 %, 3 % were assigned a support value of 12 %, and so on.

By varying the threshold E, as described in Section 3.5, we are able to decide which of the identified item sets are considered frequent. By setting the threshold to 0, we consider all identified item sets as frequent. When the threshold is increased the item sets with lower support values are not considered frequent anymore. For example, if we set the threshold to 15 % only 25 % of the identified item sets will have a higher support value and will be considered frequent. Changing the threshold this way allows us to observe its influence on the predictive power (i.e., precision and recall) of the frequent item sets.

To calculate how good the item sets are in predicting performance interactions, we compared how many of them denote the actually identified performance interactions (i.e., contain exactly the same features as the performance interactions). The low precision and recall values for MBEDTLS summarized in Table 5 show that our predictor based on the frequent item sets has only a low predictive power. Increasing the threshold value decreases the precision and recall of the predictor.

SQLITE: *Frequent Item Sets*

Applying the same frequent item set method to the control-flow interactions of SQLITE resulted in four frequent item sets with support values ranging from 20 % to 100 %. None of these frequent item sets matched the performance interactions. We could not approximate the distribution of the support values for the manually detected control-flow interactions, because they do not exhibit any commonalities with the calculated frequent item sets as it was the case for MBEDTLS.

Summary. We defined two predictors for performance interactions based on their relation with control-flow interactions. The first predictor is based on the assumption that every control-flow interaction induces a performance interaction that involves exactly the same features. The second predictor is based on the assumption that the recurring feature combinations in control-flow interactions capture the related performance interactions. The evaluation showed that both predictors have only low precision and recall values.

5 Discussion

Based on our results, we conclude that there is indeed a quantifiable relation between control-flow and performance interactions. We confirmed this by manually inspecting the code and by comparing which features are involved in the detected performance interactions and how these features interact at

the control-flow level. We found that features involved in performance interactions work closely together to implement the systems' functionality and thus also interact at the control-flow level. That is, the same features that are involved in performance interactions are also involved in control-flow interactions. Therefore, we can positively answer research question RQ1, which asked if control-flow feature interactions and performance feature interactions relate.

The relation we found among control-flow and performance feature interactions has implications for performance prediction techniques for highly-configurable systems. As we discussed in Section 4.3, the identified control-flow interactions capture the features that are involved in the performance interactions. Of course, we cannot identify these features precisely, because the same control-flow interactions also involve other features that are not involved in performance interactions (this is also a reason for direct matching prediction having low precision and recall; cf. Sec 4.4). Nevertheless, assuming that only the features from the identified control-flow interactions can give rise to a performance interaction considerably reduces the search space of the potential performance feature interactions, because otherwise we have to assume that any (valid) feature combination may give rise to a performance interaction. MBEDTLS has 134 057 valid feature combinations of two and three features, but the 84 identified unique control-flow interactions (Sec. 4.1) result in only 452 *potential* performance interactions (among two and three features). Notice that these include all 16 actually existing performance feature interactions that we identified. That is, we are able to shrink the search space of performance feature interactions by almost 300 times (452 instead of 134 057) without losing any of the actually existing performance feature interactions. Although, note that shrinking the search space for performance interactions this way is still a heuristic and may miss some interacting features. SQLITE has 524 valid feature combinations of two and three features and (based on the 39 identified unique control-flow interactions) only 131 *potential* performance interactions (among two and three features). These potential performance interactions also include all 3 actually existing performance interactions that we identified. That is, the search space shrinks by 4 times. These results have immediate consequences for performance prediction techniques based on machine learning and relying on sampling for building a training dataset: By exploiting our findings they can make sampling more focused on the configurations that potentially include interacting features, which may improve their prediction accuracy.

With respect to RQ2, which asked if relations between control-flow and performance interactions can be effectively leveraged to improve existing techniques for detecting external feature interactions or to predict external feature interactions based on internal ones, our results are twofold. The shrinkage of the search space of performance feature interactions can help to make performance prediction techniques more focused on potential feature interactions, which is a positive result. As to the predictors based on direct matching and frequent item sets, we obtained only low precision and recall values, which is effectively a negative result. One possible reason for this negative result is that the predictors rely solely on control-flow data, but features can also inter-

Table 5: Precision and recall values for the item sets as predictors for the performance interactions in MBEDTLS. (*) marks the precision and recall values for the item sets with incorporated indirect control-flow interactions.

