UNIVERSITÄT PASSAU

Detecting Control-Flow and Performance Interactions in Highly-Configurable Systems

A Case Study

Master's Thesis

Department of Informatics and Mathematics

Chair of Software Engineering

Author:	Alexander Denk
1st Corrector:	Prof. Dr. Sven Apel
2nd Corrector:	Prof. Christian Lengauer, Ph.D.
Advisor:	Sergiy Kolesnikov, M.Sc.
Date:	31.03.2017

Detecting Control-Flow and Performance Interactions in Highly-Configurable Systems
A Case Study

Abstract

In the context of highly configurable software system we use the term feature for a configuration option that configures, enables or disables a certain functionality of the system. A selection of features that is used to build or run a system we call variant. Feature interactions characterize the influence of the presence of one feature on another feature. Acquiring knowledge about feature interactions creates the ability to make predictions on the performance behaviour of software variants based on the set of features that is selected for the variant.

Internal feature interactions, such as control-flow, and data-flow interactions, are relatively easy to find using available static-analysis techniques. On the contrary, external feature interactions, such as performance interactions, are hard to find because the corresponding supervised machine learning techniques require expensive benchmarks and good sampling techniques. This process can become computationally infeasible very fast with an increasing number of features being present. Future work is set to investigate relations among internal and external feature interactions and how these relations can be used to predict external interactions based on the internal ones. This Thesis is the base for the first step towards performance predictions based on internal feature interactions.

For these purposes we selected two real-world highly configurable systems from different domains. We built feature models and a benchmarking framework and collected a large set of measurements of non-functional properties, such as compilation times, execution times, binary and main-memory footprints and energy consumption. Using the static analysis tool TypeChef, we prepared feature models and the source code in order to build a variability-aware call graph that can be utilized to detect control flow feature interactions. We used the data of the performance analysis and prediction tool SPLConqueror to identify performance feature interactions.

Finally, we analysed the collected data sets and removed unreliable measurements to provide a solid and reliable platform for further work.

Contents

1 Introduction

In this chapter we introduce the basic notions of highly configurable systems and give an introduction into feature interactions. We state the goals of our work and we give an overview of the structure of the thesis.

1.1 Highly Configurable Systems

1.1.1 General Terms

Although we found the term widely used among different publications in computer science most of them do neither provide a formal nor an informal definition of *highly configurable (software) systems*. Cohen, Dwyer and Shi used the following description in their paper "Interaction testing of highly-configurable systems in the presence of constraints":

> The concept of a highly-configurable software system arises in many different settings differentiated by the point in the development process when feature binding occurs, i.e., the binding time [1].

This description is very general as it does not limit the type of the software architecture, the times of the bindings or any other characteristic. The core of the description is that a highly configurable software system has many different settings. For our work this description can be used as definition of a highly configurable system, but we will add more restrictions to it in the following chapters.

Highly configurable systems have a great value to their customers, as they can be configured for their needs. A user interface for example only display the functions that are relevant for the use case of the customer, or a security critical software can be reduced to the essential parts in order to mitigate the attack surface to components that are really required.

In older publications the term product line was often used synonymous to highly configurable system. Newer publications distinguish between product lines and highly configurable system because not every highly configurable system is designed as real product line, which uses dedicated and feature-orientated approach for their implementation.

In the context of highly configurable software system we use the term *feature* for a configuration option that configures, enables or disables a certain functionality of the system. A selection of features that is used to build or run a system we call *variant*.

"Modeling variability is a crucial step in product-line development. [...] A common approach is to express variability in terms of common and optional features, a process called appropriately enough feature modeling. We use feature models and their graphical representation as feature diagrams, because they are currently the most popular form of variability models. [...] A *feature model* documents the features of a product line and their relationships" [2].

"A *presence condition* is an expression over a set of configuration options. The condition represents a subset of configurations in which a certain implementation artifact, such as a code fragment, is included in the corresponding system variants or in which a certain behavior can be observed" [3].

"Besides functional requirements, different application scenarios raise the need for optimizing non-functional properties of a variant. The diversity of application scenarios leads to heterogeneous optimization goals with respect to *non-functional properties* (e.g., performance vs. footprint vs. energy optimized variants). Hence, an SPL has to satisfy different and sometimes contradicting requirements regarding non-functional properties. Usually, the actually required non-functional properties are not known before product derivation and can vary for each application scenario and customer" [4].

1.1.2 Types of Variability and Implementation Concepts

The description of highly-configurable software system leaves the architectural style of the implementation completely open. Cohen, Dwyer and Shi give software product lines (SPL) as "an example of very early feature binding" [1]. "At the other end of the spectrum" they see "dynamically reconfigurable systems, where feature binding happens at runtime and may, in fact, happen repeatedly" [1].

For our work we do not use real software product lines, as they are related to a feature orientated programming approach, instead we focus on highly-configurable systems implemented using pre-processor directives of the C programming language (*#ifdef*-variability).

In this software projects we distinguish between two different types of variability:

Compile time variability: As compile time variability we understand configuration options, that have to be configured ahead of the compilation process and which will affect the resulting output of the compiler. The configuration options cannot be changed after the compilation happened any more for the resulting variant. For example *#ifdef*-variability using C pre-processor directives are evaluated by the C pre-processor. Listing 1 shows an example, where the feature for logging is only enabled, if the pre-processor macro *FEATUR_LOGGING* has been defined. The feature needs not be implemented in a single place, there can be many different places in the source code where parts of the feature are defined. Code which does not fulfil the condition of the *#ifdef*-expression will be removed by the C pre-processor and will not be passed to the C compiler.

Runtime variability: As runtime variability we understand all configuration options that do not depend on the compiled variant, but we have the prerequisite that the configuration option is part of the software variant. For example, if a variant of a program supports the adjustment of the cache size as configuration option, one kind of runtime variability

> is to configure the size statically before the program start-up in a configuration file, so called load-time variability. An alternative kind of runtime variability is to configure the option dynamically while the program is executed.

```c
#ifdef FEATURE_LOGGING
    /* Logging Code... */
#endif
```

Listing 1: Example for a feature that can be enabled using C pre-processor directives.

The two types of variability neither exclude nor require each other. A highly configurable system can only use compile time variability, only use runtime variability or it can use both in combination.

A common way in real-world systems written in C is to use compile time variability as described above. This causes a lot of problems in the context of software verification. A type error for example can be produced easily as shown in listing 2. The variable x is defined as *int* in the context of *FEATURE_A* and as *pointer on float* in the context of *FEATURE_B*. Depending on the usage of x the program may cause unpredictable behaviour or crashes in certain configurations. In the worst case the number of variants is exponentially to the number of features. This makes it computationally infeasible to test and benchmark all possible variants using standard techniques to get information about validity and performance.

```c
#ifdef FEATURE_A
    int x = 3;
#endif
#ifdef FEATURE_B
    float* x;
#endif
```

Listing 2: Example for a possible type error using C pre-processor directives.

We use the tool TypeChef[1] which is supporting the analysis of *#ifdef*-variability. In the following work we will focus on compile time variability using *#ifdef*-expressions, but we have to be aware that the runtime configuration of a variant may affect our work.

1.2 Types of Feature Interactions

In this section we give a working definition of internal and external feature interactions and provide an overview of the different observable or analysable occurrences of each type. Apel et al. define internal and external interactions as follows:

> The visibility of a feature interaction denotes the context in which a feature interaction appears. Feature interactions may appear at the level of the externally

[1] https://github.com/ckaestne/TypeChef

> observable behavior of a program, including functional behavior (e.g., segmentation faults and all kinds of other bugs) and non-functional behavior (e.g., performance anomalies and memory leaks). Feature interactions may also appear internally in a system, at the level of code that gives rise to an interaction or at the level of control and data flow of a system (e.g., data-flows that occur only when two or more features are present). We believe that there may be systematic correlations between externally-visible and internally-visible interactions, which is a major motivation for our endeavor to explore and understand the nature of feature interactions [5].

In our focus are non-functional properties that can be measured in order to detect external feature interactions that are affecting the performance in any way, but do not threaten the validity of the program itself. For example we try to exclude variants of a program that produce segmentation faults already within the feature model as they are not part of our research. Non-functional properties that are in the point of interest include: Time to compile, memory consumption, energy consumption, runtime performance, binary footprints. We will discuss what interactions we try to measure and what data we will use for further processing later on.

For internal feature interactions we will give a more detailed insight in advanced analysis techniques in the chapter 2 “Related Work”. The discussion which of the available analysis method we will use for our work is in chapter 5 “Measurements of Internal Feature Interactions”. As the extend of our work is limited, we will select one analysis method.

1.3 Motivation, Workflow and Goal of the Thesis

In this section we will explain the motivation behind the Thesis, define the goals and the research questions of our work.

1.3.1 Motivation

One of the topics in research of highly configurable systems is the prediction of the performance behaviour of certain variants. One would like to know the expected performance of a variant before it is built and rolled out without executing measurements and running complex benchmark scenarios for the variant. Holding benchmarked values available for every possible variant is impossible because there may be exponentially many, in the number of features, variants.

The current state of the art are supervised machine learning techniques [6]. As input for the algorithms a large number of performance measurements is required to get reliable results. Providing these measurements get computationally infeasible fast with a growing number of features and therefore growing number of possible variants.

By providing information about feature interactions for the machine learning system we try to reduce the number of required measurements to a computationally feasible amount. For the analysis of internal feature interactions static analysis tools are available that are working in almost linear time depending on the source code size. To provide useful internal feature interactions the overall goal of the chair's work is to find relations between the two data sets, or to disprove this assumption.

This thesis should be the the base for the first step towards finding these relations. For this we provide feature models and variability-aware call graph data as well as measurements of non-functional properties for the same subject systems.

1.3.2 Workflow

The first step of our work is to conduct a preliminary study. In this study we want to identify real-world software domains that are applicable for highly configurable software and performance benchmarking. After identifying the domains we want to find candidate subject systems fitting in these domains. To increase the external validity of the case study we aim to pick subject systems from different domains. At least we want to cover two domains with two subject systems.

As second step we want to explore methods to reverse-engineer for highly configurable systems and to find a feature model representation format which can be used for TypeChef and SPLConqueror as input model.

The third step is to setup an infrastructure for measurements of internal and external feature interactions. As far as achievable, the infrastructure should be usable for all subject systems.

The last step is to prepare two subject systems for the measurements and to use the infrastructure that we set up to execute the measurements. The validity and quality of the results should be discussed.

1.3.3 Goal

The main goal of the thesis is to execute the four steps described in section 1.3.2 in order to get feature models, prepare these models and the source code of our subject systems for the analysis of variability-aware control flow graph data and reliable measurement data of non-functional properties, especially focused on performance, for the same subject systems to provide a solid and reliable base for further analysis of internal and external feature interaction and relations between them.

1.3.4 Limitation

Our work aims for creating the prerequisites for the detection and further analysis of the feature interactions. We prepare the TypeChef control flow analysis and test that the detection of internal feature interactions works, but it was not part of our work to process the results further. We used SPLConqueror to test that our measurements are suitable for the detection of external feature interactions, but we did not analyse the the findings of SPLConqueror.

1.4 Case Studies

In the domain of highly configurable systems and software product lines case studies have frequently been used to explore the relatively new field in research. Kastner and Apel used for example a case study for converting the C code base of BerleyDB into a AspectJ project as proof of concept for a real world application of AspectJ. The lessons learned, including that case studies are an appropriate method for exploring new research fields, have been published in the paper "A Case Study Implementing Features Using AspectJ" [7]. Kolesnikov and Roth "On the Relation Between Internal and External Feature Interactions in Feature-oriented Product Lines: A Case Study" [8]. Another example for a case study is "Feature-oriented Language Families: A Case Study" of Liebig, Daniel and Apel which proposes "language families, a feature-oriented approach to language engineering inspired by product lines and program families" with the goal to "systematically manage the development and evolution of variants and versions of a software language in terms of the language features it provides" [9].

Nevertheless are case studies are a controversial topic in research. There are several critical responses on case studies in general, where the critics mostly aim not at the method itself but at the bad understanding of the method. For example Gerring criticises in his paper "What Is a Case Study and What Is It Good for?" that case studies are widely used in political science, but the conductors of the studies do not often understand the method correctly. He criticises furthermore the vague definition of the methodology case study at all [10]. This problem is not limited to political science, it can be seen as general problem in science. Bennett also sees that the researcher have to be aware of how case studies can be used to avoid misleading outcomings: "On the methodological level, however, what is useful or necessary for one method, such as random selection of cases in a statistical study, may be unnecessary or even counterproductive in another, such as case studies. This creates an obligation for understand their respective strengths and limitations[11]". The journal article of Flyvbjerg "Five misunderstandings about case-study research" tries to overcome concerns on case studies as they are often misunderstood [12].

Flyvbjerg sees for example the following three misunderstandings and tries to clarify them:

- General, theoretical (context-independent) knowledge is more valuable than concrete, practical (context-dependent) knowledge.

- One cannot generalize on the basis of an individual case; therefore, the case study cannot contribute to scientific development.
- The case study contains a bias toward verification, that is, a tendency to confirm the researcher's preconceived notions [12].

For our work we think that a case study is an appropriate research method, as we are aware of the concerns against case studies and the misunderstandings. Contrary, we see the benefits for our research goal. Case studies are good for pilot-experiments, early project states and to explore new theories that can later be refined and validated or rejected with quantitative methods. This matches our work as this thesis should provide first data to explore the hypothesis that we can find relations between internal and external feature interactions, that are usable for performance predictions.

1.5 Structure of the Thesis

In the first chapter, we provide an introduction to highly configurable systems and to the two different types of feature interactions in these systems. We show why it is important for our research to find relations between these types. Furthermore, we define the goals of the thesis and the research questions and explain why we have chosen a case study as our research method.

The second chapter gives an overview of the related work in this field of research.

We describe the preliminary study for selecting appropriate domains, the general requirements for subjects systems and how we found subject systems for these domains in the third chapter. We show the results of the evaluation and discuss the decision for mbedTLS and SQLite as the systems we selected for our case study.

In the fourth chapter we describe the process we used to generate feature models from the documentation and the source code the subject systems. We introduce a meta model format to keep the different feature model formats of TypeChef and SPLConqueror in synchronization.

The fifth chapter contains an overview of the process of detection of internal feature interactions.

We focus on the main work of this thesis in the sixth chapter. The chapter describes the existing types of external feature interactions and which of them we are capable to measure. We create a generic system setup that is capable of executing the measurements and describe the hardware we used for the measurements. Additionally, we identify the general independent and observed variables and discuss sources for measurement bias and show how we handled them.

mbedTLS is presented as first subjects system of the case study in the seventh chapter. We give an introduction to the architecture of the software, the role of cipher suites and the problems resulting from having compile time and runtime configuration options combined in one software. Next, we discuss the feature model we use for our measurements. Shortly, we show the results on the internal feature interactions. We cover the setup and execution of the performance benchmark and therefore present the results and give an interpretation.

The SQLite chapter follows a similar structure as the mbedTLS chapter. After an introduction to SQLite we discuss the architecture that led to the feature model we use for our measurements, followed by the presentation of the results of the analysis of the internal feature interactions. We describe the creation of a performance benchmark for SQLite and the execution of the benchmark in detail. We present the data of the measurements and give an interpretation.

The chapter *Validity and Threats to Validity* discusses the internal and external validity, as well as threats to both, of every part of the case study.

In the last chapter we summarize the work and its results and give an outline to future work.

2 Related Work

We discuss in the following chapter related work. In the last years multiple publications have been released that try to detect feature interactions in highly configurable systems. We present work that aims to detect these interactions, as well as we present work, that is related to the workflow we used for reverse-engineering feature models, measuring the data of non-functional properties and for creating the control flow graphs.

We do not know work that aimed for both, detecting internal and external feature interactions for the same subject system.

2.1 Feature Model Generation

She and Lotufo present in their work "Reverse Engineering Feature Models" procedures for reverse engineering feature models based on a crucial heuristic for identifying parents, where thy see the major challenge of their work. They tried to automatically recover constructs such as feature groups, mandatory features, and implies/excludes edges and evaluated their method on thee subject systems, such as the Linux kernel [13].

2.2 Highly Configurable Subject Systems

Highly configurable systems have been investigated across different publications in the context of feature interactions. SQLite for example has been discussed as subject system for internal feature interactions, as well as for external feature interactions. Liebig used SQLite in his dissertation "Analysis and Transformation of Configurable Systems" for TypeChef analysis, while Siegmund et. al. used SQLite for "Predicting Performance via Automated Feature-interaction Detection" by using SPLConqueror [14, 15]. Both works where independent from each other and they did not use the same version of SQLite nor the same feature model.

Janker used mbedTLS as subject system for evaluating his method of inter-procedural variability-aware data flow analysis by using a combination of TypeChef and SPL$^{\text{LIFT}}$ [16]. His work was not finished when we finished working on detecting internal feature interactions in mbedTLS. Therefore, we focused on detecting variability-aware control flow interactions.

2.3 Detecting Internal Interactions

Janker's work on mbedTLS was also related in the context of detecting internal feature interactions, as he used a variability-aware analysis of data flows. For the same reason as in 2.2 we could not use his analysis for our work.

Other work on detecting internal feature interactions involved trying to minimize the test-effort for highly configurable systems, published by Nguyen et al. ("iGen: dynamic interaction inference for configurable software"), Reisner et al. ("Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems") and Tartler et al. ("Configuration Coverage in the Analysis of Large-scale System Software") [17, 18, 19]. Garvin et al. examine constitutions of faults at the code level and their connection to feature interaction faults in their work "Feature Interaction Faults Revisited: An Exploratory Study" [20].

2.4 Detecting External Feature Interactions

The state of the art of detecting external feature interactions builds on machine learning techniques to predict performance based on samples of measurements. All of them use black box approaches without using knowledge of the internal structure of interactions, such as control flows or data flows. As various machine learning techniques the use for example multivariate regression and Fourier learning. The work on this topic this was published by Sarkar et al. ("Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)"), Guo et al. ("Variability-aware performance prediction: A statistical learning approach"), Westermann et al. ("Automated Inference of Goal-oriented Performance Prediction Functions"), Zhang et al. ("Performance Prediction of Configurable Software Systems by Fourier Learning (T)"), and Siegmund et al. ("Predicting Performance via Automated Feature-interaction Detection"), [21, 22, 23, 24, 15].

