Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis

Automated Logging of Function Calls in Java, Python, and Go

Fabian Engler

Supervisor	Karel Kubíček
Co-Supervisors	Dr. Carlos Cotrini Jimenez Dr. Esfandiar Mohammadi
Professor	Prof. Dr. David Basin

ETH Zürich
Department of Computer Science
Information Security Group

October 15, 2019

Abstract

Logging is an essential practice in software engineering and provides runtime information about a system. In this thesis, we propose a method that automatically logs function calls for the three programming languages Java, Python, and Go. We study four different methods to achieve automated logging, namely monkey patching, aspect-oriented programming, interpreter modification, and program transformation. For each programming language, we implement the method that requires minimal changes in the system and can be integrated efficiently into an existing project. The automatically generated logs provide features for automatic processing and analysis. For example, they can be used to automatically check compliance in General Data Protection Regulation (GDPR) projects. We evaluate our implementation on an e-commerce application and measure the performance and storage of our methods. The experiments in Java and Python show that the CPU utilization does not increase more than twofold with automated logging. However, logging all function calls creates big data sets. In Java, for example, the storage increases by a factor of about 1300. Developers can reduce the log size by specifying what paths in the source code should be automatically enhanced with logging.

Acknowledgements

First of all, I would like to thank Professor David Basin and the Information Security Group for the opportunity to write this thesis. I enjoyed working in this creative and inspiring atmosphere, and to get an insight into different areas in computer science, including software engineering, information security, and machine learning. Furthermore, I would like to thank my supervisors, Karel Kubíček, Dr. Carlos Cotrini Jimenez, and Dr. Esfandiar Mohammadi, for their support and constructive feedback. In particular, I appreciated the weekly meetings and the discussions with my supervisors. Last but not least, I would like to thank my family and friends for their support.

Contents

Chapter 1

Introduction

1.1 Motivation

Logging is a common practice in software engineering and stores important runtime information about a system. The importance of logging can be identified by many use cases such as debugging, anomaly detection, performance diagnosis, and system behavior understanding. Studies of industrial systems show that manual logging is often incomplete and varies across projects. Especially with the growing complexity of software systems, it is difficult to find consistent logging behavior. We will examine this in Chapter 2.

Automated logging adds log statements automatically to the program. The log format is consistent and enables machine processing. In this paper, we study and implement automated logging in the detailed granularity of function calls.

Functions, as the name indicates, often describe the functionality of a program. Automated logging of function calls can provide a basis to automatically check compliance with the General Data Protection Regulation (GDPR). Automatic analysis of the logs can show how the system handles personal data and verify that it is processed compliant with GDPR.

The goal is to achieve automated logging of function calls with the least intrusive method. That means that the integration requires minimal changes in the system. To the best of our knowledge, there was no previous work that automatically logs function calls in the way we want to achieve it in this thesis.

1.2 Contribution

We propose a method that automatically logs function calls for the three programming languages Java, Python, and Go. These programming languages are translated in three different ways, compiled to bytecode, interpreted, and compiled to assembly code, respectively. For each programming language, we evaluate different methods for automated logging and choose the one with the lowest degree of intrusiveness, such that the integration requires minimal changes. Furthermore, we show that aspect-oriented programming is suited to implement automated logging. We also provide implementations. This thesis provides automatically generated fine-grained logs that can be used to apply machine learning techniques and analysis of runtime behavior,

in particular, to understand runtime behavior in GDPR compliance projects. For the implementations, we measure the effect of automated logging in terms of CPU performance and storage. The results show that the CPU does not increase more than twofold. To handle the generate log data, we provide different approaches.

Chapter 2

Related Work

2.1 Research

Zhu et al. [38] propose and implement a framework *LogAdvisor*, which helps developers to make informed logging decisions. The main idea of this framework is to learn common logging "rules" automatically. A machine learning model (e.g., decision tree) is trained to perform logging predictions. The goal is to predict if a code snippet should be logged or not. To train the model, LogAdvisor extracts useful features from logged code snippets. Feature extraction is the core of "learning to log" because the quality of the features determines the performance of the model. LogAdvisor proposes a feature extraction framework that extracts structural, textual, and syntactic features from a focused code snippet. Structural features contain exception types and the name of included and called methods. Textual features contain the names of variables and types. To better identify code snippets that require no logging, the framework also extracts syntactic key features, e.g., assignment statements with special values such as `null` or `false`. Fig. 3 in the paper shows an overview of the framework.