Threshold	Precision	Recall	Precision*	Recall*
0	4.5	12.5	23.6	48.1
15	2.3	6.3	5.5	11.1
20	0	0	1.8	3.7

act via data flow. For example, they can exchange data through shared data structures. This interplay at the data-flow level can be interpreted as data-flow feature interactions (much like control-flow feature interactions, Fig. 1a), which may also induce performance interactions. For example, a feature may block other features by locking a shared data structure, which may have a negative influence on the performance of the system. Therefore, enriching the data used by the predictors with the information about data-flow interactions may increase their predictive power. So, a takeaway message here is that predictors should consider the interplay of features not only on the control-flow level, but also at the data-flow level, and other levels. Another reason may be that not all features involved in a control-flow interaction are also involved in a related performance interaction. The Jaccard index values in Table 3 show that only about half of the features in a control-flow interaction are also present in the related performance feature interaction. For example, the interaction (AES_C, AESNI_C) has the average Jaccard index of 0.46. This means that, on average, a related control-flow interaction has two other features additionally involved, in addition to features AES_C and AESNI_C. Both predictors for a given control-flow interaction are not able to distinguish among features that are involved in a related performance interaction and those that are not.

Further Observations

A further observation is related to the distribution of the number of features involved in the control-flow and performance interactions. For MBEDTLS, in most cases, interactions (both, control-flow and performance) involve two or three features. For SQLITE, in most cases, control-flow interactions involve four features, but this is only the case because every single control-flow interaction involves the two crosscutting features THREADSAFE and ENABLE_API_ARMOR. If we ignore these crosscutting features, the pictures becomes similar to MBEDTLS. The performance interactions in SQLITE involve two or three features as in MBEDTLS. From these data, we conclude that the frequency of interactions decreases with the growing number of the involved features. This shows that features tend to interact at the same rate (two or three features per interaction) independently of the type of the inter-

action (control-flow or performance). This is another indication for a relation between control-flow and performance interactions.

Finally, for MBEDTLS, we found that most of the frequent item sets that we identified in the control-flow interactions contain features from three groups of algorithms: ciphers, modes, and hashes. Even though most of the frequent item sets do not resemble existing performance interactions, they still capture the general pattern of the detected performance interactions, namely, that these interactions involve features from these three groups of algorithms. For SQLITE the frequent item sets capture the crosscutting features, such as THREADSAFE and ENABLE_API_ARMOR. The crosscutting feature THREADSAFE was involved in all identified performance interactions.

Summary

We showed for two real-world configurable systems that a relation between control-flow and performance interactions exists. Furthermore, we have shown that this finding has an immediate practical implication: Using the identified relations, we were able to shrink the search space of performance feature interactions by almost 300 times for MBEDTLS and 4 times for SQLITE. Reducing the search space is central for making techniques for detection of performance interactions and predicting performance of configurable systems more time efficient, because their algorithms will have to consider fewer potential performance interactions. In a sense, our approach boils down to a domain-specific, white-box dimension reduction technique for the underlying performance learning problem.

Furthermore, we constructed a predictor based on frequent items sets for predicting performance interactions based on their relations to control-flow interactions. This particular predictor showed a low precision and recall, though, which we attribute to the fact that the predictor relied only on one type of internal interactions. Constructing other predictors and performing fully fledged experiments with other types of internal interactions would have gone way beyond the goals and scope of the current case study.

Nevertheless, our study setup has proven successful in identifying relation among control-flow and performance interactions and can thus serve as a blueprint for further studies. It is a proof of existence, which does not prove that this relation is present in all kinds of systems, but which shall guide follow-up work to systematically study and exploit the relation. The future studies can rely on our conceptual framework for investigating relations among other types of internal and external interactions, for example, a relation among data-flow and performance interactions. Considering both data-flow and control-flow interactions for constructing a predictor for performance interactions will potentially result in higher precision and recall of the predictor.

6 Threats to Validity

Internal Validity

Due to technical limitations of TypeChef, we were unable to identify the exact number of indirect function calls between features (i.e., calls made using function pointers) and, consequently, the exact support values for the corresponding item sets (Sec. 4.4). We approximated these support values based on the distribution of the support values for the item sets calculated from direct function calls. Our approximation method may result in an inaccurate calculation of the precision and recall values of the frequent item set predictor. Nevertheless, we expect that improving the approximation would rather improve the precision and recall of the predictor.

Due to the static nature of the call-graph analysis employed by TypeChef, the collected information about the calls may be an approximation, and, as a consequence, a threat to internal validity. To mitigate this threat, we verified all control-flow interactions (identified using call-graph analysis) that are related to performance interactions by manually inspecting the source code of the subject systems and by confirming that these control-flow interactions actually exist.