3 Preliminary Study

For a case study, the selection of subject systems is a critical point as not every subject system is representative for other systems. Our main goal is to select a system that exhibits the phenomena we want to study. A bad selection can implicate the false rejection or false reassurance of a hypothesis. This chapter presents the setup and the results of the preliminary study we have conducted to find appropriate subject systems.

3.1 Selection of Software Domains

The internal and external validity of the case study should be maximized wherever possible. Therefore we need real-world subject systems that are not of the same application domain. On the other hand, we need subject systems where we are able to measure the performance. Software domains that are generally unsuitable for this, such as programs that focus on the graphical user interface or command line tool collections, have been excluded prior to.

We grouped the candidates for subject systems in the following domains[2], based on the experience from previous studies, but left open to extend the list when we find suitable subject systems that cannot be categorized in one of these domains [25, 15, 26]:

Domain	Examples
Databases	*PostgreSQL*, *MySQL*
Embedded Databases	*BerkleyDB*, *SQLite*
In-Memory Databases	*WhiteDB*
Webservers	*Apache HTTP Server*, *nginx*, *Lighttpd*
TLS-Libraries	*OpenSSL*, *axTLS*, *PolarSSL*, *LibreSSL*, *mbedTLS*
VPN-Libraries	*OpenVPN*
Encoding-Libraries	*libJPEG*, *mozjpeg*, *libPNG*, *libXSLT*, *libXML x264 Codec*
Compression-Libraries	*7-Zip*, *gnu gzip*
Interpreter	*Zend PHP*
Compiler	*GNU Compiler Collection (GCC)*, *Tiny C Compiler (TCC)*

Listing 3: Example domains we used to group candidates for subject systems.

Benchmarking client-server scenarios, for example, usually requires much more hardware resources for benchmark execution compared to single application benchmarks. Domains such as webservers have been given a lower priority to avoid time consuming benchmark setups in a single domain that are blocking subject systems from other domains.

[2]The examples are not restricted to software using *#ifdef*-variability

3.2 Requirements for Subject Systems

The general requirement is that we need to be allowed to use the software system for research purposes. We only use software that is available under open source or comparable licence models and we ignore proprietary software with publicly available source code because we cannot afford to analyse if our research may violate the concrete license terms.

Another requirement is that we only take highly configurable systems into consideration. As highly configurable we decided to require a minimum of ten configuration options. With no other restrictions in the feature models this would allow in theory 1,024 possible variants. For the statistical evaluation of the data we do not want a lower number of variants.

For static analysis we will use the tool TypeChef which is capable of analysing code written in the C programming language [27]. Moreover, TypeChef focuses on analyzing variability of configuration options that are realized using C pre-processor directives (*#ifdef*-variability). Consequently, we are limited to analyse software that is written in C and is using this type of configuration options. We have the ability to integrate TypeChef into a make build process chain and we can work with file lists. However, Janker found build systems in the context of his Master's Thesis that need to be aware of the variability in the build process [16].

The other tool we use, SPLConqueror, does not have requirements on the subject systems programming language and source code structure because it does not use the source code. SPLConqueror takes feature model information and measurement data as input, for example the binary file sizes of the programs or performance measurements. Our work is focused on performance and for that we need the ability to provide measurements for SPLConqueror. To get this measurements we are bound to software that can be benchmarked using existing benchmarks targeting performance, or the use case and domain of the software should allow the creation of a performance benchmark in a feasible amount of time.

3.3 Setup and Execution of the Evaluation

The terms highly configurable software and *#ifdef*-variability are not widely used under software developers which makes it hard to identify potential subject systems. Even if a software uses this variability model, it mostly does not provide a formal feature model. We found different projects documenting the configuration options in completely different ways and different granularity.

In the first stage of our preliminary study we collected a list of software projects that are written in C and that use an open source license. We used the software lists available at Wikipedia[3] for this and used additional sources covering specific application domains like NoSQL databases. Additionally, we considered all GNU and GNOME tools to be able to

[3] `https://en.wikipedia.org/wiki/Category:Lists_of_software`

provide candidates as well as software that has been used in other research projects at the chair. All in all, we listed 252 software projects. We removed all software that is definitely not written in C or did not satisfy our license requirement. The remaining list contains 117 projects.

For these projects we needed to check if they are using *#ifdef*-variability and if this variability can be freely configured or is only used as platform abstraction layer. Listing 4 shows such an example where code targets a special platform but cannot be freely configured on this platform.

```
#ifdef WIN32
    #ifdef HAVE_INT8
        /* Win32 Code for 8-bit integer. */
    #endif
    #ifdef HAVE_INT16
        /* Win32 Code for 16-bit integer. */
    #endif
#endif
```

Listing 4: Example for an *#ifdef*-expressions that is used for platform abstraction.

Contrary to that listing 5 shows a feature *PREFIX_ALGORITHM* that has a common prefix *PREFIX_*, this is what we have often seen as an indicator for being a configuration option that can be changed for the demands of the use case of the software. The naming conventions differ from project to project, so we cannot generalize if a macro is used as feature, for platform abstraction or as code macro.

```
#ifdef PREFIX_ALGORITHM
    #ifdef PREFIX_LOGGING
        /* Logging code for feature PREFIX_ALGORITHM */
    #endif
    /* Code of feature PREFIX_ALGORITHM. */
#endif
```

Listing 5: Example for an *#ifdef*-expressions that are likely to represent a feature.

To support our filter process we wrote a tool that processes C code without the need of a compiler or lexer. We decided to use this approach to avoid parsing and compiling errors caused by C++ code and non standard C dialects, which where often used only for small code parts. This is working as heuristic and will not cover all edge cases, but for the masses of source code we processed it worked well. The tool collected statistics over lines of source code, lines of comments and number of macros. Additionally, it collected all macros and the context the macros are used and provided a grouping on the longest common prefixes of the macros.

We downloaded the source codes of all candidates of the list and used the tool for an analysis. Software that was not using macros or using macros in the context of platform abstraction and to replace parts of the C code was excluded immediately. For the remaining projects we read the documentation, ensured the project is written in C and does not contain C++ code that can be excluded from the source. Moreover, we tried to put the macros in the context of a feature model.

Some projects turned out to be unmanageable because of the hardly understandable project and configuration option structure, for example GCC. Making useful changes in the configuration options require in-depth knowledge of the project. We dropped these projects from our list.

For the remaining projects we tried to find benchmarks that are publicly available or alternatively investigated if it is possible to create a benchmark. We estimated the effort to create benchmarks.

3.4 Evaluation Results

A result of the described evaluation progress we extracted a list of 32 projects, categorized in domains and containing information about the feasibility of benchmarks, code metrics and the estimated number of features. It would be infeasible to determine the actual number of

Name	Type	K LoC	Features	Benchmark / Method
PolarSSL / mbedTLS	SSL/TLS Library	45	100+	Throughput of Connection
axTLS	SSL/TLS Library	16	10+	Throughput of Connection
OpenSSL	SSL/TLS Library	250	30+	Throughput of Connection
LibreSSL	SSL/TLS Library	170	30+	Throughput of Connection
PostgreSQL	Database	710	30+	TPC-Suite
SQLite	Embedded-Database	90	100+	SQL-Query Performance
BerkeleyDB	Embedded-Database	520	30+	SQL-Query Performance
WhiteDB	In-Memory Database	22	<10	Yahoo Cloud Serving
libJPEG	Encoding-Library	17	20+	Encoding Time
libPNG	Encoding-Library	44	30+	Encoding Time
libXML2	Encoding-Library	215	20+	Encoding Time
libXSLT2	Encoding-Library	30	10+	Encoding Time
PHP	Interpreter	900	30+	Script Execution Time
Apache Webserver	Webserver	140	30+	Requests per Second
lighttpd	Webserver	35	20+	Requests per Second
nginx	Webserver	100	30+	Requests per Second

Table 3.1: 16 candidates for subject systems grouped by domain.

features, because it requires to check the definition and the corresponding documentation of thousands of macros manually.

Table 3.1 shows an extract of this list containing the favoured 16 subject systems with the domain they belong to, the estimated count of the lines of code as well as the lower bound of the number of features. The last column states a benchmark strategy or if available a concrete benchmark. We grouped the entries by domain and ordered the domains by our estimated benchmark with the lowest effort being on top. We decided for mbedTLS as our first subject system. mbedTLS offers a very extensive architecture and configuration option documentation and allows the construction of a structured feature model. Benchmarks are achievable by rewriting the integrated compatibility test server and client programs into performance benchmarks. The project does not require additional libraries and is written in ANSI C. When we started mbedTLS has not been used in this research field, so we had to begin from scratch.

As second subject system we chose SQLite as it offers a well documented and clear configuration model and allows benchmarking by using SQL-queries. Moreover, SQLite has the advantage to be combinable into a single source code file, is independent of vendor libraries and written in ANSI C. SQLitewas used for previous research topics in the context of highly configurable systems, for example in Liebig's work that is presented in the paper "Morpheus: Variability-aware Refactoring in the Wild" [28].

4 Generation of Feature Models

In this chapter we discuss the requirements for feature models of the subject systems, we introduce a meta model format to generate feature models for TypeChef and SPLConqueror from the same source model and describe how we reverse-engineered the models from the source code and the documentation.

4.1 Requirements for Feature Models

For our purpose to compare internal and external feature interactions we have different requirements for feature models that have to be fulfilled.

In our preliminary study we came across many different systems using *#ifdef*-variability, but most of them are not intended to be designed like a software product line. For some cases the variability was not even documented outside of the source code. Consequently, none of the candidates for subject systems had a formal feature model available. For some systems, such as BerkeleyDB, a feature model has been constructed in other research projects. BerkeleyDB was refactored to FAME-DBMS and therefore a feature model was needed [29]. However, these are not models that have been created by the developers of the systems, either, and have not been the concept for the implementation.

As a consequence, we have to design feature models for our subject systems from scratch, or we have to use previous work from other authors. This gives us on the one hand the ability to design the model for our needs, but is on the other hand a threat to validity because the model can be designed too specific and not represent the software at all.

On the one hand, we have the demand to represent the real-world system as precisely as possible to increase the internal validity of the case study. Therefore, we want to construct feature models containing all features that can be configured and model all constraints between this features.

On the other hand, we need models that are usable for benchmarking. For example the modularity concept of the subject system mbedTLS allows single modules, such as the cipher or hash module, to be used standalone. The benchmark for mbedTLS measures the throughput of connections and for that it needs all modules that are required for a connection to be set up. We cannot avoid to add these restrictions to the feature model, even the original intention of the software allows more flexibility at this point.

Moreover, we want to exclude variants that produce feature interactions in the context of unexpected failures. If a variant, for example, leads to a type error, we exclude the variant from the model as the variant has no value for our later research task. TypeChefś variability-aware analysis for type errors can be used to identify such variants.

Another requirement we need to fulfil is that the model's number of valid variants it can produce is feasible. This requires us to reduce the number of features as long as we get a number of variants we can handle. This number depends on the estimated benchmark runtimes and must be estimated for each subject system. Removing features must keep the model valid.

The last requirement we have to respect is that we want to ensure that features being present are actually called while running the benchmark. For example, if a variant of mbedTLS contains two ciphers and only one can be benchmarked, the presence of the second cipher has no value for our measurements. However, there may be an influence of the presence of the unused code we discuss in section 6.2. We decided to restrict the model to variants where we can be sure that a feature is used if it is present.

4.2 Representation Formats of Feature Models

This section describes the common methods to represent feature models and introduces a meta format that we used to generate these models.

4.2.1 Representation Methods

There are different methods available to represent feature models. We give an introduction to the common different methods in the following:

Visual representations: A common way to represent feature models is to visualize them in diagrams using different notations. We can compare this method to drawing UML diagrams. There are multiple notation techniques that can be grouped into basic feature models, cardinality-based feature models and extended feature models. In the following we stick to the notation used by Thüm and Kästner which is supported by FeatureIDE [30].

SPLConqueror XML format: SPLConqueror defines an XML schema for feature models. The schema supports the common elements of feature models, such as mandatory and optional features, parent-child relations, alternative groups, inclusion and exclusion constraints and offers the ability to define additional boolean constraints in a conjunctive normal form. XML is used to load models into SPLConqueror.

FeatureIDE XML format: Like SPLConqueror defines FeatureIDE an XML schema for feature models. Both schemas are different. One main use case for the schema is the persistent storage of visual feature models.

Feature Expression format: The Feature Expression format defines the feature model in a conjunctive normal form of boolean terms. A term, for example, looks like $defined(A)$ $=> defined(B)$, which states that the feature A implicates the presence of feature B. A line in the text-file contains one term, a comment or a blank line. The text based representation is human readable. TYPECHEF requires all features that are used in terms, as well as features that are completely optional and have no relations to other features and are therefore not used in terms, to be declared. Usually this is done in a file named *open-features* which contains a newline separated list of all feature names.

The list covers representations we needed for our work. There are other common representation techniques for feature models available, such as the DIMACS standard format.

4.2.2 A Meta-Model for Generation of Feature Expression and SPLConqueror Models

In our case study we have, all in all, three software components demanding feature models as inputs: TYPECHEF, SPLCONQUEROR and FEATUREIDE.

TYPECHEF accepts a CNF DIMACS feature model or a FEATURE EXPRESSION model as input format, while SPLCONQUEROR only accepts its own XML format as input. For TYPECHEF we prefer using the human-readable FEATURE EXPRESSION model to avoid the need for additional tool support. Neither TYPECHEF nor SPLCONQUERORoffer a good visual representation of the input models. SPLCONQUEROR's graphical user interface is capable of rendering the feature model by using a standard Windows Forms tree component and different colours. However, this only gives a very weak impression of the model, because SPLCONQUEROR cannot render the model very well, for example without scrolling and by using common visual elements for feature model representation. There is not any other known application that offers a visual representation for one of the three models. As a result, we used FEATUREIDE as additional application for creating the visual representations. FEATUREIDE uses XML to store the model but the XML schema definition is not compatible with SPLCONQUEROR's XML schema definition.

Overall, there is a minimum of three separate representation formats that have to be used for our case study. The need for three different formats makes it problematic to keep all three models consistent as the model evolves, for example because of bugfixes, extensions or reductions. Moreover, none of the models or their implementations offers an optimal platform for creating new models from scratch and convert them to the other two formats:

- The FEATURE EXPRESSION format supports a wide range of boolean expressions which makes it possible to represent an arbitrary model. FEATURE EXPRESSION models can be converted into SPLCONQUEROR models by using SPLCONQUEROR's boolean constraints. The disadvantage is that these SPLCONQUEROR models do not represent the high level semantics of the model and for that cannot be visualized. Although the model is basically human-readable, bigger models cannot be managed very well. The FEATURE EXPRESSION model parser of TYPECHEF lacks support of implicated alternative groups.
- SPLCONQUEROR's XML format is not human-readable and therefore needs tools support for generation. Like the FEATURE EXPRESSION model, bigger models with

hundreds of features cannot be managed very well without being liable to lose the overview of the model. This can easily cause errors in the model that result in bugs and the generation of unwanted variants.

- FeatureIDE's XML has the same disadvantages as SPLConqueror's XML format. Moreover, we use FeatureIDE only for visualizing the model and therefore it is not in the focus of our research.

Thus, we decided to create a new source format that satisfies our needs.

For a custom meta model format that can be used for generating all three models in our case study, we have the following requirements that had to be satisfied:

- The model should be human-readable to ensure that the model can be handled without additional tool support.
- The creation of the model should not require a graphical user interface.
- The memory consumption for parsing should be low as thousands of variants may be excluded. Using a DOM-based parser for XML for example consumes too much memory.
- The semantic structure of the model should be maintained. For example a group of alternatives should not be represented as boolean formulae.
- The model should be transformable into other target models, such as the XML formats of SPLConquerorand FeatureIDE or the Feature Expression used by TypeChef.

The following meta model format addresses all these challenges.

In order to avoid a complex notation we preferred using a simple text format to XML or JSON[4]. Blank lines will be ignored and can be used to separate statement blocks for a better structure of the model. Lines starting with *%%* are threaded as comments. The current reference implementation of the parser ignores the content of the comments. Basically, it makes sense for future implementations to transform the comments into target model comments if the target model supports comments. Every other line contains the actual directives that are used to construct the target models. A directive starts with the name of the directive followed by a whitespace and - depending on the type of the directive - a single feature, a list of features or a boolean expression.

The following directives with the described semantics are currently supported:

mandatory <feature>: Declares a mandatory feature. In SPLConqueror's model these features will be represented as direct non-optional children of the root feature. In the Feature Expression model it will result in a simple $defined(< feature >)$-expression and in an entry in the open-features file.

[4] JavaScript Object Notation

optional <feature>: Declares an optional feature. In SPLConqueror's model these features will be represented as direct optional children of the root feature. In the Feature Expression model it will result in an entry in the open-features file. The Feature Expression model itself does not need any modification here.

declared <feature>: Declares a feature that is not fully optional or mandatory but used in expressions. We use this to enable a better error reporting as we want to avoid the introduction of new features. This can be compared to the declaration of a variable. In SPLConqueror's model these features will be represented as direct optional children of the root feature. In the Feature Expression model it will result in an entry in the open-features file. The Feature Expression model itself does not need any modification here.

constraint <expression>: Creates a boolean expression that may use a common set of boolean operators, for example conjunction (&), disjunction (|) , implication (=>) and equality (<=>). We transform the expression into a conjunctive normal form. For SPLConqueror we extend the constraints section and add each term of the normal form as constraint. In the Feature Expression we simply add each converted term of the normal form.

alternative <parent> <feature>+: Declares an alternative group, where at least one child has to be selected. In SPLConqueror's model we add the child features as children of the parent feature. In the Feature Expression we convert this into a boolean expression.

route <parent> <feature>+: Declares an alternative group, where exactly one feature can be selected. In SPLConqueror's model we add the child features as children of the parent feature and define for each children the excludes of all other features of the group. In the Feature Expression we convert this into a boolean expression in conjunctive normal form. We do not use the $oneOf(< feature > +)$ expression here because it offers no support for implications, so the alternative children cannot be implicated by the parent feature.

excluded <feature>+: Declares an excluded variant. We transform the excluded into a constraint.

```
%% This is a comment.
mandatory   PRE_BASE
mandatory   PRE_ALGORITHMS
optional    PRE_LOGGING
declared    PRE_ALGORITHM_1
declared    PRE_ALGORITHM_2
declared    PRE_HELPER_1
route       PRE_ALGORITHMS PRE_ALGORITHM_1 PRE_ALGORITHM_2
constraint  PRE_ALGORITHM_1 => PRE_HELPER_1
excluded    PRE_BASE PRE_LOGGING PRE_ALGORITHMS PRE_ALGORITHM_2
```

Listing 6: Example for a feature model using different directives.