The authors evaluate LogAdvisor on two industrial systems with millions of C# code lines. Overall, the *balanced accuracy*, which is the average of the proportion of logged instances and the proportion of unlogged instances that are correctly classified, is high. The results ranging from 84.6% to 93.4%, indicating a high similarity to the manual logging decisions made by developers. The authors performed a user study, where 37 developers decided whether to log or not to log on code snippets with removed logging statements. The accuracy of correct log statement recovery was 60% for the group without logging suggestions, while the group with LogAdvisor suggestions achieved an accuracy of 75%.

The paper highlights the importance of logs for system behavior as we intend to achieve in our thesis. In particular, the paper explains feature extraction for machine learning methods. However, there are significant differences in our work. First of all, LogAdvisor only provides suggestions to developers where to log and does not construct any log statements automatically. Secondly, the evaluation reveals that developers log only about 8% of the methods and that LogAdvisor will not suggest to log most of the function calls. In contrast, we want to log all function calls. Thirdly, LogAdvisor does not extract information about methods as detailed as we aim to do with arguments and return value.

Fu et al. [15] provide an empirical study of the logging behavior of developers on two large industrial systems, each with millions of code lines. From these systems, the authors randomly sample 100 logging statements and divide them into five categories. The categories are assertion-check (19), return-value-check (14), try-catch (27), logic-branch (16), and uncategorized logging (24). A logged code snippet is a block of source code whose behavior the logging statement intends to log. For example, return-value-check code snippets check the return value of a function call for special values (e.g., `null`, `false`, `-1`). Further analysis of the source code shows that developers log only 30-42% of the catch blocks in exceptions and 8-9% of the function calls. Moreover, there are correlations between logged code snippets and specific keywords, e.g., "delete". Developers do not log the majority of catch blocks (58-70%) for three main reasons. First, passing logging to subsequent operations. Secondly, exceptions are recoverable. Thirdly, exceptions are not critical. In other words, the decision to log for a code snippet is often highly related to the semantics of the code snippet.

The authors further examine logging behavior via a questionnaire survey with 54 experienced developers. The questionnaire consists of two specific questions: what scope of source code and what factors do developers mostly consider on the decision whether to log? For the source code scope, the participants consider the function containing the exception (69%) and the corresponding try block (61%) as the most important information. 57% of the participants consider the exception type and 46% the function calls related to the exception as the most important decision factors.

Based on the findings from the empirical study, the authors propose a machine learning model to predict whether to log for a code snippet. For this purpose, the authors consider try-catch and return-value-check snippets. To understand the functionality of these snippets, the authors extract the names of included and called functions, the name of the class, and the keywords in comments as features. In addition, the snippets contain a label logged or unlogged. With these features and labels, the authors train a classifier model, which shows a high precision of 90.2% using 10-fold cross-validation.

The paper presents that, in most cases, the functionality of a code snipped can be well understood based on the names in this snippet. This is important for our thesis since we also want to log information about functions to describe the behavior of a program. However, we aim to log more information than the name about functions such as argument and return values. Moreover, the proposal from the paper only considers code snippets that contain try-catch and return-value-checks. Here, we show the advantage of our approach over Fu et al.'s with an example that is logged in our approach but is not suggested to log in Fu et al.'s approach. Consider the code snippet in Figure 2.1. Fu et al.'s approach would not suggest to log this snippet because it contains no try-catch block and no return-value-check. Our approach logs this snippet because we log all function calls. Overall, this paper proposes a method

```
void transferTo(Account bank, double x){
    withdraw(x);
    bank.deposit(x);
}
```

Figure 2.1: counterexample

to suggest whether to log or not, but does not generate logs automatically as we intend to achieve.