External Validity

We have chosen a case study as our research method (Sec. 3.1), which suits well the exploratory nature of our study, which aims at the initial investigation of the relation between control-flow and performance feature interactions. The downside of using this research method is that it cannot be efficiently applied to multiple reasonably large configurable systems. In fact, it threatens the external validity of our study, since we focused on analyzing two systems and our results may not hold for other highly-configurable systems. Nevertheless, our study setup has proven successful and can thus serve as a blueprint for further studies that can rely on our conceptual framework for studying relations among external and internal interactions. We conjecture, that the relation between performance and control-flow interactions that we identified in our subject systems is likely to exist in systems with a larger number of features as well.

7 Related Work

In recent years, a number of papers aimed at detecting feature interactions in highly-configurable systems. We summarize and subdivide them according to our classification (Apel et al, 2013b) into those considering internal feature interactions and those considering external feature interactions. The fact that we were able to clearly assign related work to one of the feature interaction classes shows that previous studies focused on one interaction class at a time

and did not consider relations among different classes of feature interactions. The only exception is early work by Siegmund et al (2013b), who analyzed the nesting structure of preprocessor directives (i.e., internal interactions) to accurately predict the binary footprint of system configurations (external interaction). To our best knowledge, there is no other work that studied multiple types of interactions in combination and investigated their relation, as we do in our case study.

Internal Feature Interactions

Information on internal feature interactions is often used by techniques that aim at minimizing test-suite and test-effort for highly configurable systems. Reisner et al (2010), Nguyen et al (2016), and Tartler et al (2012) apply symbolic evaluation as well as dynamic and static program analysis to infer minimal sets of features responsible for a given code coverage. Kim et al (2011) apply use program analysis to identify features that do not interact with other features with respect to the test suite. Garvin and Cohen (2011) explore the connection between feature interactions and interaction faults. Lillack et al (2018) extend static taint analysis to automatically track load-time configuration options to the code locations where they influence the control flow, thereby identifying control-flow interactions induced by these configuration options. von Rhein et al (2018) applies variability-aware control-flow and data-flow analyses—which are a basis for detecting corresponding control-flow and data-flow interactions—to five real-world configurable systems. Meinicke et al (2016) developed a dynamic analysis based on variability-aware execution to identify control-flow and data-flow interactions, which potentially enables for more precise detection of interactions compared to a static analysis. They also conducted a controlled experiment showing the effectiveness of the technique for detecting internal interactions (Soares et al, 2018). Passos et al (2018) conducted an extensive study of feature scattering on the Linux kernel, which may serve as an indicator of potential internal interactions.

External Feature Interactions

There is a number of recently proposed performance prediction techniques for highly configurable systems: Guo et al (2013), Siegmund et al (2012), Sarkar et al (2015), Thereska et al (2010), Westermann et al (2012), Zhang et al (2015), and Nair et al (2018a) use machine-learning techniques, such as, CART, multivariate regression, Fourier and spectral learning for learning a performance function based on the performance measurements of a configuration sample. None of these approaches exploits information on internal feature interactions for performance prediction. Another direction of research focuses on improving sampling methods for finding optimal configurations: Nair et al (2018b) use a sequential model-based method to optimize search for optimal configurations. Nair et al (2017) use an ensemble of weak learners to rank configurations according to their performance. Kaltenecker et al (2019) propose

distance-based sampling as a new sampling strategy to find a representative set of configurations based on which performance of other configurations can be predicted. Guo et al (2018) combines CART, systematic re-sampling, and parameter tuning to learn accurate performance models from a small sample of measured configurations, without additional measurements for validation. All of these techniques learn performance (external) feature interactions as an integral part of the overall black-box learning process, that is, without considering the internal feature interactions.

8 Conclusion

In our case study we explored the relation among internal (control-flow) and external (performance) feature interactions that occur in highly configurable systems. Using the encryption library MBEDTLS and the database engine SQLITE as real-world subject systems, we identified control-flow and performance feature interactions using static program analysis and machine learning. Analyzing the interactions, we found that they can be related based on the involved features. By manually inspecting the code, we confirmed the causal relation between the interplay of features at the control-flow level and the identified performance interactions among the same features. Furthermore, based on the identified relation, we defined two predictors for performance feature interactions and conducted a preliminary evaluation of these predictors. The evaluation showed that the predictors have low precision and recall, presumably, because features also interact at the data-flow level. Future predictors based on the internal feature interactions should consider both control-flow and data-flow interactions to improve their predictive power.