The listing 6 shows an application example of the meta feature model format using most of the directives presented above. Figure 4.1 shows the corresponding diagram to the model without the boolean constraints.

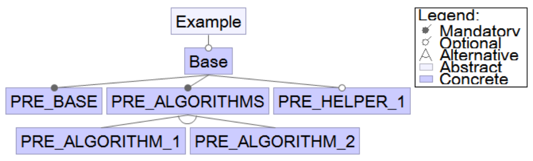


Figure 4.1: Visualization of the model of the example in listing 6.

The reference implementation is based on the .NET Framework 4.6 and is written in C# with smaller parts in F# and Iron Python. For converting boolean expressions into a conjunctive normal form we use the PBL library of the University of Utha [31].

4.3 Reverse Engineering Feature Models from Source Codes and Documentation

In this chapter we present our method to reverse engineer feature models from the source code and documentation.

4.3.1 Manual Pre-Processing

The first step of creating a feature model is to acquire as much information as possible to design a model that is close to the system. To get a first impression of each subject system we read the documentation of the system architecture and the configuration options. We used the architecture information to get the structure of the model, for example if the architecture suggested a cipher package we noted that the cipher algorithms can be placed in a cipher group. From the documentation of the configuration options we extracted additional dependencies between different options.

In the next step we inspected the source code and extracted all configuration options and the code snippets where they are used. We decided for every option if it represents a feature according to documentation and the usage in the source code. In many cases we found several options forming a single feature. For code generation we needed to define a wrapping parent feature, if none of the options had already presented one, and to define the multiple configuration options as mandatory children. Most of the systems provide a configuration utility, a configuration source file or a source code file to check the variability constraints. For example MBEDTLS uses a *config.h* that contains a list of nearly all configuration options[5] and a *check_config.h* that contains many, but not all, dependencies. Additionally, the source code documentation contains information for every option; how the option is called and in some cases also dependencies were listed. We process all these information and write it down for all features.

4.3.2 Generation using a Reduction Method

Our first approach was to create a model by reducing a model representing the whole system, meaning we started with a structured model that contained all features and all constraints we had found out in the previous phase. We added all constraints for the benchmark and made sure that the features are used by the benchmark.

This led to the problem that it is unknown for most subject systems how many variants we are dealing with because it is computationally infeasible to calculate all variants. For MBEDTLS for example we had a feature model initially represented as graph containing over 220 nodes and over 430 edges. Without the additional constraints we are talking of $2^2$20 variants.

In the next step we would have to remove features as long as we get a number of variants we can deal with. This approach is error-prone and the black box model for the number of variants does not scale.

We decided to discard this approach for our use cases. This approach provided the advantage to start with a model that is very close to the source code and is likely to meet the intention of the developers, but is we found out that it is not suitable for models containing tens or hundreds of features.

4.3.3 Generation using a Construction Method

Our second approach starts with an empty model. We add all mandatory features to the model that are required to get a working build that can be benchmarked. Not every subject system may have mandatory features as the core of the software may not be configurable at

[5]We detected some configuration options that can be modified but are not present in this file.

all. In the next step we add all optional features that have no dependencies to other features, for example such features as Logging. Then we add features with dependencies on other features, for example exchangeable algorithms often belong to a group of alternatives. In this step we only add as many features as we need to get enough variants, and we try to add features that are present in real-world use cases. For example, it makes no sense to add seven hash algorithms in an encryption product, but only one cipher if the real world use case is an equal distribute of the usage of the available ciphers. The last step is, of course, highly dependent on the structure of the feature model and requires in-depth domain knowledge to get it semantically correct.

4.3.4 Refinement

After constructing a model and generating all variants refinements have to be made.

The variants can be compiled and benchmarked with a low number of iterations to find variants producing compile-time and runtime errors. Runtime errors can, for example, be a result of unexpected combinations of algorithms. Such configurations reveal errors in the model or may be caused by missing dependencies. As not every error can be analysed in-depth we exclude such variants and define them as not valid.

Moreover, we used TypeChef to analyse the source code and find type errors that are present in some variants. This errors have to be analysed in detail because they may be a hint for missing constraints in the model, for example because of undocumented dependencies.

5 Detecting Internal Feature Interactions

In this chapter we present the types of internal feature interactions we detected.

5.1 Internal Feature Interactions Analysis Methods

To detect internal feature interactions we know different types of analysis methods. For understanding the different methods it is necessary to understand how interactions can be characterized: "To let features interact, we need corresponding coordination code. For example, if we want to coordinate the fire-alarm and floodcontrol features of the alarm-and-emergency system example, we have to add additional code for this task (e.g., to deactivate flood control in the case of fire)" [8].

Common methods to analyse aim to get information about this coordination code, for example the presence conditions for certain data flows or control flows, such as method calls.

A call graph, which is a control flow graph that represents method calls, can be extended to be aware of this variability by using presence conditions for edges that represent method calls. The next section will focus on this type of analysis.

Beside control flows are data flows analysis a method to detect interactions of features. A more advanced example for this is Janker's inter-procedural data flow analysis combining the tools TypeChef and SPL$^{\text{LIFT}}$ enabling a variability-aware analysis of even large scale C projects [16]. Less advanced analysis methods are capable of analysing data flow only intra-procedural.

5.2 TypeChef Analysis

TypeChef is a "a research project with the goal of analyzing ifdef variability in C code with the goal of finding variability-induced bugs in large-scale real-world systems, such as the Linux kernel with several thousand features (or configuration options). [...] TypeChef was started with the goal of building a type system for C code with compile-time configurations. TypeChef was originally short for *Type Checking Ifdef Variability*. Over time it has grown into an infrastructure of all kinds of analyses" [27].

As said in the description TypeChef has constantly evolved over the years and has been used in different publications in different areas of software research. Therefore TypeChef covers "a call graph analysis with a corresponding pointer analysis is currently developed in a fork" and "a variability-aware control-flow and data-flow analysis (subproject CRewrite)" [27].

As input, beside the source code, TypeChef can use "a variability model [that] describes the intended variability of the program" [32]. We favour the ability of TypeChef to process variability models in the feature expression format that are covering dependencies between features. All available configuration options of the variability model must be declared in a file that contains a newline separated list of these options. Additionally, we use a C-header file that contains all mandatory features as static configuration parameters. To generate these required files programmatic we use the conversion utility which is processing our meta model format.

For our main goal to find internal feature interactions we focus on the support of TypeChef to build variability-aware call graphs. In our work we are restricted to get one call graph per C source code file which was presented in chapter 4.

The purpose of the call graph data we generate for single C source code files is to aggregate them to a call graph for the whole subject system in the next step and to use the aggregated data for finding relations between internal and external interactions. This process exceeds the extent of this Thesis as well as contributing other analysis types would be beyond the scope as they are still in an early development stage. Moreover, we are currently not able to find indirect control flow interactions, such as calls using function pointers instead of direct calls, due to technical limitations of the current TypeChef implementation.

```c
void f1 (int argc, char *argv[]) {
    #ifdef A
        #ifdef B
            f2();
        #enfif
        #ifndef B
            f3();
        #endif
    #endif
}
#ifdef B
    void f2 () {
        wait(10);
        f3();
    }
#endif
#ifdef A
    void f3 () {
        printf("Task complete.");
    }
#endif
```

Listing 7: A configurable program using C pre-processor directives.

Listing 7 illustrates source code using C pre-processor directives to implement variability. The function *f1* is always compiled, function *f2* is compiled if feature *B* has been enabled and function *f3* is compiled if feature *A* has been enabled. If a feature is not enabled, meaning the pro-processor directive has not been defined, the corresponding function will be removed by the pre-processor and therefore not be compiled.

If both features *A* and *B* are enabled function *f2* is called, if feature *A* is enabled but not feature *B* then function *f3* is called.

In other words, we have the presence condition for the call *f1* → *f2* of *A & B*. For the call *f2* → *f3* we have the presence condition *A & !B*.

Figure 5.1 shows the corresponding call graph. In this case we have a feature interaction of the features *A* and *B* that affects the overall performance.

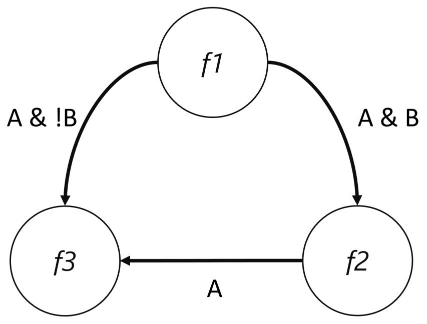


Figure 5.1: Minimal example of a variability-aware call graph.

In general the function *f2* can do optimizations before function *f3* is called. This would result in an overall speed-up. On the other hand it can contain for example logging functionality, which would result in a decrease of the overall performance. The example shown in figure 5.1 contains a *wait(10)* call, that causes the program to wait 10 seconds. This feature interaction is visible in the call graph, but we do not know the effect on performance.

6 Detecting External Feature Interactions

In this chapter we discuss the measurements of non-functional properties we use to detect external feature interactions and present our testing-framework.

6.1 Types of Non-Functional Properties

We are able to measure different types of non-functional properties we can use to detect external feature interactions. Most of them can be seen in the context of performance. For our work the following types of non-functional properties are evaluated:

Binary Footprint: The file size of the compiler output library or program. This non-functional property has often been a part of the research of highly configurable systems, for example used Siegmund the property in his work on "Measuring Non-Functional Properties in Software Product Line for Product Derivation" [33]. The size does not change over time and is easy to measure. We will collect the information, but we will not focus on the property, as it has no main impact on the runtime performance. We decided to collect the property to allow further usage in future work.

Memory Footprint: The amount of memory that is consumed while the program is running. The value may change over time, so we can record for example the peek value or the average or median values for a program execution. Measuring the real memory consumption is not trivial if we do not use a memory profiler such as Valgrind[6]. The Unix tool time for example measures the memory consumption only at certain intervals. This can lead to unreliable and unreproducible results.

Energy Consumption: Using power distribution units (PDU) that are capable of measuring the energy consumption per node enable us to measure the energy consumption while a benchmark is running. We should be aware of the accuracy of the measurements, as many other factors may have an influence, such as the temperature of the environment.

Execution Time: The execution time can basically be defined as the time the program needs to complete a certain task. An important decision is, if the startup and shutdown times of the program are included in the measurements or not. Depending on the use case this may have a significant impact on performance. If a real-world use case includes the startup time it should be considered to be part of the measurement, it it does not include the startup time should be considered not to be part of the measurement [34].

[6]http://valgrind.org/

Throughput of Connections: The property is related to the execution time, but can be refined, as the actual throughput of a connection between two endpoints may not be constant over the time. For example we can measure the peek throughput, the median or average throughput. The average throughput is the amount of data that is processed divided by the execution time if we assume the data is not compressed. For example we can measure the throughput of a network connection or in memory.

6.2 Role of Unused Code

One of our goals is to avoid unused code as far as possible, in order to get all code, that is present in a build benchmarked. This means we want a one-to-one mapping between the compile-time and the runtime configurations of the variants. Therefore, we executed a pre-experiment to find out, if we have a significant performance difference between a variant that contains features that need an explicit enabling, but are not used, and variants that are using all features present.

We took a variant of MBEDTLS that only contained the algorithms that are actually used, and we took a variant that contained all algorithms, even the ones that are not used in the benchmark. We did this for 32 randomly picked variants. None of the variants showed a *significant* performance difference, to the variant containing all the features using the same runtime configuration. Figure 6.1 shows typical violin plots for the measurements for the same runtime configuration with 30 iterations. As we can see are the median and the interquartile ranges for both measurements equal, and the violins are identical within the tolerance of our measurements.

For our use case this indicates that features not only have to be present to be relevant for performance, they actually have to be used. This fact cannot be generalized. For example on an embedded platform with much more limited resources the presence of unused code may have an effect on the overall performance, as the idling program consumes a significant amount of the main memory, compared to our use case where the idling program itself consumed under 0.1% of the total available main memory.

6.3 Compiler Influence

Different compilers can produce different measurement results, especially the filesize of the binary is clearly affected. Moreover, the optimization level of the compiler changes the results. For example when we tested building the library module *libmbedtls.a* of our subject system MBEDTLS with different compilers and optimization settings, we get slightly different results. For gcc 4.9.3 on a 64 bit Windows machine we get the following filesizes: These results have been expected because the different optimization levels perform different transformations like inlining of methods. *O0* does not perform any optimizations and therefore generates the biggest file. The optimization levels *O1* and *O2* produce a significant smaller binary. Different optimizations like the removing of unreachable code take effect. *O3* is the highest optimization level and produces a bigger binary than *O1* and *O2*. This is not surprising as *O3* inlines functions even if they are not declared as inline functions[7]. TCC[8] 0.9.26 produces a binary with 221 KB using the default settings.

The results are deterministic in all four cases and fully reproduceable. For example there is no time based sampling in the C compilers present like Java's JIT compiler implementation uses. However, the content hashes of the binaries are not reproduceable because the binaries contain timestamps with a fixed size.

The common real world use case is that *O0* won't be used because it does not perform secure and trivial optimizations. *O3* won't be used for deployment because it performs optimizations that may cause bugs. As *O1* and *O2* produce nearly the same results we use the default setting of the mbedTLS build system that referrers to *O2*. We should keep in mind that the hypothesis may not hold for all optimization settings in general and different compilers may not reflect the settings of gcc.

[7]https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[8]http://bellard.org/tcc/

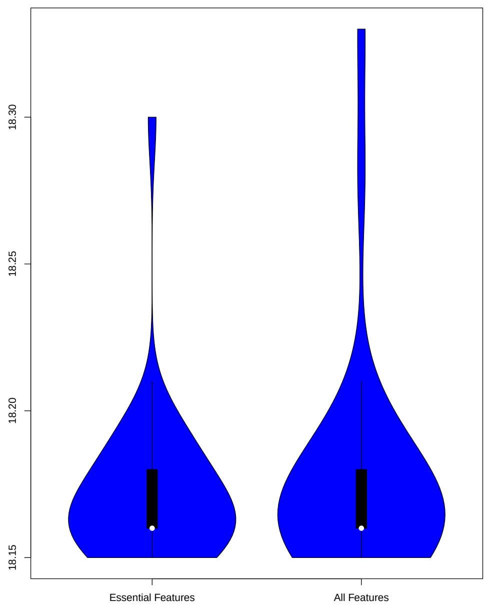


Figure 6.1: Violin plots for measurements of the same runtime configuration. The left plot it for variants containing only features actually used, the right plot is for variants containing unused code. The y-axis represents the duration of a connection in seconds.

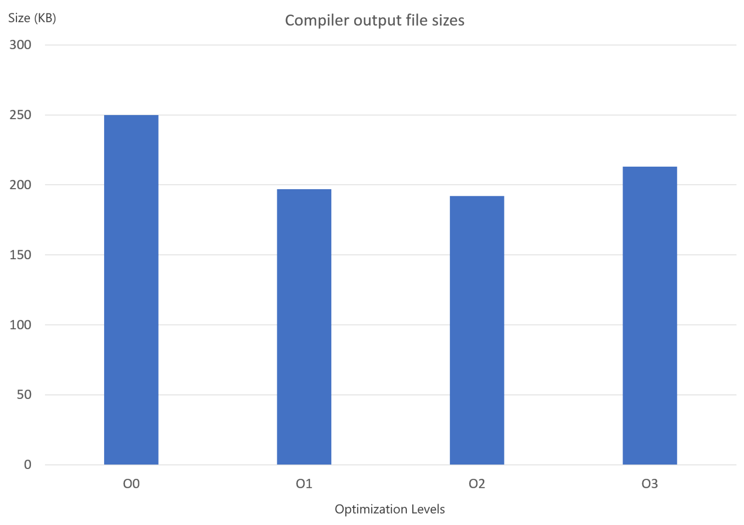


Figure 6.2: Output file sizes in KB using optimization settings between O0 and O3.

6.4 Network and IO Influences

The network and the disc IO can cause unpredictable and heavy delays and both can be a bottleneck and limiting factor for our measurements. To get the non-functional properties we need to avoid as many limiting factors as possible. Such limitations can cause variants that have actually a different performance behaviour to show the same behaviour. For example if the network bandwidth limits the throughput, two variants may need the same time to transfer the same amount of data because the faster variant cannot send faster than the network allows. Therefore no clear results can be expected if the maximum performance is limited by the network or disc performance.

We avoid disk access completely on the test systems by using RAM discs instead of the much more slower NFS mounted network file system, and we try to avoid network delays as far as possible. Network delays can for example be avoided if we use the same machine for hosting the server and the client in a client-server scenario. However, this is not a solution that can be applied for all possible subject systems, for example web servers need multiple clients for the simulation of a real-world use case.

6.5 Testing System

We use a cluster of 14 machines with identical hardware and software configurations for our measurements. Every machine has a Intel i5-4590 processor with four cores, four threads and 6 MB first level cache. They use 16 GB RAM and 256 GB solid state disks. Half of the RAM is used as RAM disk. As compiler we use GCC 4.9 on a Ubuntu Linux in version 16.04.

The cluster uses a power distribution unit (PDU) which can record the energy consumption per node. The power distribution units offer a XML-based API to retrieve the current consumption values. We store this data, but we have to be aware of inaccuracies caused by network delays.

We are able to execute the benchmarks in parallel.

6.6 Statistical Variables

In this section we discuss the independent variables and dependent variables, as well as the confounding variables, that are present in our test system.

6.6.1 Independent Variables

As dependent variables for all subject systems and all external feature interactions we want to record we define the configuration that forms a variant as independent variable. A configuration is a set of features that is present in the variant.

6.6.2 Dependent Variables

We have multiple variables we want to observe and that are supposed to be depending on the configuration of the variant.

Binary Footprint: Using *#ifdef*-variability changes the source code that is actually compiled. We expect significant changes in the binary sizes. The recording of the variable is trivial.