Shang [31] provides a pilot empirical study on large software systems. The goal is to bridge the divide between software development and software operation. The paper presents two approaches to leverage field knowledge from operations to improve the software quality in development. First, a high *log churn* enables identification of software bugs. Log churn denotes the frequency of changing log statements in the source code. I.e., a log statement that has changed many times has a high log churn. Using the development history to measure the amount of log churn, one can build statistical models to predict post-release bugs. Secondly, developers can measure test coverage by comparing logs from a test system with logs from the production system. The coverage is the ratio between the called functions in the test system and in the production system. The paper also presents two approaches to leverage development information to improve the quality of operations. First, one can document log lines automatically by attaching development history and bug reports to log lines, such that operators can understand the rationale behind a log entry. Secondly, development can indicate the affected log statements in a new release such that the operators can apply a log filter to check the new release in production more efficiently.

An empirical study of ten large software systems highlights that logs change at a rather high rate across versions, although 40-60% of these changes are not necessary.

This paper shows the importance of logs to gain knowledge about a program. However, this paper does not describe a mechanism of automated logging for function calls. The paper studies the gap between software development and operation empirically and shows approaches that use logs to improve the quality both in development and operations. Overall, the focus is mainly on analyzing logs for a specific purpose and not automatically generating logs of function calls as we want to accomplish in our work.

2.2 Frameworks

Spring Cloud Sleuth [27] implements a distributed tracing solution. A distributed tracing is a method to profile and monitor applications to analyze performance and identify bugs. It is particularly used in microservices architecture to track communication between processes. Sleuth builds upon the widely used *Spring* [26] framework for Java. Sleuth enables us to capture interactions with external systems and store them, e.g., in logs. For example, sending a remote procedure call (RPC) can be such an interaction. Sleuth can serve as an essential tool for enhancing logs between distributed systems. Moreover, it helps to solve the diagnosis problems of multiple services. Sleuth provides a way to track function calls in network communication similarly as we intend in this thesis. However, Sleuth does not track function calls within processes in contrast to our approach.

Apache Thrift [3] is a lightweight framework for scalable cross-language services development. Apache Thrift is available for several programming languages, including Java, Python, and Go. Apache Thrift allows us to define data types and service inter-

faces in a definition file. The compiler generates code to build RPC clients and servers that communicate across programming languages. Apache Thrift's primary goal is to enable efficient communication across programming languages. Apache Thrift comes with an associated code generation mechanism for RPC. However, this mechanism does not apply for calls that are not remote to a server or client. Consequently, we cannot use this framework for our purpose.

Chapter 3

Methodology

3.1 Overview

In this chapter, we study different methods for automated logging of function calls and describe which method is best suited for each programming language.

In general, we need to implement the following three tasks to enable automated logging. We denote it as *automation tasks*.

1. Retrieve all functions and *context data*. By context data, we mean the function name, the argument names and types, and the return type. Additionally, we want to be able to include or exclude functions from modules and packages.

2. For each function F we want to log, we create a new function F'. This new function F' calls the original function F and logs the call to the original function F. We also log its context data along with the argument values and return value.

3. We create a new mechanism that works as follows. During the code's execution, whenever a function F is going to be called, the mechanism replaces that call to F with a call to F'.

The challenge lies in the combination of the second and third tasks, that is, to properly propagate the enhanced function address throughout the original code while being able to call the original function addresses.

Table 3.1 shows an overview of the best-suited methods for Java, Python, and Go. We will explain these methods in the following sections.

	Java	Python	Go
task 1	AOP pointcut	traversal and code inspection	AOP pointcut
task 2	AOP join point and advice	monkey patching	AOP join point and advice
task 3	AOP compiler	monkey patching and `exec`	AOP compiler

Table 3.1: overview of methods and programming languages

3.1.1 Intrusiveness

The aim is to apply the least intrusive method for each programming language. In general, intrusiveness is hard to define. In this thesis, low intrusiveness means that automated logging does not interfere with the program behavior. It also means that users can use it without having to spend too much time modifying the source code or the compiler. In other words, the more supported method for a programming language is less intrusive.