Beside this negative result, using the identified relation among control-flow and performance feature interactions, we are still able to shrink the search space of performance feature interactions (by almost 300 times for MBEDTLS and by 4 times for SQLITE) without losing any of the performance feature interactions actually existing in our subject systems. Performance prediction techniques that rely on sampling can use our results to make their sampling more focused on configurations with potential performance interactions.

All in all, our study setup has proven successful and can thus serve as a blueprint for further studies that can rely on our conceptual framework for studying relations among external and internal interactions.

Avenues of Future Work

Extending TYPECHEF with variability-aware, inter-procedural data-flow analysis will empower it to automatically detect control-flow interaction among features that interact using indirect function calls (i.e., calls made using function pointers). This will fully automate the detection of control-flow interactions, making it straightforwardly applicable in practice. Another immediate consequence of this extension would be the ability to detect data-flow interactions,

which may induce performance interactions (as we discussed in Section 2). A future study shall then investigate the relation among data-flow and performance interactions relying on our conceptual framework.

Deriving new predictors for performance interactions and evaluating them is another avenue of future work. Assuming that we extended TypeChef with a fully-fledged, inter-procedural data-flow analysis (von Rhein et al, 2018), an obvious next step would be to enrich the predictor that we presented in this study with data-flow interactions. As we discussed in Section 5, considering data-flow interactions may substantially increase the predictor's precision and recall.

Acknowledgements Kolesnikov's, and Apel's work has been supported by the German Research Foundation (AP 206/5, AP 206/6, AP 206/7, AP 206/11) and by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) project No. 849928. Siegmund's work has been supported by the German Research Foundation under the contracts SI 2171/2 and SI 2171/3. Kästner's work has been supported in part by the National Science Foundation (awards 1318808, 1552944, and 1717022), the Science of Security Lablet (H9823014C0140), and AFRL and DARPA (FA8750-16-2-0042).

References

Apel S, Batory D, Kästner C, Saake G (2013a) Feature-Oriented Software Product Lines. Springer

Apel S, Kolesnikov S, Siegmund N, Kästner C, Garvin B (2013b) Exploring feature interactions in the wild: The new feature-interaction challenge. In: Proc. FOSD Workshop, ACM, pp 1–8

Borgelt C (2012) Frequent item set mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2(6):437–456

Bruns G (2005) Foundations for features. In: Feature Interactions in Telecommunications and Software Systems VIII, IOS Press, pp 3–11

Duong T, Rizzo J (2011) Here come the ⊕ ninjas. `https://web.archive.org/web/20150630133214/http://www.hpcc.ecs.soton.ac.uk/~dan/talks/bullrun/Beast.pdf`, Accessed: 2019-03-01

Ferreira G, Kästner C, Pfeffer J, Apel S (2015) Characterizing complexity of highly-configurable systems with variational call graphs: Analyzing configuration options interactions complexity in function calls. In: Proc. HotSoS, ACM, p 17:1–2

Flyvbjerg B (2006) Five misunderstandings about case-study research. Qualitative Inquiry 12(2):219–245

Garvin BJ, Cohen M (2011) Feature interaction faults revisited: An exploratory study. In: Proc. ISSRE, IEEE, pp 90–99

Guo J, Czarnecki K, Apel S, Siegmund N, Wasowski A (2013) Variability-aware performance prediction: A statistical learning approach. In: Proc. ASE, IEEE, pp 301–311

Guo J, Yang D, Siegmund N, Apel S, Sarkar A, Valov P, Czarnecki K, Wasowski A, Yu H (2018) Data-efficient performance learning for configurable systems. Empirical Software Engineering 23(3):1826–1867

Jaccard P (1912) The distribution of the flora in the alpine zone. New phytologist 11(2):37–50

Kaltenecker C, Grebhahn A, Siegmund N, Guo J, Apel S (2019) Distance-based sampling of software configuration spaces. In: Proc. ICSE, ACM, to appear

Kang K, Cohen S, Hess J, Novak W, Peterson A (1990) Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University

Kästner C, Apel S, ur Rahman SS, Rosenmüller M, Batory D, Saake G (2009) On the impact of the optional feature problem: Analysis and case studies. In: Proc. SPLC, pp 181–190

Kim C, Batory D, Khurshid S (2011) Reducing combinatorics in testing product lines. In: Proc. AOSD, ACM, pp 57–68

Kolesnikov S, Siegmund N, Kästner C, Grebhahn A, Apel S (2018) Tradeoffs in modeling performance of highly configurable software systems. Software and Systems Modeling (SoSyM) pp 1–19, online first

Lillack M, Kästner C, Bodden E (2018) Tracking load-time configuration options. IEEE Transactions on Software Engineering (TSE) 44(12):1269–1291