Memory Footprint: The presence of different features can affect the memory consumption. For example the size of the program itself changes, and moreover, features can consume memory if they are used. The recording of the variably is not trivial and needs a memory profiler to ensure accuracy. In the following we will record the data using the command line utility TIME, as memory measurements are not our primary goal. We will be aware that we have to drop the data if the accuracy is too bad.

Energy Consumption at Runtime: If a feature is active we suppose that a feature either will consume CPU time or perform optimizations that changes the CPU time consumption of other features. Concluding we expect a change in the overall energy consumption of the program. The basic measurement is possible as we can collect data from the API of the power distribution units, but we expect a low accuracy caused by network delays and other sources for measurement bias. We collect the data but will not further evaluate the data if the quality does not fit our requirements.

Performance at Runtime: Performance can be characterized in different ways, for example we can measure throughputs and queries per second. We decided to use a simple measurement that can be used for all subject systems: The execution time of the program to complete a certain resource intensive task. The execution time can be measured very accurately. Performance interactions are our main interest, so we will focus on them in the following chapters.

6.6.3 Confounding variables

We identified the following confounding factors and tried to mitigate them using different techniques, such as keeping the value of the variable constant or using randomization as far as possible.

Compiler and software environment: We described in section 6.3 the effect of compiler optimizations on the size of the binary. The same optimizations can affect performance or course. We use always O2 settings for optimization and keep a constant software environment. For example we avoid updates and used the same software versions on all nodes.

Runtime configuration: Subject systems may offer methods to change configuration options at runtime. We use fixed runtime configuration parameters or use a one-to-one mapping to the compile time configuration.

Latencies: Energy measurements require a communication with an API which has a certain response time. Additionally, we have to deal with latencies between the client of the measurement and the provider of the API. Our possibilities to mitigate this effect are very limited with the current benchmark setup.

Network and IO overhead: To avoid overhead caused by the network and the IO of the nodes we will use RAM disks to mitigate this effect.

Environment effects and random effects: For the mitigation of random effects, such as unavoidable operating system background tasks, and environmental effects, such as an increasing system temperature we use randomization of the variants and multiple executions of our measurements. Moreover, we use a high number of iterations.

Utilization of processor and basic workload: Every operating system and the active background tasks cause a base load on the hardware we are not able to control. With our measurements we found out that we can utilize between 98 % and 100 % of the processor if a benchmark is limited by the processor. We have to keep this fact in mind when we analyse the standard derivation and the variance. This may be a cause for significant differences for both values of different measurement series.

Subject system related variables: For each subject system we will discuss additional confounding variables that are depending on the investigated system.

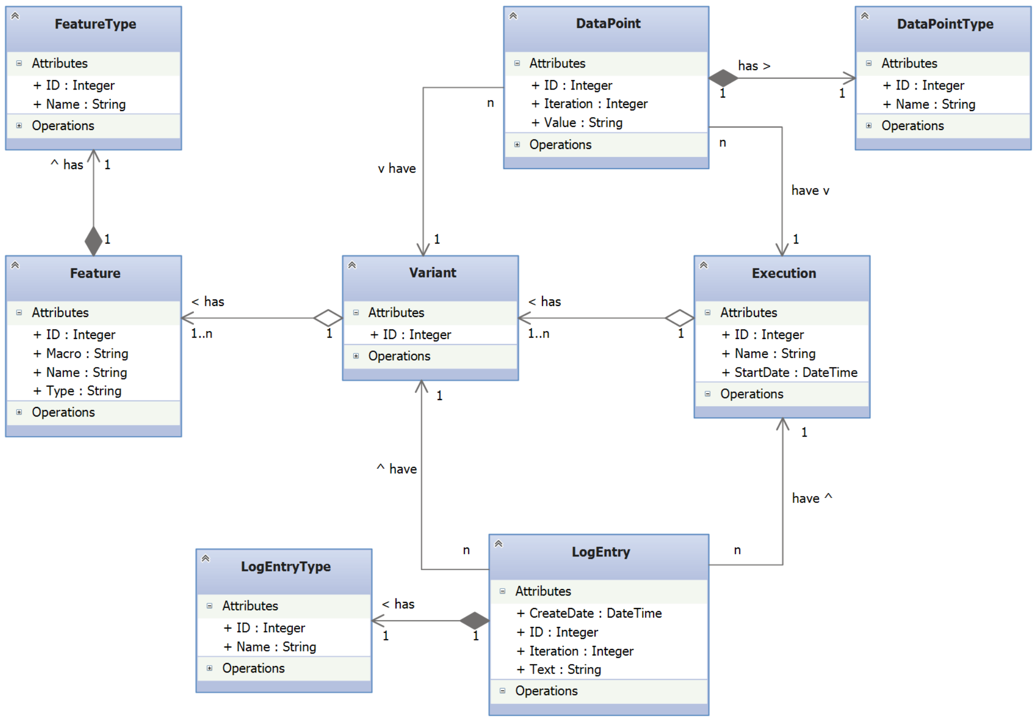


Figure 6.3: UML model of the reporting database.

6.7 Benchmark Execution and Reporting Framework

We implemented the benchmarking infrastructure using Python. A wrapper script generates the compilation, benchmark and measurement workflow jobs that can be executed on a node of the cluster. Every job is a python script that manages the RAM disk, triggers compilation process and executes the benchmark. The script implements a detailed logging and measures the time of each sub-task, such as the compilation, the whole benchmark duration or a single iteration of the benchmark. The implementation is designed to generate as little overhead as possible.

For getting the filesize of the compiled binaries we use Python's *os.path.getsize(file)* function. The time and memory recordings are done by using time[9]. Energy measurements use the XML-based API of the power distribution units, where we store the snapshots of the raw data.

We write every measurement data point and every log entry in log-files on the network file system. To enable a faster reporting while the subject system is benchmarked and to provide a better exporting functionality for the data we use an additional SQLite database. Figure 6.3 shows the UML class digram of the database structure we use for storing the measurements and logs. The database is located on a remote server and uses a JSON-

[9]On the Windows reference environment this is done via Cygwin.

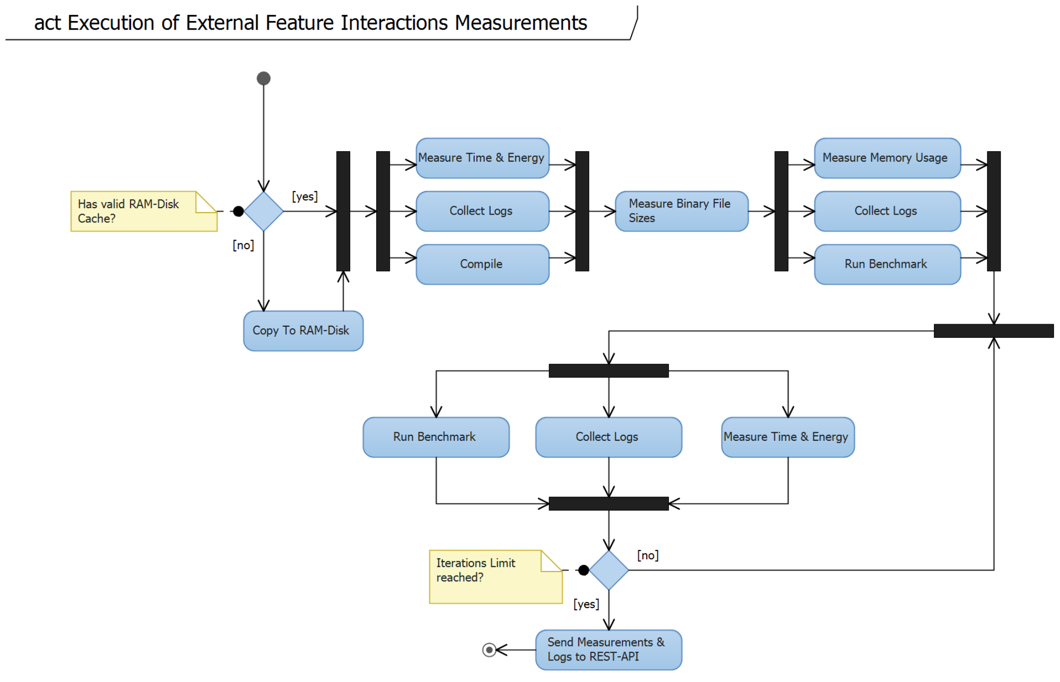

act Execution of External Feature Interactions Measurements
Has valid RAM-Disk Cache?
[yes]
[no]
Copy To RAM-Disk
Measure Time & Energy
Collect Logs
Compile
Measure Binary File Sizes
Measure Memory Usage
Collect Logs
Run Benchmark
Run Benchmark
Collect Logs
Measure Time & Energy
Iterations Limit reached?
[no]
[yes]
Send Measurements & Logs to REST-API

Figure 6.4: UML model of the standard workflow of the benchmark.

based[10] REST[11]-API. We send the data to the API using cURL at the end of every job execution, to avoid any effect on the benchmarks.

Figure 6.4 illustrates the workflow of the whole benchmark process as UML activity diagram. The term benchmark refers to the performance benchmark we use. We implemented this part very generically, for example we can use a complex client-server model for benchmarking or a simple program execution. The workflow can be executed as single job.

[10] JavaScript Object Notation
[11] Representational State Transfer

7 mbedTLS

This chapter describes the experimental setup and execution to create variability-aware call graphs and to measure non-functional properties in order to find internal and external feature relations in the subject system mbedTLS.

7.1 Description of mbedTLS

mbedTLS is a lightweight open source library for transport layer security designed for embedded platforms. It offers support for the SSL/TLS protocol versions SSLv3, TLS v1.0, TLS v1.1 and TLS v1.2.

The main use case for mbedTLS is - according to ARM - an embedded environment. Being available on most platforms and for most operating systems mbedTLS is also used in non-embedded software like OpenVPN. Therefore, it can be considered as alternative to the widely-used OpenSSL library whose code quality was the subject of relentless criticism after the heartbleed attack.

mbedTLS has a very modular and well designed and documented software architecture. Compile-time configuration options can easily be identified because they are prefixed with MBEDTLS_.

mbedTLS uses make as build system and includes a range of demo applications and test cases. The compatibility test application is re-used for the benchmarks of the case study.

7.2 Software Architecture

Next we provide a more detailed overview of the software architecture of mbedTLS, cipher-suites in common, the feature model design and finally, the setup, execution and results of the benchmark. The case study uses the mbedTLS version 2.2.1 which was released in January 2016. For prior experiments and evaluation PolarSSL[12] 1.3.9 was used.

mbedTLS is generally designed to be a library for multiple cryptographic purposes related to the context of Transport Layer Security. Basically, single modules of the library can be used completely independent without providing the whole SSL/TLS functionality. For example, it is possible to use only the hash module in a third-party application for hashing data. The other modules are not needed to be compiled in this case. For our case study we use mbedTLS as a SSL/TLS library, that is all modules in combination. Partial variants that are not supporting a full SSL/TLS communication will be ignored because we are interested

[12]PolarSSL was renamed to mbedTLSin 2016

in detecting feature interactions and the SSL/TLS module contains most of the coordination code that is in the focus of our interest.

Figure 7.1 illustrates the high-level architecture of mbedTLS. The follwing modules form the library:

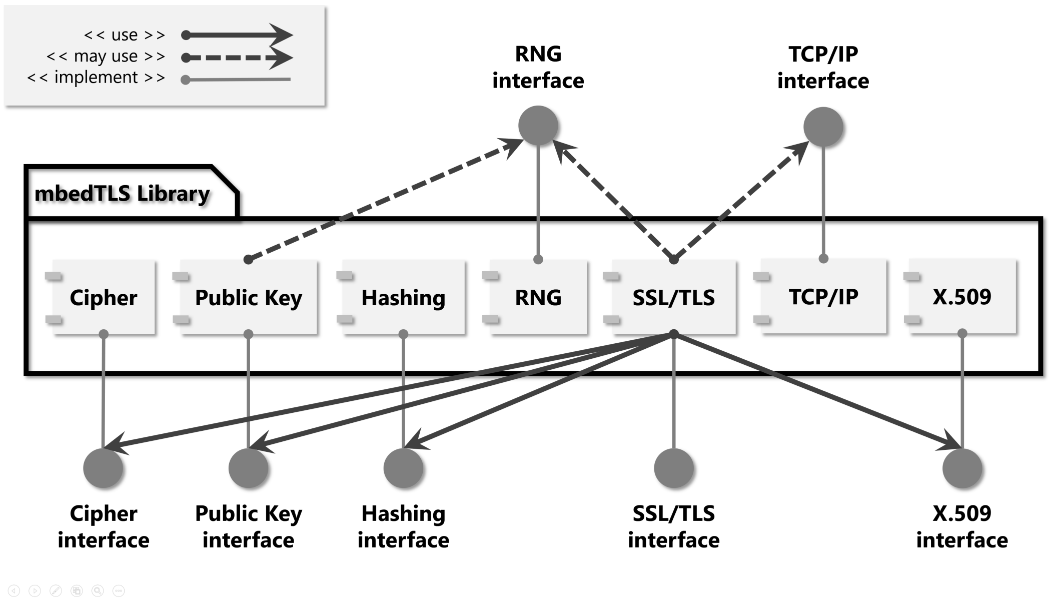


Figure 7.1: mbedTLS high-level architecture overview [35].

Cipher: The cipher module contains all supported symmetric cipher-algorithms such as AES, DES, and ARC4. They implement the cipher-interface and can be used exchangeable. Every cipher needs to implement minimum one cipher operation mode. The operation mode defines how data amounts larger than the block size can be encrypted with or without generating patterns in the encrypted data. Patterns in the encrypted data can be used to break the encryption. Beside the different block modes also a stream mode can be implemented. Most algorithms only implement a sub-set of the available modes.

Public Key: The public key module contains the public key infrastructure for handling secure key exchanges between server and client. This involves the usage of asymmetric encryption algorithms such as RSA and key exchange protocols like Diffie-Hellman. Every key-exchange protocol implements the public key interface and is exchangeable. The module makes heavy use of the random number generator.

Hashing: The hashing module wraps the hash-algorithms such as RipeMD 160, MD5, and the algorithms of the SHA family. To be exchangeable they implement the hashing interface. Hash functions are mainly used by the SSL/TLS module for message authentication and by block ciphers for chaining blocks.

RNG: The RNG module handles the random number generation. The public key exchanges depend on this module. One particularity of this module is that it can use the AES cipher as core of a random number generator (RNG). In this case the AES cipher cannot be excluded from the source code, even if another cipher, for example DES, is used for encryption. This poses certain problems for the feature model that are discussed in section 7.5. Implementing the random number generator (RNG) interface makes the default algorithm exchangeable.

SSL/TLS: This module is the core module of the library that puts all other modules together in order to provide a SSL/TLS interface that enables the creation of encrypted secure connections. Figure 7.2 shows the usage hierarchy of the modules.

TCP/IP: mbedTLS does not rely on any platform dependent C library function other than the functions provided by *libc*. The TCP/IP module offers an interface and an implementation for a complete TCP/IP protocol stack.

X.509: The X.509 module contains the X.509 certificate infrastructure for parsing, generating and validating certificates according to the X.509 standard. Although there is only a single implementation in mbedTLS the module implements a X.509 interface that makes the implementation exchangeable with other implementations.

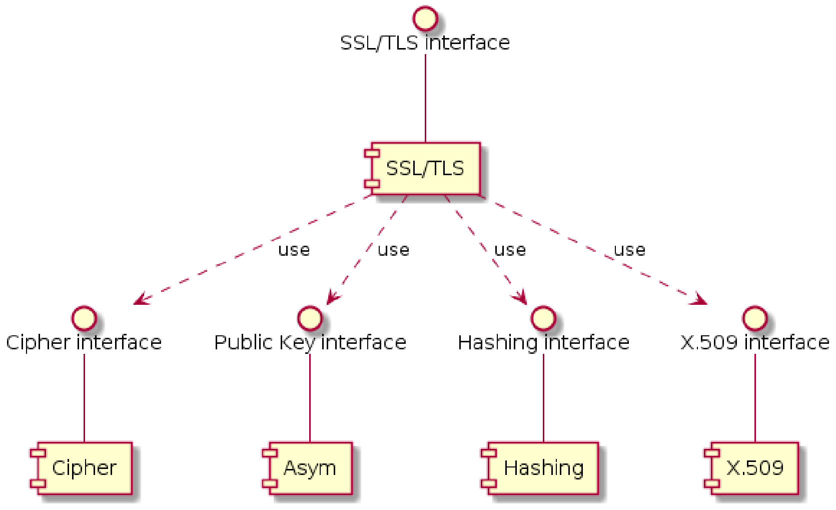


Figure 7.2: mbedTLS hierarchy and usage of modules [35].

7.3 Cipher Suites

mbedTLS supports different types of variability. On the one hand, we have the compile time options that define what parts of the source code are compiled and how they are compiled, on the other hand, there is runtime variability which can be redefined for every connection. The specification of a connection is called cipher suite and is shown in listing 7.1. It defines what algorithms and protocols are used by a TLS connection and is therefore represents runtime variability in mbedTLS. Attention has to be paid that TLS 1.3[13] uses a different

[13]Currently TLS 1.3 has the status "draft", see: https://tools.ietf.org/html/draft-ietf-tls-tls13-11.

format and cannot use the format below. Therefore, upgrading the subject system to use TLS 1.3 will require adjustments.

$$TLS\_ECDHE\_PSK\_WITH\_AES\_128\_CBC\_SHA256 \tag{7.1}$$

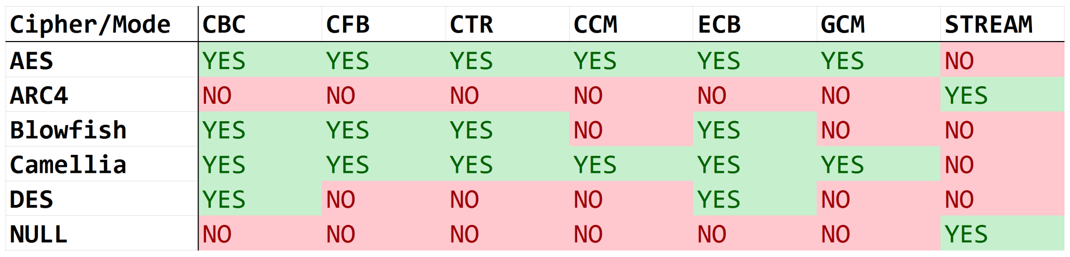
Protocol: The first part of the cipher suite defines the protocol that is used for the connection, but does not include any version information. Our subject system does not use the outdated SSL protocol any more, so the protocol is always TLS.