Note that the difference between monkey patching and aspect-oriented programming is small. The gap from these methods to interpreter modification and program transformation is significantly higher. Also note that monkey patching mainly is intended for dynamic programming languages, such as Python. That means that aspect-oriented programming can be less intrusive for compiled languages, such as Java and Go. Overall, we aim to implement monkey patching or aspect-oriented programming in the first place.

We illustrate an overview of the intrusiveness of the different methods in Figure 3.1. We explain these methods in the following sections.

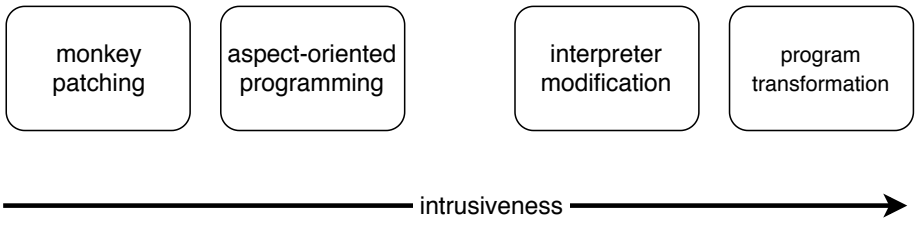

Figure 3.1: intrusiveness of methods

3.2 Logging Methods

3.2.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a programming paradigm to implement *cross-cutting concerns* [37]. Cross-cutting concerns describe features that occur in many different parts of a program. For instance, adding a log statement to all methods is such a concern since it spans over all parts of the program. AOP allows to capture cross-cutting concerns in a concise and modular way. An *aspect* describes such a concern. As an advantage, we implement an aspect once at one single point rather than all over the program. AOP avoids code duplication and enables efficient implementation and maintenance. AOP requires a compiler that combines source code with the aspects. However, AOP requires no changes in the original source code. Therefore, we have a low degree of intrusiveness for this method. AOP also works if we do not have the source code available. We illustrate how AOP works in a simple example built on a question on Stack Overflow [34]. In this example, we have two methods `foo()` and `bar()` where we want to apply a logging aspect.

```
void foo(int value){
    doSomething(value);
```

```
}
int bar(){
    doSomethingElse();
}
```

The logging aspect prints a statement after any method call. Note that the expression `any method is called` is simplified and would be implemented as a regular expression in a real implementation.

```
aspect logging{
    after(any method is called):{
        log.write("called method...");
    }
}
```

The aspect compiler weaves the above aspect together with the source code and produces a result like the following. For simplicity, we show the compiler's output as source code, but in reality, the compiler outputs bytecode.

```
void foo(int value){
    doSomething(value);
    log.write("called method...");
}
int bar(){
    doSomethingElse();
    log.write("called method...");
}
```

However, AOP can also introduce disadvantages. An *Anti-pattern* or *action at a distance* describes a major issue of this programming paradigm. For instance, it is not immediately obvious where part of a method's functionality comes from looking at the body of the method in the source code. In our case, we use one single aspect, namely logging, where we augment methods with a log statement. Otherwise, we do not interfere with the existing code. A developer working on a new project might be wondering where these log entries come from, but the behavior is not changed. Furthermore, there exist development tools that provide support to implement aspects. An example for Eclipse can be found here [10].

3.2.2 Monkey Patching

Monkey patching is a method to modify program behavior of dynamic programming languages at runtime. For instance, we can extend a function with a log statement. We illustrate how monkey patching works in a simple example for Python. The example is based on an article from a website [16]. In this example, we replace the address of `foo` with `monkey_f`. First `monkey_f` calls the original `foo` and then adds a logging statement. Calling `foo` after executing monkey patching then yields the original `foo` augmented with the logging statement. In module `monkey.py` we implement `foo()` as follows.

```
class A:
    def foo(self):
        print("foo() is being called")
}
```

Then we add a log statement to `foo()` at runtime in the following module.

```
import monkey
def monkey_f():
    monkey.A.foo() # call original function
    print("logging...")
monkey.A.foo = monkey_f # replace address
obj = monkey.A()
obj.foo()
```