Maqbool O, Babri H (2007) Hierarchical clustering for software architecture recovery. IEEE Transactions on Software Engineering 33(11):759–780

Medeiros F, Kästner C, Ribeiro M, Gheyi R, Apel S (2016) A comparison of 10 sampling algorithms for configurable systems. In: Proc. ICSE, ACM, pp 643–654

Meinicke J, Wong C, Kästner C, Thüm T, Saake G (2016) On essential configuration complexity: Measuring interactions in highly-configurable systems. In: Proc. ASE, ACM Press, pp 483–494

Nair V, Menzies T, Siegmund N, Apel S (2017) Using bad learners to find good configurations. In: Proc. ESEC/FSE, pp 257–267

Nair V, Menzies T, Siegmund N, Apel S (2018a) Faster discovery of faster system configurations with spectral learning. Automated Software Engineering 25(2):247–277

Nair V, Yu Z, Menzies T, Siegmund N, Apel S (2018b) Finding faster configurations using FLASH. IEEE Transactions on Software Engineering (TSE) pp 1–1, DOI 10.1109/TSE.2018.2870895, online first

Nguyen T, Koc U, Cheng J, Foster JS, Porter A (2016) iGen: Dynamic interaction inference for configurable software. In: Proc. FSE, ACM, pp 655–665

Passos L, Queiroz R, Mukelabai M, Berger T, Apel S, Czarnecki K, Padilla J (2018) A study of feature scattering in the Linux kernel. IEEE Transactions on Software Engineering (TSE) Online first

Qiao Y, He J, Yang Y, Ji L (2013) Analyzing malware by abstracting the frequent itemsets in API call sequences. In: Proc. TrustCom, IEEE, pp 265–270

Reisner E, Song C, Ma K, Foster JS, Porter A (2010) Using symbolic evaluation to understand behavior in configurable software systems. In: Proc. ICSE, ACM, pp 445–454

von Rhein A, Grebhahn A, Apel S, Siegmund N, Beyer D, Berger T (2015) Presence-condition simplification in highly configurable systems. In: Proc. ICSE, IEEE, vol 1, pp 178–188

von Rhein A, Liebig J, Janker A, Kästner C, Apel S (2018) Variability-aware static analysis at scale: An empirical study. ACM Transactions on Software Engineering and Methodology (TOSEM) 27(4):18:1–18:33

Sarkar A, Guo J, Siegmund N, Apel S, Czarnecki K (2015) Cost-efficient sampling for performance prediction of configurable systems. In: Proc. ASE, IEEE, pp 342–352

Shull F, Singer J, Sjøberg D (2007) Guide to Advanced Empirical Software Engineering. Springer

Siegmund N, Kolesnikov S, Kästner C, Apel S, Batory D, Rosenmüller M, Saake G (2012) Predicting performance via automated feature-interaction detection. In: Proc. ICSE, IEEE, pp 167–177

Siegmund N, von Rhein A, Apel S (2013a) Family-based performance measurement. In: Proc. GPCE, ACM, pp 95–104

Siegmund N, Rosenmüller M, Kästner C, Giarrusso P, Apel S, Kolesnikov S (2013b) Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Information & Software Technology 55(3):491–507

Siegmund N, Grebhahn A, Apel S, Kästner C (2015) Performance-influence models for highly configurable systems. In: Proc. ESEC/FSE, ACM, pp 284–294

Soares LR, Meinicke J, Nadi S, Kästner C, de Almeida ES (2018) Exploring feature interactions without specifications: A controlled experiment. In: Proc. GPCE, ACM Press, pp 41–52

Tartler R, Lohmann D, Dietrich C, Egger C, Sincero J (2012) Configuration coverage in the analysis of large-scale system software. SIGOPS Operating Systems Review (ACM OSR) 45(3):10–14

Thereska E, Doebel B, Zheng A, Nobel P (2010) Practical performance models for complex, popular applications. SIGMETRICS Performance Evaluation Review 38(1):1–12

Westermann D, Happe J, Krebs R, Farahbod R (2012) Automated inference of goal-oriented performance prediction functions. In: Proc. ASE, ACM, pp 190–199

Yin R (2003) Case Study Research–Design and Methods. Sage

Zave P (2009) Modularity in distributed feature composition. Software Requirements and Design: The Work of Michael Jackson pp 267–290

Zhang Y, Guo J, Blais E, Czarnecki K (2015) Performance prediction of configurable software systems by Fourier learning. In: Proc. ASE, IEEE, pp 365–373