Key-exchange: For setting up a secure connection between client and server the key that is used for the encryption algorithm has to be exchanged. The key exchange part of the cipher suite defines which mechanism or which protocol is used for this. ECDHE_PSK states that the elliptic curve Diffie-Hellman protocol is used in combination with a pre-shared key.

Cipher algorithm and mode of operation: Within the cipher suite specification the token WITH is used as separator between the key-exchange and the cipher information. After these token the cipher algorithm is specified and optionally it contains information about the mode of operation, if more than one mode is supported by the cipher. The example uses AES_128_CBC which states that the algorithm AES should be used with 128 bit key size[14]. As mode of operation CBC is used.

Figure 7.3 illustrates which cipher algorithms can be combined with which modes, because a cipher does not necessarily implement all modes. Using ARC4 or no cipher implicates the stream mode, all other ciphers implicate at least one of the block cipher modes.

Hash algorithm: The last part of the cipher suite defines the hash function that is used for the message authentication code. In the example SHA256 defines that SHA 2 algorithm is used with a 256 bit output. It is important to keep in mind that the connection setup of TLS requires other hash functions for different purposes such as key derivation. The specification of SHA256 in the cipher suite does not affect these functions, because they are defined by the protocol itself and cannot be configured.



Cipher/Mode	CBC	CFB	CTR	CCM	ECB	GCM	STREAM
AES	YES	YES	YES	YES	YES	YES	NO
ARC4	NO	NO	NO	NO	NO	NO	YES
Blowfish	YES	YES	YES	NO	YES	NO	NO
Camellia	YES	YES	YES	YES	YES	YES	NO
DES	YES	NO	NO	NO	YES	NO	NO
NULL	NO	NO	NO	NO	NO	NO	YES

Figure 7.3: mbedTLS compatibility between cipher algorithm and cipher modes.

[14] AES has a fixed block size of 128 bit.

The negotiation of the cipher suite is done between the sever and the client in the setup phase of a connection. For this case study we will always support a single cipher suite only for specifying explicitly which cipher suite is used. Basically the protocol supports a re-negotiation for an active connection, but we do not use this feature to ensure the transparency of the process. Enabling the feature would create for example the problem, that the cipher suite used at the end of the connection need not be the same cipher suite that was used initially.

7.4 Feature Model

The Config.h, which contains the configuration options that define the variant, of mbedTLS contains 247 macro definitions that we can consider to represent a feature. Most of the macros are prefixed with Mbedtls_, but we have found some exceptions. Some macros for example only use the cipher mode as prefix. Beside the Config.h mbedTLS has a Check_config.h that defines constraints for features and generates compilation errors if this constraints are violated. After reading the high level documentation of mbedTLS we processed the documentation of the files Config.h and Check_config.h. We found out that often multiple macros form a single feature.

The overall structure of the feature model follows the packages shown in figure 7.2. We had to acquire deep domain knowledge as we needed to build the model from scratch.

In our first approach we extracted all features from the Config.h and added the dependencies described in the Check_config.h. After that we grouped the features into the packages of the architecture diagram and added calling dependencies for each feature to avoid unused code. This approach lead to a feature model that represents the real software architecture but is not suitable for benchmarks as it allows millions of variants to be generated.

For getting a computational feasible model we use a new approach that only uses the essential parts for mbedTLS. Moreover, we try to avoid unused code to be present in the build. The construction of the model follows the following steps:

Mandatory Features: In the first step we determined the mandatory features for a minimal product supporting the SSL/TLS protocol module. For this case study we use the TLS version 1.2 as only supported protocol. We used the documentation and test builds for this. We found out that some of features of other modules, such as the cipher AES or the hash MD5 are part of the mandatory features. The use of AES is not exclusively bound to the cipher suite, but it is also used for the mandatory random number generator. The problem details and the solution for the case study is described in section 7.5.

Cipher Suites: Next we chose features to build cipher suites that are constructed by a hash function[15], a cipher algorithm[16], a mode of operation for the the cipher algorithm[17]and a method for the exchange of keys[18]. As described in the cipher suites section 7.3 we are only able to combine cipher algorithms with compatible operation modes. Basically it is possible to combine every hash function with every cipher and every key exchange mechanism. Most key exchange methods require multiple other macros to be turned on in order to work as intended. To limit the dependencies to a minimum we only used the simple PSK methods. Other methods require more advanced features like a certificate infrastructure. This is error-prone to emulate for benchmarks and not required to get enough working variants.

Optional Features: After adding cipher suites, we add features that are fully optional, which means the features can be turned off and on without any other implications. Moreover, these features do not require a runtime configuration, they will be active if they are present. For example, if AESNI_C is present the AES implementation will switch automatically to the native instructions (NI) variant.

Benchmark Features: For being able to create a executable benchmark binary the essential features of the benchmark wrapper are added, for example the IO functions to read the contents of the payload require the feature MBEDTLS_FS_IO.

Constraints: After the first benchmark execution we detected that many configurations are not working because of compile time or runtime errors. The errors occur for example because of type errors or undocumented dependencies. We added these variants as excluded configurations to the model. These variants will not be part of the analysis.

The figure 7.4 shows the feature model that was created with this process. We removed the mandatory features from the graphics as they cannot be configured and are therefore always present. At the top level, there are four modules of the cipher suite that are only containing alternative groups and we can see the group of optional features at the bottom.

The figure only illustrates the semantic features but does not represent the model we used for the code generation. To create a working build of the mbedTLS library we need to enable several additional macros that are implicated by single features. Because of the implications by single features and because this macros cannot be defined stand-alone we assume this to be a part of the feature.

As the feature model of mbedTLS is very complex[19] and auto generated from a meta model, we want to ensure that the variants calculated by SPLConqueror are correct. Therefore we wrote a Haskell program that generates all variants and we matched the output of the Haskell program with the output of SPLConqueror. Both outputs contain the same 19,200 variants so we assume the the input model of SPLConqueror was correct. The documented source code of the program is show in listing 8. We removed all Mbedtls_ prefixes in macro

[15]RipeMD160, SHA1, MD5, SHA256 or SHA512
[16]AES, ARC4, Blowfish, Camellia, DES or none
[17]CBC, CCM, CFB, CTR, ECB, GCM or stream
[18]PSK, DHE-PSK or ECDHE-PSK
[19]The SPLConqueror model contains all mandatory and all implicated macro definitions.

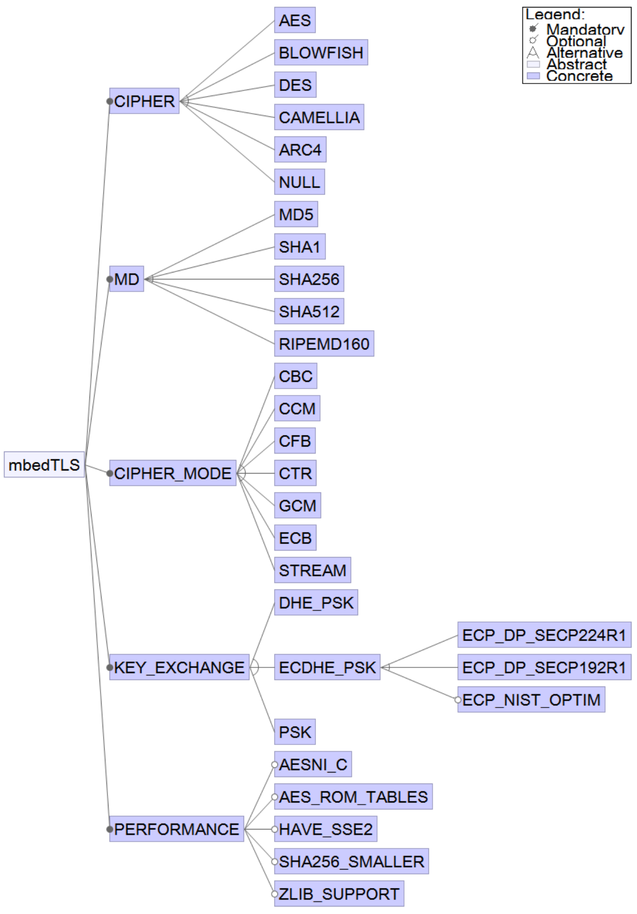


Figure 7.4: Feature model of mbedTLS used for performance benchmarks.

names and do not show the external dependencies for brevity. External dependencies are for example mandatory features that are required by the benchmark implementation itself, such as the TCP/IP or file IO implementation. The original version is available online[20].

7.5 Shared Code

A problem we had to solve is that some features are shared across different modules and used in different contexts. For example AES is used by the random number generator, which is mandatory, and is also used as cryptographic block cipher algorithm. Another example is the implementation of the hash algorithm MD5 that can be used in the context of the cipher suite for generating the MAC, and it is also required by the TLS module for the connection setup.

This creates the problem that we cannot use an exact one-to-one mapping between compile-time and runtime configuration, because some modules are mandatory in the code base for all variants, but are not actually used as they are intended to be used. We need a

[20] https://github.com/DE120/mbedtls/blob/master/feature-models/variants.hs

```haskell
module Variants where

-- NOTE: A function to calculate the power set of an list was taken from
-- https://mail.haskell.org/pipermail/haskell-cafe/2003-June/004484.html.
-- Signature: powerset :: [a] -> [[a]]

-- The list of all optional features that can be off and on without other dependencies (5).
optionalFeatures = ["HAVE_SSE2", "AES_ROM_TABLES", "SHA256_SMALLER", "ZLIB_SUPPORT", "AESNI_C"]

-- The list of hashing algorithms (5).
hashes = ["RIPEMD160_C", "SHA1_C", "MD5_C", "SHA256_C", "SHA512_C"]

-- The ECDHE-PSK key exchange requires one elliptic curve.
-- Additionally an optional optimization feature that can be activated (4).
ecdhepsk           = [["KEY_EXCHANGE_ECDHE_PSK_ENABLED", curve, optimization] |
    curve         <- ["ECP_DP_SECP192R1_ENABLED", "ECP_DP_SECP224R1_ENABLED"],
    optimization  <- ["ECP_NIST_OPTIM", []]]

-- Combine the two simple key exchanges with the more complex ECDHE-PSK key exchange (6).
keyExchanges = [["KEY_EXCHANGE_PSK_ENABLED"], ["KEY_EXCHANGE_DHE_PSK_ENABLED"]] ++ ecdhepsk

-- Create all possible combinations for cipher modes and the supported cipher algorithms (20).
ciphers = [["CIPHER_MODE_STREAM", c] | c<-["ARC4_C", "CIPHER_NULL_CIPHER"                   ]]
       ++ [["CIPHER_MODE_CBC",    c] | c<-["AES_C",  "BLOWFISH_C", "CAMELLIA_C", "DES_C"]]
       ++ [["CIPHER_MODE_CFB",    c] | c<-["AES_C",  "BLOWFISH_C", "CAMELLIA_C"         ]]
       ++ [["CIPHER_MODE_CTR",    c] | c<-["AES_C",  "BLOWFISH_C", "CAMELLIA_C"         ]]
       ++ [["CIPHER_MODE_ECB",    c] | c<-["AES_C",  "BLOWFISH_C", "CAMELLIA_C", "DES_C"]]
       ++ [["CCM_C",              c] | c<-["AES_C",  "CAMELLIA_C"                       ]]
       ++ [["GCM_C",              c] | c<-["AES_C",  "CAMELLIA_C"                       ]]

-- Now combine the power set of the optional features with the cipher suite information.
-- The cipher suite contains one hash, one cipher operating in a cipher mode and a key exchange.
-- Total we get 2^5 x 5 x 20 x 6 = 19,200 possibilities.
variants         = [optionals ++ [hash] ++ cipher ++ keyExchange |
    optionals   <- powerset optionalFeatures,
    hash        <- hashes,
    cipher      <- ciphers,
    keyExchange <- keyExchanges]

-- Print all possible combinations.
main = print variants
```

Listing 8: A Haskell program for generating a list of all possible variants according to the feature model for verifying the SPLConqueror output.

one-on-one mapping, because otherwise the benchmark is not able to decide which variant should actually be benchmarked. For example if AES and DES are present, we want to benchmark the DES cipher because AES is only needed for the random number generator. Programmaticly this is hard to decide as we do not know the context of AES and DES. Additionally, AES used as cipher has a significantly different impact on performance than AES used as part of the random number generator which is used only a few milliseconds for connection setup.

To solve this problem we introduce an extra feature for every context, that refers to the implementation of the feature. For example we split up *MBEDTLS_AES_C* into *MBEDTLS_AES_C* and *MBEDTLS_AES_C_CIPHER_USAGE*. This allows SPLConqueror differentiate between AES feature being used as a cipher and as part of the random number generator, but still including all mandatory code, and allows the benchmark a one-to-one mapping between the cipher suite that is used and the cipher algorithm that is present in the binary.

The data can be evaluated after merging *MBEDTLS_AES_C* and *MBEDTLS_AES_C_CIPHER_USAGE* back into a single feature. If the features are not merged, we will not be able to analyse relations to the control flow graphs.

7.6 Internal Feature Interactions

To detect the internal feature interactions we converted the feature model from the meta format, described in 4.2.2, to a Feature Expression model suitable for TypeChef. Beside the Feature Expression model we created an *open-features.txt* file that contains all configuration options. We prepared the source code of mbedTLS to work with TypeChef, for example we needed to removed the content of the file config.h. If a configuration is defined in the header config.h, TypeChefwill use this configuration instead of the variability expressed by the feature model.

To test if the preparations of the feature model and source code are working for TypeChef we generated variability-aware control flow graphs for all source code files that contain the presence conditions for all calls. These can be used to detect feature interactions, as described in 5.2. The analysis of variability-aware control flow graphs and the extraction of the feature interactions is part of further work.

7.7 External Feature Interactions

This section describes the creation of a performance benchmark for mbedTLS. We present the execution of the measurements, the measured data, discuss the results and show examples for external feature interactions.

7.7.1 Benchmark Description

We found no publicly available benchmark that is capable of performance benchmarking the library itself. Therefore, we used as base for the custom benchmark of the connection performance the compatibility test of mbedTLS that has on the one hand a server wrapper application and on the other hand a client wrapper for application for the library. The compatibility test is shipped with the original source code.

We evaluate the connection performance by measuring the time that is needed to transfer a fixed amount of data from the benchmark server application to the benchmark client application. We chose to measure this property of TLS connections because it is the value that is interesting for most real-world applications using secure connections, such as a browser that receives data from a web-server, a mobile or an embedded device that queries data from an API.

We compile the server and the client with the same library, so both contain the same features. The server is an application, that starts up and loads a payload file in the memory and starts waiting for a client. If a client connects the TLS connection between both is established and the server starts transmitting the payload to the client. The cipher suite is that is used for the transmission is explicitly defined by the wrapper script that started the performance benchmark. The server is terminated by the wrapper script if all client benchmarks are done.

The server has a high startup time to load the file into the memory, so we decided to focus our benchmark iterations on the client and keep the server alive for all iterations. We have to consider this while processing the benchmark data, because either the decryption or the encryption process may be the limiting factor, although they share most of the code.

We had to modify the code of the original server in order to add support for payload data to transmit to the client that is read from instead of being defined in the source code as constant of characters. The original payload data had a size less than 1 KB. Moreover, we had to add support for multiple clients, as the original compatibility test quit after serving a single client.

The client is a minimal wrapper around the library which can establish a connection to a server, receive data and quit after the connection is closed by the server. It has no measurable startup and shutdown time.

As payload we use a 2 GB file that is based on a base64-encoded image embedded in HTML. Smaller payloads have shown unstable measurement results in pre-tests. The payload can be compressed to around 90 % of the original file size using gzip with default settings. We tested this, because some of the variants we test use a compression feature. This can either improve the performance, if the time to compress and decompress is lower than the time that can be saved by sending only around 90 % of the data packages, or this can lead to a decrease of the performance, if the additional time needed for the compression and decompression overhead is higher than the time that is saved. A real-world use case that could be compared to this scenario is the download of a large file from a server using a secure connection. However, usually the network is the factor in such scenarios that slows down the transmission, so we have to assume a use case where the network capacity is high enough to transmit all data a server can provide and a client can receive the data fast enough.

Using in-memory benchmarks for cryptographic purposes is quite common, for example TrueCrypt uses in-memory benchmarks to compare the performance of encryption algorithms without paying attention to bottlenecks caused by hard disk or solid state drive performance.

7.7.2 Confounding Variables

Additionally, to the confounding variables we have already defined for the general benchmark setup in 6.6.3, mbedTLS needs to use a client server setup, which introduces a new confounding variable: The server-client scenario. We compile the server and the client with the same library, so both are based on the same variant and use the same set of features. Server and client are running on the same processor, but the trade-off that remains is that we do not know if the server or the client is the slower part.

Another confounding variable is cryptography itself. Many parts of the connection handshake are using random numbers and the derived keys are unpredictable. These may have influences on the performance of the connection. In asymmetric cryptography, which is used for exchanging the symmetric keys between server and client, for example the calculation time depends on the value of the key. For generating the key the random number generator is involved. Although we do not expect a measurable performance decrease, we use 30 iterations of the performance benchmark for each variant to mitigate these effects.

7.7.3 Benchmark Execution

The estimation (7.2) shows the expected maximum run-time of the benchmarks. The calculation is based on the maximum measured times of 32 single variants that have been chosen randomly. As the measurements are executed parallel on the cluster. Not all variants are expected to compile properly, for that the real execution time is considered to be below the time per node t_{node}.

Estimation of the measurements execution time:

$$\begin{aligned}
c_{nodes} &= 14 && (7.2a)\\
c_{iterations} &= 30 && (7.2b)\\
c_{variants} &= 19200 && (7.2c)\\
t_{iterations} &= c_{iterations} * 60s = 1800s && (7.2d)\\
t_{compile} &= 120s && (7.2e)\\
t_{complete} &= c_{variants} * (t_{compile} + t_{iterations}) \approx 34,560,120s && (7.2f)\\
t_{node} &= t_{complete}/c_{nodes} \approx 2,468,580s \approx 4.1w && (7.2g)\\
& && (7.2h)
\end{aligned}$$

To produce stable and reliable results we use 30 iterations for the performance benchmark.

The execution of all measurements took about three days and 1,921 of 19,200 variants produced measurements, the other variants produced compilation or runtime errors. In conclusion of the result we refined the feature model by excluding these variants that produced no measurements. The main reason for compilation errors where type errors resulting from configurations that seem not to be testes by the developers of mbedTLS. Beside compilation errors we had uncommon[21] cipher suites that produced runtime errors, such as segmentation faults and TLS errors caused by invalid decrypted data blocks.

7.7.4 Measurements Data

We collected for all 1,931 working variants measurement data of compilation times, binary footprints of the executable, energy consumption while compiling and energy consumption while running the performance benchmark, the overall execution time of the performance benchmark and the execution times of every iteration of the performance benchmark. Furthermore, we collected the maximum main memory consumption in an extra execution of the benchmark.

7.7.5 Measurements Data Analysis Results

The measurements of the properties main memory consumption and energy consumption were not reliable and not reproduceable. The measurements of the energy consumption had a measurement bias caused by various network latencies we could not control. We found out that the main memory measurements suffer from TIME that is only sampling the maximum memory consumption. We tested this using 30 iterations for memory measurements for a single variant. Therefore, we did not further evaluate the data.

We did not further analyse the collected compilation times.

The measurements of the binary file sizes are fully reproduceable and deterministic.

In the performance benchmark fastest iteration of the fastest variant needed 6.72 s to complete, the slowest iteration of the slowest variant took 73.45 s.

We calculated the descriptive standard derivations and variances for all variants using the programming language *R*. The median standard derivation over all variants was 0.094 s and the median variance 0.008 s^2. Compared to the measurement differences between different variants this value is very low.

Figure 7.5 shows the violin plots of two randomly picked variants *v1* and *v2*. We found out, that the violins of the many variants showed outliers on the top, the lower performance. We took this as reason to investigate the cause of this outliers and used our reporting database to join the data of the measurement values with the data from the logs. Measurement values in interquartile ranges had a CPU utilization of 100 %. Outliers were strongly related to a utilization of 99 %. This is an indicator that the CPU utilization may be the cause of the outliers. For a further analysis detailed measurements of the CPU utilization with a higher accuracy would be needed.

[21] A list of common used cipher suites, for example defined by RFCs, can be found on the page: `http://www.thesprawl.org/research/tls-and-ssl-cipher-suites/`.

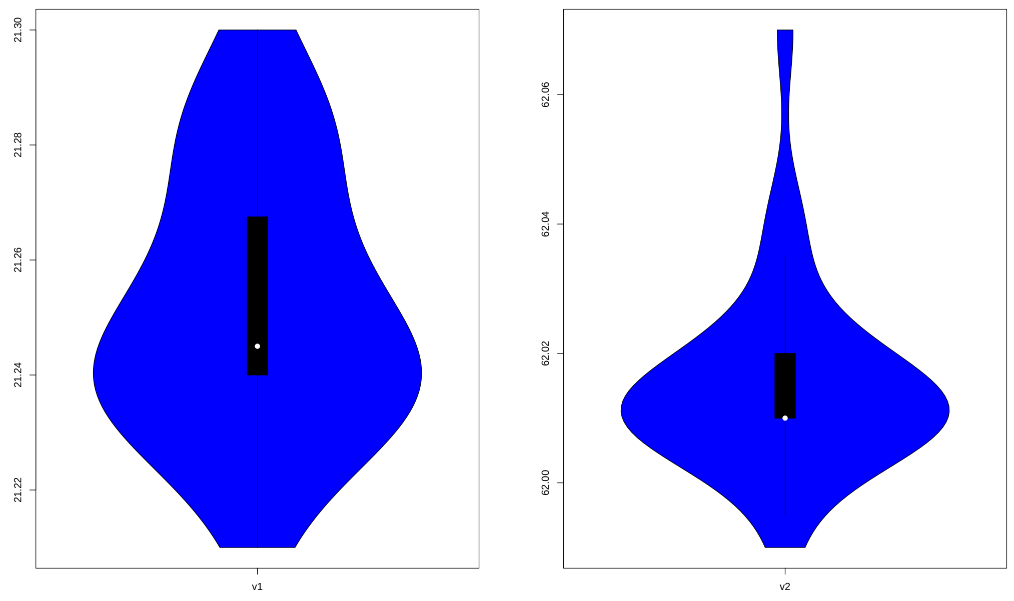


Figure 7.5: Violin plots for measurements of two different variants of mbedTLS. The y-axis represents the execution time of a iteration of the performance benchmark in seconds.

7.8 Measurements Conclusion

The measurements of the main memory usage and consumption have been unreliable and non-deterministic and are not usable for further evaluations.

The measurements of the binary file sizes and the measurements of the performance benchmark have been produced stable results that can be further processed. We imported the data successfully in SPLConqueror and started an analyses to detected feature interactions, to test if our measurement data is suitable for detecting interactions. We found out that the results of mbedTLS can be used to detect external feature interactions, but we did not further evaluate the results of the analysis as this is not part of the thesis. We provide our measurement data in files using the comma separated value format.

7.9 Summary

For mbedTLS we were able to construct a modular and extensible feature model based on the different groups of algorithms. Extensions beside these groups, for example including the X.509 certification infrastructure, would introduce much more dependencies and complexity. We prepared the source code and feature model for analysis by TypeChef that can be used to extract all feature interactions. For the measurement of external feature interactions focused on performance interactions, we have created a client server benchmark and executed the performance measurements and presented the results.

8 SQLite

This chapter describes the experimental setup and execution to create variability-aware call graphs and to measure non-functional properties in order to find internal and external feature relations in the subject system SQLite.

8.1 Description of SQLite

SQLite is “an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.”[36]. Being an embedded database library SQLite covers another application domain which is a real-life use case. It has been analysed using TypeChef, in the dissertation of Liebig “Analysis and Transformation of Configurable Systems” [14], and in the context of SPLConqueror in the paper of Siegmund et. al “Predicting Performance via Automated Feature-interaction Detection” [15]. Contrary to our work Siegmund used a sample of 100 variants, while we aim to benchmark all variants that can be generated by a feature model. Moreover, we use the same feature model for TypeChef and SPLConquerorand a newer version of SQLite, so we could not re-use their work. Like mbedTLS SQLite is not only limited to the embedded sector it is widely used by applications of multiple domains, for example for it is used by Skype as internal database. The reporting module of the thesis uses SQLite as data storage.

Compared to other common database management systems there are no performance benchmarks publicly available that satisfy the requirements of this study. With the shell interface of SQLite it is possible to generate a performance benchmark using publicly available data sets.

8.2 Feature Model

This section describes the configuration options of the SQLite library, the selection of a sub set of configuration options and how these options where combined to form a feature model suitable for the measurements of external feature interactions.

8.2.1 SQLite Configuration Options

SQLite has a well documented list of compile time configuration options[22]. Currently there are about 165 macro definitions that can be considered to represent a feature. Nearly all

[22] https://www.sqlite.org/compile.html

macros that can be used for configuring SQLite are prefixed with *SQLITE_* while platform dependent macros are prefixed with *HAVE_*.

As the documentation states the development team of SQLite spends a lot of effort that every configuration option should lead to a working variant of the library. Moreover, it is clearly said that they cannot guarantee or test every possible combination and only a listed set of configuration options is fully tested to be compatible in every combination [37].

According to the documentation we classify the features into binary features which are enabling or disabling a part of the source code and numeric features. In contrast to mbedTLS most of the features do not depend on the presence or absence of other features. In general this makes the feature model simpler and much easier to handle.

Most of the features do not need an additional runtime configuration in order to be called while the binary is executed. They will be used if they are present or their usage depends on the executed queries. For example if *SQLITE_DEBUG* is enabled the debug code will be called without the need for any additional configuration at runtime or if *SQLITE_LIKE_DOESNT_MATCH_BLOBS* is enabled queries using a *LIKE* expression in a SQL statement will be threaded differently.

SQLite uses macro definitions as negative feature expressions, using the prefix *SQLITE_OMIT_**, to omit parts of the source code. For example the presence of the macro *SQLITE_OMIT_SHARED_CACHE* causes the shared cache feature to be omitted. For the case study all features that can be omitted are assumed to belong to the core functionality and will not be omitted by default. The decision was made because the benchmark should basically be able to execute an arbitrary SQL query that is supported by the language specification of SQLite[23]. In all real-world use cases of SQLite we came across we saw always the full SQLite language standard to be supported.

For embedded systems there would be the possibility to minimize the binary and memory footprint by omitting features that are not relevant for the use case. For example if there is no usage for the *CAST* operator in the wrapping software the library does not need to support the operator and *SQLITE_OMIT_CAST* can be enabled. Configuration options that are not prefixed with *SQLITE_OMIT_* can affect the size of the binary.

8.2.2 Handling Numeric Configuration Options

Beside the macros that are using *#ifdef*-variability some configuration options of SQLite use numeric options. Numeric options currently cannot be analysed by TypeChef. A numeric option does not disable parts of the code, it is evaluated at runtime by the code. This can be used as boolean flags, for implementing different operation modes of a feature or for configuring numeric properties, such as cache sizes. For example *SQLITE_THREADSAFE* supports the values 0 for no thread safety, 1 for the highest level of thread safety (serializing) and 2 for multi-threading support with restrictions. This can be modified to a binary feature

[23] https://www.sqlite.org/lang.html

if level 2 is omitted. Other numeric options define limits, sizes or other enumerations. We used the default values for these options as they were not part of our feature model.

However, there are also numeric options that are only supporting 0 and 1 as valid values, e.g. *SQLITE_DEFAULT_MEMSTATUS*. This kind of configuration option is likely to produce another call graph but has no effect on the size of the binary. They are implemented as numeric options because they are passed directly to C-structs and used as booleans as shown in listing 9. We can replace it by an *#ifdef*-macro and *#ifndef*-macro that are wrapping the actual value to use as boolean like demonstrated in listing 10. The default value from listing 11 has to be removed from the source code as the option may be undefined if the feature is disabled.

```
/*
** The following singleton contains the global configuration for
** the SQLite library.
*/
SQLITE_PRIVATE SQLITE_WSD struct Sqlite3Config sqlite3Config = {
   SQLITE_DEFAULT_MEMSTATUS,  /* bMemstat */
   // [...]
};
```

Listing 9: Original code using the configuration option in the C-struct as boolean.

```
SQLITE_PRIVATE SQLITE_WSD struct Sqlite3Config sqlite3Config = {
   #ifdef SQLITE_DEFAULT_MEMSTATUS
        1
    #endif
    #ifndef SQLITE_DEFAULT_MEMSTATUS
        0
    #endif
    ,  /* bMemstat */
   // ...
};
```

Listing 10: Rewritten code using *#ifdef*-variability to switch the boolean values.

```
#if !defined(SQLITE_DEFAULT_MEMSTATUS)
#    define SQLITE_DEFAULT_MEMSTATUS 1
#endif
```

Listing 11: Definition of default value for undefined configuration option.

8.2.3 Selection of Features and Creation of a Feature Model

It is infeasible to benchmark all feature combinations, so we focused on those features that may have an influence on performance and may have performance interactions. Compared to mbedTLS SQLite does not offer a wide range of algorithms with different performance characteristic that can be varied. Also there is no wrapper concept like a cipher suite that requires minimum one feature of a feature category to be present in order to form a valid suite. Basically a free selection of options in the list of available configuration options can be performed to get enough features for a suitable number of variants.

We studied the documentation of SQLite to as features that could have a measurable impact on the performance according to the documentation have been flagged. The development team lists configuration options to be tested in order to be combinable with each other. These options have been given a higher priority in order to avoid compile time or run time errors as far as possible. Also it has been taken into account that a feature should be called in the benchmark suite.

In order to allow the benchmarks to run with multiple iteration with a suitable runtime in combination and a practicable amount of time we limited the number of variants to 5,000. On the other hand minimum 1,000 variants are wanted for later evaluations of the generated data. Both numbers have been estimated by performing test measurements. Twelve configuration options have been identified to be enough to generate these variants.

For the feature model there are twelve independent and optional binary features, which results in $2^{12} = 4096$ variants.

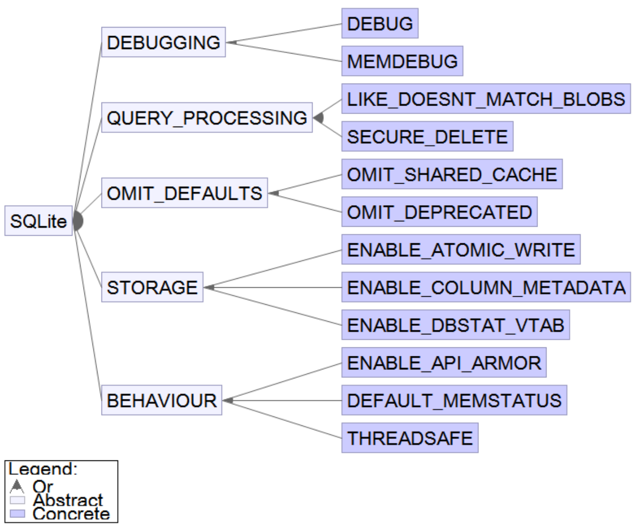


Figure 8.1: SQLite Feature Model.

Figure 8.1 shows the features in a visualized feature model. To increase the readability of the model and to clarify the semantics the features are grouped in the model and the *SQLITE_* prefixes of the features are removed. The parent features are not part of the code and only for structuring purposes.

This model is used as input for SPLConqueror which is able to generate all possible variants that are valid for the model presented above. The output variants of SPLConqueror are used to generate the C code that is combined with the template of SQLite's *config.h* in order to generate the input configuration for the compiler.

8.3 Internal Feature Interactions

To detect the internal feature interactions we converted the feature model from the meta format, described in 4.2.2, to a Feature Expression model suitable for TypeChef. Beside the Feature Expression model we created an *open-features.txt* file that contains all configuration options. We executed the build process using the tool make that creates the two agglomeration[24] source code files *sqlite3.h* and *sqlite3.c* from the original source code. These files can compiled to the library. For all features that are part of the feature model, we removed the default values of the configuration options, otherwise TypeChef would use this values instead of performing a variability-aware analysis of the configuration option. Moreover, we modified the source code by rewriting the numeric configuration options to use *#ifdef*-variability, as described in 8.2.2.

Using this setup TypeChef can be used to generate variability-aware call graphs to detect internal feature interactions, as described in 5.2. We tested the generation of these graphs, but the actual analysis of the graphs and the extraction of the feature interactions is part of further work.

8.4 External Feature Interactions

This section discusses the benchmarks for SQLite. We present the execution of the measurements, the measured data, discuss the quality of the results and show examples for external feature interactions.

8.4.1 Availability of Benchmarks for SQLite

There are multiple de facto standard benchmark suites available for databases for example TPC-C [38] using a client-server paradigm. These benchmarks cannot be used for SQLite because SQLite is a in-process library and therefore linked against the using software. In order to setup a performance benchmark for this scenario the library has to be part of the benchmark software. Contrary to mbedTLS the library can be benchmarked without the use of a client server paradigm. Beside using a universal database benchmark specialized benchmarks for SQLite are available. The SQLite team itself offers a documented performance benchmark that is outdated and not usable with SQLite 3 [39]. Specialized or not publicly available papers and articles describing benchmarks are excluded, for example

[24]https://sqlite.org/howtocompile.html

Moriki Yamamoto's and Hisao Koizumi's artcile "An Experimental Evaluation using SQLite for Real-Time Stream Processing" only covers the field of stream processing [40]. After working through 30 articles, papers and performance benchmark setups no benchmark was found that is compatible with the current SQLite version and is suitable for the use case of the case study.

8.4.2 Performance Benchmark Design

As none of the available benchmark is suitable for the currently used SQLite version a basic benchmark suite has been created with keeping in mind that benchmarking a database is not trivial. SQLite includes a command line interface (CLI) [41] which is capable of accessing database files and performing various operations on the database. The CLI offers the ability to execute queries stored in files and use this to process large amounts of data. This makes it possible to use the CLI for executing a benchmark consisting of arbitrary queries. Basically most benchmark suites are made up of a set of queries that are executed on the database.

The SQLite Project page itself lists a wide range of different use cases covering small embedded databases as well as websites with up to 100,000 hits a day [42]. So this can not be considered a typical use case for SQLite and no representative benchmark for all use cases. As the main goal of the Thesis is to acquire data to find correlations between internal and external interactions of features the benchmark should provide data that may show the impact of the presence of different features. The outdated benchmark of the SQLite team gives an impression how a benchmark setup can be structured [39]. In the following not all parts of the benchmark are reproduced, but the basic kinds of queries are included.

As data source, we do not use random generated data for the current benchmark, because there is no benefit over using a real world data set and furthermore the seeding of realistic data is not trivial. To get measurements with a low standard derivation and a small variance the benchmark should run for multiple seconds. An embedded use case is not suitable for this because the small amounts of data implicate shorter run times even for thousands of queries to be executed. This favours a larger data set of some hundred MB.

One large data set that is available under public domain and used science is the database of the Internet Move Database (IMDB)[25]. The set is used in publications like "Predicting IMDB Movie Ratings Using Social Media" [43] and "A Dataset Search Engine for the Research Document Corpus" [44], but not in the context of benchmarking. The missing schema information and the undocumented text format of the files make the set hard to convert into database inserts.

Another large data set that can be used for research under creative commons license is provided by Stack Exchange, Inc. The data set covers "an anonymized dump of all user-contributed content on the Stack Exchange network" [45]. It is stored in well defined XML and therefore good to process, filter and transform into SQL insert statements. The set has been used in other scientific publications across different contexts, for example in Anderson's

[25] http://www.imdb.com/interfaces

case study on knowledge discovery "Discovering value from community activity on focused question answering sites"[46], and in Kartik's data mining paper "Mining Questions Asked by Web Developers"[47].

The data set of Stack Exchange of the English[26] Exchange is used. The XML has been processed in order to create a schema and INSERT-statements. Statistics about the processed data set are shown in the table listing 12. The foreign key relations have been created by using the semantics of the field names defined in the XML files, for example an attribute USERID was interpreted as foreign key relation referencing the primary key of the table USERS.

table	data rows	insert query size	insert query files	indexes	keys
badges	301,676	22.3 MB	1	2	1
comments	569,222	145.0 MB	12	1	2
posthistory	663,503	403.1 MB	15	0	2
postlinks	37,906	2.3 MB	1	2	2
posts	271,229	240.7 MB	6	1	0
tags	922	0.0 MB	1	0	1
users	138,924	7.6 MB	1	1	0
votes	1,400,067	77.4 MB	3	0	2
sum	**3,383,449**	**898.4 MB**	**40**	**7**	**10**

Listing 12: Stack Exchange data set statistics.

The UML diagram in figure 8.2 shows the table structure of the benchmark data. The schema is reconstructed from XML files provided by Stack Exchange without any changes to the semantics of the schema, but columns have been removed to reduce the overall size of the data. No aggregation, composition and multiplicities are used in the diagram, only the foreign key constraints are expressed. The data types are very generic, for example CREATIONDATE columns are stored as TEXT. SQLITE only supports a limited set of five data types (NULL, INTEGER, REAL, TEXT, BLOB) for storage [48].

The CLI for SQLITE produced on buffer overflows both test platforms (Windows and Linux) and encoding errors if the filesize exceeded around 64 MB or contained Unicode characters. To work around both issues the files containing the inserts for the data we have decided to split the files into chunks smaller than 64 MB. Moreover, all Unicode characters have been replaced by underscores as their presence is not essential for the benchmark.

The benchmark was made up of six phases and covers the most common use cases. We used the outdated SQLite benchmarks as orientation.

CREATE In the first phase we run queries that are creating all tables. For the overall benchmark the process has no performance relevance as it happens almost instantly, but the step is needed to create the schema for the inserts.

[26] https://english.stackexchange.com

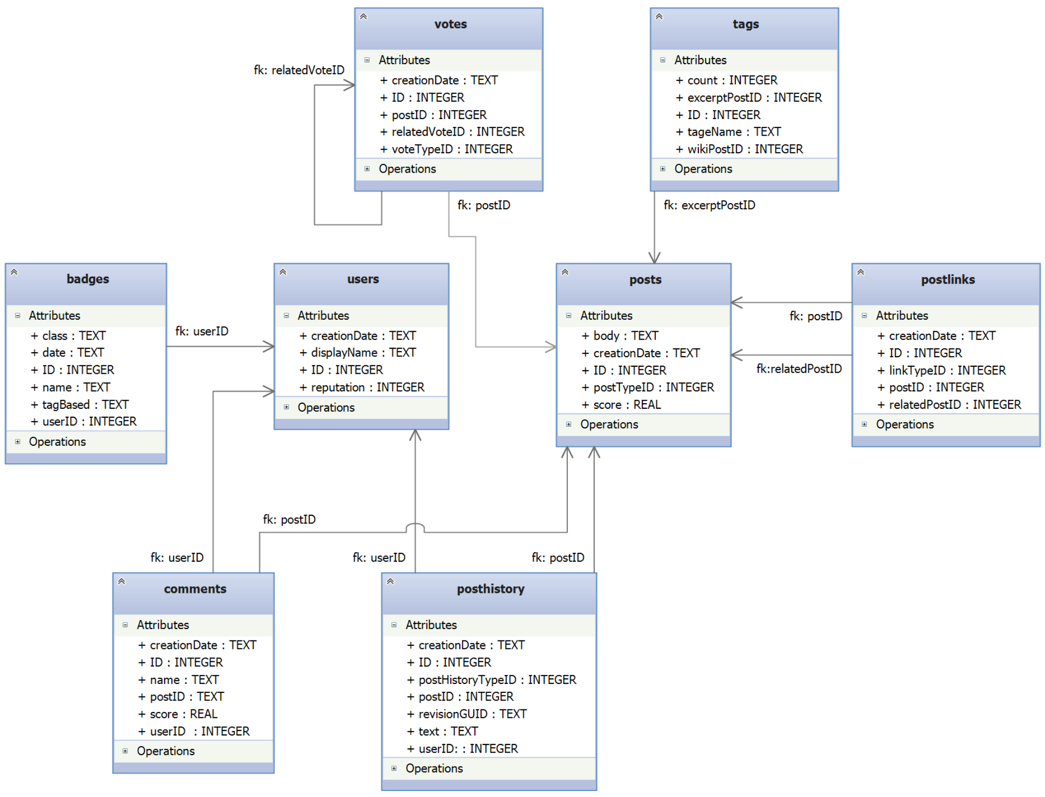


Figure 8.2: UML model of the benchmark's table structure.

`INSERT` In the second phase data will be inserted using inserts with multiple values. Every file exactly contains one insert statement. Using more inserts per file causes frequent index updates and frequent writes to the database file which result in a decrease in performance. As the insert phase should not dominate the overall benchmark and also a great amount of data is required to measure the performance of the reading queries, we decided to reduce the inserts by using multiple values for each insert.

`SELECT` The third phase runs a set of eleven different non trivial reading query types on the database. Each query type is executed once. The queries cover the most present language features as joins, projections, aggregation, sorting, numeric range selections, grouping, wildcard searches using a Like clause and correlating inner queries. Four sample queries are shown and described in listing 13.

`UPDATE` Update queries are executed with Where clauses on multiple rows and different tables. One Update query uses an inner query for selecting the new value from the row before.

`DELETE` After updating the data Delete queries are executed and all tables will be empty after the execution of the queries. This has the same effect as a Truncate query, but the Delete queries also include Where clauses with conditions of different complexity.

DROP Finally a drop statement that resets the schema is executed for all empty tables.

As the benchmark is very structured and does not simulate the behaviour of most real world examples where Insert, Select, Update and Delete statements are in no particular order and their shares are heavily depending on the purpose of the software that uses SQLite. Nevertheless, the suite ensures that most features that are present in the build are covered by the benchmark, but we do not expect every code path to be covered. For example the feature SQLITE_LIKE_DOESNT_MATCH_BLOBS is not present in half of the generated variants and the benchmark includes a query with a Like clause that consumes a measurable amount of time. We expect these variants to show different performance characteristics as the option should improve the performance according to the documentation [37].

```sql
-- q1:   projection of u.id
--       aggregation of u.id using the count function
--       full joins between users, posts and posthistory on primaries
--       selection by u.id and grouping by u.id
SELECT   u.id, COUNT(u.id)
FROM     users u, posts p, posthistory ph
WHERE    ph.postID = p.id AND ph.userID = u.id
AND      u.id < 1000
GROUP BY u.id;

-- q2:   projection of c.score
--       aggregation of c.id using the count function
--       grouping by c.score
--       selection with an inner query with and aggregation function
SELECT   c.score, COUNT(c.id)
FROM     comments c
GROUP BY c.score
HAVING   c.score = (SELECT MAX(c.score) FROM comments c);

-- q3:   projection of pl.id and pl.creationDate
--       selection by date comparison and ordering by date
SELECT   pl.id, pl.creationDate
FROM     postlinks pl
WHERE    pl.creationDate < '2016-06-01'
ORDER BY pl.creationDate ASC;

-- q4: selection with a like clause and a wildcard expression on c.text
SELECT * FROM comments c WHERE c.text LIKE '%english%';
```

Listing 13: Example for four reading queries of the benchmark.

The performance impact on queries that are passed directly to the SQLite CLI, resulting by the limitations of the Standard-IO, caused us to store the queries in files. We use the ability of the CLI to pass references to SQL files that are read by using the file IO subsystem. To eliminate this biasing influence on the measurements makes sense as in real-world systems the standard IO is not used by SQLite and queries are usually passed directly from the host application to the library.

8.4.3 Confounding Variables

An additional confounding variable in the context of SQLITE is non-deterministic behaviour in the benchmark. We can take the following query as example for an unreproducible performance "SELECT * FROM TABLE ORDER BY RANDOM() LIMIT 1". The "ORDER BY RANDOM()" clause of the query makes it unreliable how long the query needs to terminate. We decided to avoid all constructs in our queries that cause unreproducible query executions.

8.4.4 Benchmark Execution

The estimation (8.1) showed the expected maximum run-time of the benchmarks. The calculation is based on the maximum measured times of 32 single variants that have been chosen randomly. As the measurements are executed parallel on the cluster. Not all variants are expected to compile properly, for that the real execution time is considered to be below the time per node t_{node}.

Estimation of the measurements execution time:

$$
\begin{aligned}
c_{nodes} &= 14 && (8.1a)\\
c_{iterations} &= 50 && (8.1b)\\
c_{variants} &= 4096 && (8.1c)\\
t_{iterations} &= c_{iterations} * 40s = 2000s && (8.1d)\\
t_{compile} &= 50s && (8.1e)\\
t_{complete} &= c_{variants} * (t_{compile} + t_{iterations}) \approx 8,400,000s && (8.1f)\\
t_{node} &= t_{complete}/c_{nodes} \approx 600,000s \approx 6.9d && (8.1g)\\
& && (8.1h)
\end{aligned}
$$

To produce stable and reliable results we decided to use a high number of 50 iterations for the performance benchmark.

The execution of all measurements took about three days and 1,538 of 4,096 variants produced measurements, the others did not compile successfully. In conclusion of the resulted we refined the feature model by removing one feature and excluded the variants not compiling because of type errors from the feature model. One reason for the failure of 2,048 variants was a feature that was working well in our testing environments, but broke on the productive run because of an incompatibility between our SQLITE version and the compiler version.

8.4.5 Measurements Data

We collected for all 1,538 working variants measurement data of compilation times, binary footprints of the executable, energy consumption while compiling and energy consumption while running the performance benchmark, the overall execution time of the performance benchmark and the execution times of every iteration of the performance benchmark. Furthermore, we collected the maximum main memory consumption in an extra execution of the benchmark.

8.4.6 Measurements Data Analysis Results

The measurements of the properties main memory consumption and energy consumption are not usable for the same reasons described in 7.7.5.

We did not further analyse the collected compilation times.

The measurements of the binary file sizes are fully reproduceable and deterministic.

In the performance benchmark fastest iteration of the fastest variant needed 26.65 s to complete, the slowest iteration of the slowest variant took 44.77 s.

We calculated the descriptive standard derivations and variances for all variants using the programming language *R*. The median standard derivation over all variants was 0.093 s and the median variance 0.010 s^2. Compared to the measurement differences between different variants this value is very low.

Figure 8.3 shows the violin plots of two randomly picked variants *v1* and *v2*. We suggest the same reason for outliers as described in 7.7.5.

8.5 Measurements Conclusion

The measurements of the main memory usage and consumption have been unreliable and non-deterministic and are not usable for further evaluations.

The measurements of the binary file sizes and the measurements of the performance benchmark have been produced stable results that can be further processed. We imported the data successfully in SPLConqueror and started an analyses to detected feature interactions, to test if our measurement data is suitable for detecting interactions. We found out that the results of SQLite can be used to detect external feature interactions, but we did not further evaluate the results of the analysis as this is not part of the thesis. We provide our measurement data in files using the comma separated value format.

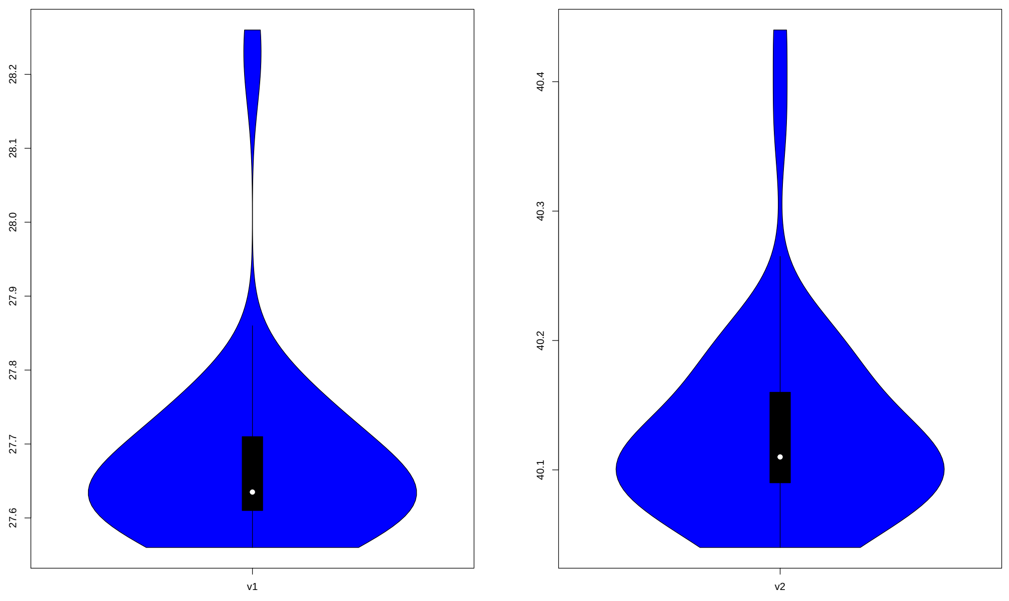


Figure 8.3: Violin plots for measurements of two different variants of SQLite. The y-axis represents the execution time of a iteration of the performance benchmark in seconds.

8.6 Summary

For SQLite we were able to construct a very modular and extensible feature model because of the low number of dependencies. Based on this feature model we have set up a TypeChef project and prepared the generation of the variability-aware call graphs. For the measurement of external feature interactions focused on performance interactions, we have created an extensible benchmark and executed the performance measurements. We showed that the results of the measurements allows a further processing of the data. Due to time constraints we have not finished the analysis of the analysis of SPLConqueror, yet.

9 Validity and Threats to Validity

In this chapter we discuss the validity and the threats to the validity of the case study. The awareness of the threats to validity is one of the most important topics in empirical science according to Siegmund et. al.: “Reviewing papers that were recently published in major software-engineering venues, we found that 91 % presented an empirical study, but only 54 % discussed threats to validity, and only 23 % differentiated between different kinds of validity. Given that we include EMSE as major empirical software-engineering journal, this is an alarmingly high number of authors who do not seem to be aware of the threats to validity to their study” [49].

9.1 Types of Validity

There are multiple types of validity that can be discussed, such as construct validity or statistical conclusion validity. In the following we will focus on external validity and internal validity, because they are an important aspect for our case study.

According to Carter and Porter “internal validity relates to the extent to which the design of a research study is a good test of the hypothesis or is appropriate for the research question” [50]. This means observable changes in dependent variables should be caused by changes in the independent variables. Therefore confounding variables and measurement bias are the biggest threats.

Caleder et. al. define that “external validity examines whether or not an observed causal relationship should be generalized to and across different measures: persons, settings and times” [51]. In our case this means that external validity is a factor of representativeness we need to archive in order to be able to transfer our results to other highly configurable subject systems. Moreover, it is generally assumed that a high external validity is a crucial factor in empirical research: “Research that is weak in external validity is not considered to provide an adequate test of theory” [52, 51].

9.2 Preliminary Study and Selection of Subject Systems

Internal Validity The internal validity of the evaluation of subject systems is threatened by the fact that we have no approved state of the art technology available that enables us to find out if a software system is a highly configurable systems which satisfies the needs of our analysis methods. We had to use a heuristic in combination with a manual selection process as described in chapter 3. The heuristic may have overseen a system, as well as we could make errors during the manual selection process. Still, we selected far more candidate systems as we could process. All selected systems fitted our requirements.

External Validity The preliminary study evaluated over 250 candidate subject systems. The candidates come from various common application domains to increase the external validity. However, our capacity for subject systems we could analyse in-depth was very limited through the complex process for the measurements of external feature interactions. This caused us to select two subject systems of two different domains and with different feature model characteristic, as we discussed in 8.2 and 7.4. The validity of the selection is threatened by the fact that both systems can be seen in a similar domain, namely in the domain of embedded software. Both systems are optimized to run in environments with very limited resources and may not be representative for systems that have no optimizations of this kind.

9.3 Generation of Feature Models

Internal Validity A threat to the internal validity are errors in the model that result from undocumented constraints and errors in the documentation or source code that result in invalid variants.

The internal validity of the generation of the feature models was increased by using a strict validation of the generated variants. For example, we conduct a extra test using a functional program to verify if the variants generated by the program match the variants generated by SPLConqueror, as described in 7.4.

External Validity The external validity is threatened by feature models that are not representative for highly configurable system. We found out that the feature models we can construct are very dependent on the point of view. For example mbedTLS can be seen as library for TLS-communication or as modular set of packages for cryptographic purposes. The structure of the code enables both kinds of usages. We decided to see it as library for TLS-communication and built the feature model for that purpose. This should not harm the external validity in this case as this is the most common real-world application scenario we found for mbedTLS. Another threat are the restrictions we had to add for enabling useful benchmarks. These restrictions are not present in most real-world use cases. They increase the internal validity of the experiment, but limit the external validity because it is not possible to generalize our models for arbitrary use cases of our subject systems.

9.4 Detecting Internal Feature Interactions

For our analysis of internal feature interactions we used state of the art analysis techniques implemented in TypeChef. We did not need to modify the original TypeChef analysis. Therefore we can assume that the internal validity of our results is very high.

The internal validity can be threatened as we only used variability-aware control flow graphs as analysis method. Features can interact on data flow level, too, but we did not consider this kinds of interactions. We found out that mbedTLS uses function pointers to implement a kind of module interfaces, as we described in section 7.2. Calls using this function pointers cannot be detected by TypeChefusing the control flow analysis techniques.

9.5 Detecting External Feature Interactions

Undiscovered sources for measurement bias threaten the internal validity of measuring non-functional properties and therefore, detecting external feature interactions. We used randomization and a high number of iterations, as well as logging as many environment parameters as possible to control measurement bias. The analysis of the results revealed no evidence for unwanted measurement bias so far. For example, we had stable and reproduceable measurements for performance benchmarks.

We defined the dependent, independent and confounding variables carefully and mitigated the effect of the confounding variables or controlled them. We discussed this in 6.6.3 in detail.

Another general threat is the accuracy and the precision of the measurements. The evaluation of the results of both subject systems have proven a good quality of the measurements of performance. Energy and memory consumptions have not been used as we found out that the measurements are not reliable enough.

The internal validity is threatened by the assumptions we have made and the limitations of the measurements. Because of the many parameters that are present in the setup it is not feasible to measure all variants. We had to stick, for example, to the compiler's default optimization levels for the subject systems and we only had two test clusters available for our measurements using the same operating system. To mitigate the threat, we used real-world parameters or validated single parameters against other test environments as far as possible. For example, we benchmarked randomly picked variants on a Windows operating system and validated that the sample showed the same measurement characteristics as on the main benchmark system running Linux.

We used only two subject systems and can therefore not generalize.

9.6 Subject System: mbedTLS

The internal validity of the measurements is considered to be very high as we used a careful and long-term engineered benchmark setup and have been aware of many common sources of measurement bias. The measurement results have been reasonable and stable showing a low standard derivation and variance. There is a remaining risk that we have still undiscovered sources of measurement bias in our measurements.

The client server paradigm, that has to be used, is difficult to benchmark in general. We had identical variants of server and clients, assuming that we do not care which of the two is the limiting factor, as both are using identical feature sets. This could be a threat to the internal validity.

The external validity in the context of mbedTLS is threatened by the benchmark that cannot be generalized to represent an arbitrary use case of mbedTLSẆe see no danger in the general setup to benchmark connections as it reflects the common real-world use case of mbedTLS, but because of the long runtimes we had to decide for a single size and type of the payload we used to benchmark the connection. This is the biggest drawback as there are many other scenarios mbedTLS can be used for, for example a high number of connections transferring only a small amount of data.

9.7 Subject System: SQLite

We consider the internal validity of the measurements as very high as we used a solid benchmark setup and have been aware of many sources of measurement bias. This produced stable and reasonable measurement results. There is a remaining risk that we have still undiscovered sources of measurement bias in our measurements.

The biggest threat to the external validity of the analysis of SQLite is the representativeness of the benchmark we have built. SQLite can be used in various use cases starting with a small embedded database on a mobile phone for managing a contacts database with only few entries up to the backend of a website with millions of entries and far more transactions. This makes it hard to represent all use cases in a benchmark; this made us focus on the quality of the measurements. To increase the external validity in the context of SQLite use cases we used a publicly available and commonly used data set that can represent a real-world use case, and we orientated our benchmark on the queries of older SQLite benchmarks used by the development team. However, we cannot generalize our benchmark to represent other use cases.

9.8 Overall Case Study

We tried to mitigate all threats to the internal validity by using either approved analysis techniques or using various validation methods for our measurements. The remaining risk, we cannot exclude completely, is the presence of measurement bias in the external feature interactions measurements. As we found external feature interactions using SPLConqueror and we were able to find them in the source code and in the variability-aware call graphs, too, we assume both measurements to be reasonable.

The external validity of the overall case study we conducted is very limit because the external validity of case studies is limited in general. Two subject systems are not enough to generalize findings.

All in all, we think that we have reached a level of internal validity that enables our work to continue with finding relations between internal and external feature interactions. We strongly encourage future work to refine our work with other subject systems and to target the limitations of our measurements.

10 Conclusion

This chapter discusses the challenges of working on the case study, we summarize our work, and we draw a conclusion from the results.

10.1 Challenges

The field of highly configurable software and feature orientated software is a relatively new field in research. Therefore we expect to face new and unforeseen problems which cannot be solved by applying well known best practices. Moreover, we came in touch with many different computer science disciplines, each requiring a deeper knowledge, some of them even requiring an in-depth domain knowledge.

For example, we needed awareness in empirical methods of computer science to design proper benchmarks and to avoid common mistakes and measurement bias in our benchmarks. We needed domain specific knowledge in transport layer security and cryptography to create working builds of mbedTLS and to face compiler time and runtime errors; we had to manually inspect and to debug the C code and build scripts and we had to create working SQL benchmarks.

Especially in our benchmarks we had to deal with long execution times required to find errors that have been affecting only some variants. We had to analyse the C code to find the real source of an error as we often had the problem of avalanche of errors.

In total, all of these factors could be solved but they led to an extensive amount of time that had to be spent on. This should be kept in mind while conducting further case studies of this kind.

Initially we thought of our subjects systems as being black boxes we can handle with existent benchmarks and analysis software so that we could focus on finding relations between the data sets. Actually the real-world systems showed up to be more complex to analyse than expected. Problems we came across have been errors in the documentation, missing constraints of features, poorly up-to-date documentations, outdated benchmarks, bugs in the source code resulting in not working variants. Most of these problems, such as outdated documentation files, could be classified as classical software engineering problems that have been a research topic for more than forty years. We faced this problems mainly by analysing the source code manually and adding the undocumented constraints to the feature models. If we could not detect additional constraints we excluded the variants from the feature model, but we did not edit the original source code of our subject systems to fix bugs or misbehaviour. These problems led to acquire in-depth domain specific knowledge of the subject system to understand the whole system and to adjust the benchmarks. The originally intended black box approach became a white box approach.

We learned that the availability of approved analysis tools and a documentation software project are not enough to run out of the box. A highly configurable and modular system still may suffer from unexpected feature interactions if it was not developed and tested excessively in respect of this development approach. Most software project use static test cases that cover the common and expected use cases, but none of the projects we came across used static variability-aware analysis tools or generic test cases to analyse or test uncommon variants. Both subject systems of our case study showed different characteristics in the aspects which needed manual reworking. MBEDTLS for example had undocumented constraints we needed to find by processing the source code manually. SQLITE offered a command line interface that was not capable of processing queries that contained unicode characters forcing us to develop a work-around. Such unforeseeable challenges lead to a lengthy and partially manual analysis process.

10.2 Summary of our Work

In the preliminary study we identified a list of open source software projects written in C using pre-processor directives for implementing variability as potential subject systems. We classified the projects in different domains and ran a heuristic to find subject systems that fulfil our requirements on the configuration options. In the next step we studied the documentation of the projects and analysed the source code and the configuration options manually. We selected two subject systems for an analysis in TYPECHEF and for measurements of external feature interactions.

We developed a text-based description format for feature models in order to create a maintainable source model for generating the target models of TYPECHEF and SPLCONQUEROR. We explored different approaches to reverse-engineering of feature models for MBEDTLS and SQLITE using the source code and the documentation, finally creating feature models that are suitable for measurements of external feature interactions, such as performance interactions.

For our analysis of internal feature interactions we used TYPECHEF to generate call graphs per source code file and to find type errors, based on the feature models of our subject systems.

MBEDTLS and SQLITEhad both no suitable performance benchmarks, so we used the compatibility test suite of MBEDTLS as base for a throughput benchmark, and for SQLITE we used the integrated CLI tool to execute queries on a large data set. In the next step we set up measurements for external feature interactions focusing on benchmarks for measuring performance interactions. Moreover we collected data about binary footprints and energy consumption. We provided an overview of the measured data and we discussed the internal and external validity of the measurements. Based on that analysis data we used SPLConqueror to identify the influence on performance of features interactions. For mbedTLS we verified the presence of these interactions by analysing the source code.

10.3 Conclusion, Contribution and Future Work

Our preliminary study produced a list of over 100 software projects that contains around ten potential subjects systems for future analysis. The heuristic tools can be reused for the analysis of other software projects.

We created a description format for feature models that can be used to maintain a single model and to keep the TypeChef and SPLConqueror models in synchronization. The software for the generation is available online.

For mbedTLS we reverse-engineered two feature models. The first model contains all configuration options we considered to be usable as freely configurable feature. This model can be used for further analysis that is not linked to performance measurements and therefore have no need to limit the size of the model. The second model follows a minimalistic and modular approach for modifying the number of variants that can be generated and is optimized for benchmarking TLS connections.

We created an execution and reporting framework for our subjects systems that is capable of executing performance benchmarks on a remote server infrastructure and can collect the data in the file system as well as in a remote database using an Rest API. The framework can be used for other subject systems with minor modifications.

As mbedTLS had no real publicly available performance benchmark for the measurement of connection throughput under controlled conditions we created a custom client-server benchmark that can be used for arbitrary feature models that produce valid variants. The payload can be modified and reused for analysing the impact of different payload sizes.

SQLite had, like mbedTLS no benchmark that fitted our requirements; this made us engineer a benchmark based on stack-exchange data. The benchmark can be used as base for further measurements and can be refined to cover different use cases of SQLite.

Finally we collected large data sets of internal and external feature interactions. We found internal feature interactions in variability-aware call graphs that can be post processed in future work and we found external feature interactions by analysing the measurement results with SPLConqueror.We showed that the measurements of the variants show significant performance differences and we stated the quality of the data. The data sets will be used in future work to study a possible relation between internal and external interactions.

11 Acknowledgements

11.1 Credits

I want to thank *Prof. Dr. Sven Apel* and *Prof. Christian Lengauer, Ph.D.* for supervising this thesis and I wand to thank the whole Chair of Software Engineering at the University of Passau letting me be part of their research projects. Especially I want to thank *Sergiy Kolesnikov* for his substantial support and constant guidance.

Furthermore I want to thank all current and former staff members who helped me with their suggestions and time, especially *Alexander Grebhahn*, *Jörg Liebig*, *Janet Siegmund* and *Norbert Siegmund*.

Finally I want to thank my wife *Claudia* for reading the whole Thesis and for her patience.

11.2 Case Study and Tool Availability

The project repositories of the subject systems are available online and hosted at GitHub. A request for reading permissions is currently required, as some papers build on top of the studies are not released, yet.

The repository of the mbedTLS subject system is available at:

https://github.com/DE120/mbedtls

The repository of the SQLite subject system is available at:

https://github.com/DE120/sqlite

Additional tools will be made available at:

https://github.com/DE120/ma-toolchain

Nomenclature

AES	Advanced Encryption Standard, a block cipher algorithm.
ARC4	Alleged RC4, a cryptographic stream cipher algorithm.
CBC	Cipher block chaining, a mode of operation for block ciphers.
CCM	Counter mode with CBC-MAC, a mode of operation for block ciphers.
CFB	Cipher feedback, a mode of operation for block ciphers.
CLI	Command Line Interface.
CTR	Counter mode, a mode of operation for block ciphers.
DES	Application Programming Interface, a block cipher algorithm.
DHE-PSK	Diffie–Hellman with pre-shared key, a key exchange protocol.
ECB	Electronic codebook, a mode of operation for block ciphers.
ECDHE-PSK	Elliptic curve Diffie–Hellman with pre-shared key exchange protocol.
GCM	Galois/Counter Mode, a mode of operation for block ciphers.
MAC	Message authentication code, a method to authenticate a message.
MD5	Message Digest 5, a hash function.
PSK	Pre-shared key, a key exchange mechanism.
SHA1	Secure Hash Algorithm, a hash function.
SHA256	Secure Hash Algorithm 2 with 256 bit output, a hash function.
SHA512	Secure Hash Algorithm 2 with 515 bit output, a hash function.
SPL	Software Product Line.
SSL	Secure Socket Layer, an encrypted network protocol.
TLS	Transport Layer Security, a newer version of SSL.
TPC-C	A database benchmark suite.

List of Figures

List of Listings

Bibliography

[1] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM. 1.1.1, 1.1.2

[2] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Publishing Company, Incorporated, 2013. 1.1.1

[3] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-condition simplification in highly configurable systems. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 178–188, Piscataway, NJ, USA, 2015. IEEE Press. 1.1.1

[4] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. Software Quality Journal, 20(3-4):487–517, September 2012. 1.1.1

[5] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: The new feature-interaction challenge. In Proceedings of the 5th International Workshop on Feature-Oriented Software Development, FOSD '13, pages 1–8, New York, NY, USA, 2013. ACM. 1.2

[6] 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, pages 284–294, New York, NY, USA, 2015. ACM. 1.3.1

[7] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In Proceedings of the 11th International Software Product Line Conference, SPLC '07, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society. 1.4

[8] Sergiy Kolesnikov, Judith Roth, and Sven Apel. On the relation between internal and external feature interactions in feature-oriented product lines: A case study. In Proceedings of the 6th International Workshop on Feature-Oriented Software Development, FOSD '14, pages 1–8, New York, NY, USA, 2014. ACM. 1.4, 5.1

[9] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented language families: A case study. In Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM. 1.4

[10] John Gerring. What is a case study and what is it good for? The American Political Science Review, 98(2):341–354, 5 2004. http://www.jstor.org/stable/4145316. 1.4

[11] Andrew Bennett. Models, Numbers, and Cases. Methods for Studying International Relations. University of Michigan Press, Michigan, 2004. DOI: 10.3998/mpub.11742. 1.4

[12] Bent Flyvbjerg. Five misunderstandings about case-study research. Qualitative Inquiry, pages 219–245, 2006. 1.4

[13] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse engineering feature models. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM. 2.1

[14] Jörg Liebig. Analysis and Transformation of Configurable Systems. PhD thesis, Universität Passau, 2015. 2.2, 8.1

[15] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012. IEEE Press. 2.2, 2.4, 3.1, 8.1

[16] Andreas Janaker. TypeChef meets SPL$^{\text{LIFT}}$ interprocedural data-flow analysis of configurable software systems. Master's thesis, University of Passau, December 2016. 2.2, 3.2, 5.1

[17] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. igen: Dynamic interaction inference for configurable software. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 655–665, New York, NY, USA, 2016. ACM. 2.3

[18] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 445–454, New York, NY, USA, 2010. ACM. 2.3

[19] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems, PLOS '11, pages 2:1–2:5, New York, NY, USA, 2011. ACM. 2.3

[20] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering, ISSRE '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society. 2.3

[21] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '15, pages 342–352, Washington, DC, USA, 2015. IEEE Computer Society. 2.4

[22] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski. Variability-aware performance prediction: A statistical learning approach. In 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 301–311, Nov 2013. 2.4

[23] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM. 2.4

[24] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. Performance prediction of configurable software systems by fourier learning (t). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '15, pages 365–373, Washington, DC, USA, 2015. IEEE Computer Society. 2.4

[25] Norbert Siegmund, Marko RosenmüLler, Christian KäStner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Inf. Softw. Technol., 55(3):491–507, March 2013. 3.1

[26] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. Software Quality Journal, 20(3-4):487–517, September 2012. 3.1

[27] Christian Kaestner. TypeChef project repository at GitHub., 2017. https://github.com/ckaestne/TypeChef. [Online; Accessed: 2017-02-28]. 3.2, 5.2

[28] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware refactoring in the wild. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 380–391, Piscataway, NJ, USA, 2015. IEEE Press. 3.4

[29] Department of Computer Science, Embedded System Software Group at TU Dortmund et. al. Project Prototypes, 2017. https://ess.cs.uni-dortmund.de/Research/Projects/FAME/prototype.shtml. [Online; Accessed: 2017-02-28]. 4.1

[30] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. Sci. Comput. Program., 79:70–85, January 2014. 4.2.1

[31] Tyler Sorensen. ABOUT PBL, 2017. http://formal.cs.utah.edu:8080/pbl/index.php. [Online; Accessed: 2017-02-28]. 4.2.2

[32] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking #ifdef variability in c. In Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM. 5.2

[33] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, and Gunter Saake. Measuring non-functional properties in software product line for product derivation. In 15th Asia-Pacific Software Engineering Conference (APSEC 2008), 3-5 December 2008, Beijing, China, pages 187–194, 2008. 6.1

[34] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. SIGPLAN Not., 42(10):57–76, October 2007. 6.1

[35] ARM Limited. High Level Design., 2017. https://tls.mbed.org/high-level-design. [Online; Accessed: 2017-02-28]. 7.1, 7.2, 11.2

[36] The SQLite Development Team. About SQLite, 2017. https://sqlite.org/about.html. [Online; Accessed: 2017-02-28]. 8.1

[37] The SQLite Development Team. Compile-time Options, 2017. https://www.sqlite.org/compile.html. [Online; Accessed: 2017-02-28]. 8.2.1, 8.4.2

[38] TPC. TPC-C, 2017. https://www.tpc.org/tpcc/. [Online; Accessed: 2017-02-28]. 8.4.1

[39] The SQLite Development Team. Database Speed Comparison, 2017. https://www.sqlite.org/speed.html. [Online; Accessed: 2017-02-28]. 8.4.1, 8.4.2

[40] Moriki Yamamoto and Hisao Koizumi. An experimental evaluation using sqlite for real-time stream processing. Journal of International Council on Electrical Engineering, 1(68):68–72, 3 2013. DOI: 10.5370/JICEE.2013.3.1.068. 8.4.1

[41] The SQLite Development Team. Command Line Shell For SQLite, 2017. https://www.sqlite.org/cli.html. [Online; Accessed: 2017-02-28]. 8.4.2

[42] The SQLite Development Team. Appropriate Uses For SQLite, 2017. https://www.sqlite.org/whentouse.html. [Online; Accessed: 2017-02-28]. 8.4.2

[43] Andrei Oghina, Mathias Breuss, Manos Tsagkias, and Maarten de Rijke. Predicting imdb movie ratings using social media. In Proceedings of the 34th European Conference on Advances in Information Retrieval, ECIR'12, pages 503–507, Berlin, Heidelberg, 2012. Springer-Verlag. 8.4.2

[44] Meiyu Lu, Srinivas Bangalore, Graham Cormode, Marios Hadjieleftheriou, and Divesh Srivastava. A dataset search engine for the research document corpus. In Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, ICDE '12, pages 1237–1240, Washington, DC, USA, 2012. IEEE Computer Society. 8.4.2

[45] Stack Exchange, Inc. Stack Exchange Data Dump., 2017. https://archive.org/details/stackexchange. [Online; Accessed: 2017-02-28]. 8.4.2

[46] Ashton Anderson, Daniel P. Huttenlocher, Jon M. Kleinberg, and Jure Leskovec. Discovering value from community activity on focused question answering sites: a case study of stack overflow. In The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012, pages 850–858, 2012. 8.4.2

[47] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Mining questions asked by web developers. In Proceedings of the Working Conference on Mining Software Repositories (MSR), pages 112–121. ACM, 2014. 8.4.2

[48] The SQLite Development Team. Datatypes In SQLite Version 3, 2017. https://www.sqlite.org/datatype3.html. [Online; Accessed: 2017-02-28]. 8.4.2

[49] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 9–19, Piscataway, NJ, USA, 2015. IEEE Press. 9

[50] Samuel Porter and Diana Carter. Validity and reliability. Wiley-Blackwell, 4th edition, 2000. Cited x8. 9.1

[51] John G. Lynch, Jr. On the external validity of experiments in consumer research. Journal of Consumer Research, 9(3):225, 1982. 9.1

[52] John G. Lynch. Theory and external validity. Journal of the Academy of Marketing Science, 27(3):367–376, 1999. 9.1

Eidesstattliche Erklärung

Ich, Alexander Denk, versichere hiermit, dass ich meine Masterarbeit mit dem Thema

> Detecting Control-Flow and Performance Interactions in Highly-Configurable Systems - A Case Study

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Passau, 31. März 2017

Alexander Denk