The output from calling `foo()` is.

```
foo() is being called
logging...
```

The main advantage of monkey patching is that we can use the standard interpreter and implement this method solely with code modification. Therefore, monkey patching has a low degree of intrusiveness. Furthermore, we have a high flexibility to implement log statements. For example, we can insert the log statements before or after a function call and modify the contextual information that we want to log. A main disadvantage is that monkey patching mainly restricts to dynamic programming languages. Moreover, there are similar disadvantages as in AOP regarding readability and understandability of a program since the effect is not immediately visible in the source code. Especially, debugging is hard with monkey patching since we may not get information where a bug happened. Despite monkey patching is used in practice, we need substantial implementation and testing effort to build an automated logging framework. A restriction of monkey patching is that we need the source code to apply it.

3.2.3 Interpreter Modification

Interpreter modification enables data manipulations. For our purpose, we can develop our interpreter that adds log statements to every function call. More concretely, we can take an existing interpreter and modify the interpreter such that it adds a log statement to every function call. Interpreter modification is used in different academic work. For instance, Wang et al. [36] propose a framework that uses a modified Python interpreter to check that a program respects user policies and sensitive data. We illustrate an overview in Figure 3.2. In this figure, we replace the interpreter with a modified interpreter to achieve automated logging.

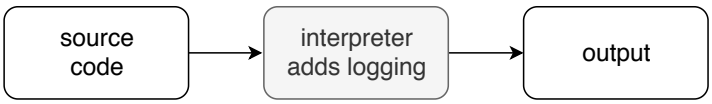

Figure 3.2: interpreter modification

However, the intrusiveness of this method is high. A system that uses this method needs to use a modified interpreter that is not widely used instead of a standard interpreter. Also, this method is limited to interpreted languages.

3.2.4 Program Transformation

Program transformation is a general approach that takes a program and produces another program. A compiler implements such a transformation. For instance, we transform source code into bytecode. During this transformation, we can add log statements to every function. When the original function is called after compilation, then we automatically log the function call. For example, AspectJ is an implementation of a program transformation. We illustrate an overview in Figure 3.3. In this figure, we replace the compiler with a modified compiler to achieve automated logging.

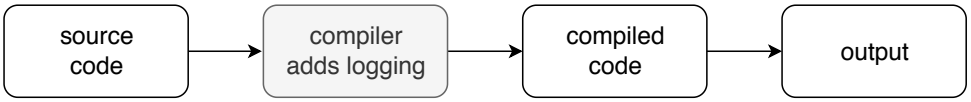

Figure 3.3: program transformation

Similarly, as interpreter modification, the intrusiveness of this method is high since we need to replace a standard compiler by a modified compiler that is not widely used.

3.3 Java

In this section, we explain the least intrusive implementation in Java to achieve automated logging of method calls. Java is an object-oriented and compiled programming language. To study how AOP works for Java, we will briefly introduce the two-step compiling process for Java. The Java compiler (Javac) takes source code (.java files) and produces platform-independent bytecode (.class files). In a second step, the Java virtual machine (JVM) compiles bytecode into machine code. This step includes the use of a class loader, a bytecode verifier, and a JIT compiler. We visualize the process in Figure 3.4.

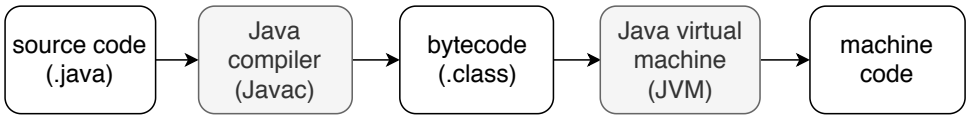

Figure 3.4: Java two step compilation

3.3.1 Method

We implement automated logging in Java with AOP. AOP is widely supported in Java, whereas monkey patching is not supported since it is intended for interpreted programming languages. Thus, AOP is the least intrusive method for Java.

We can express logging as a cross-cutting concern and achieve all three automation tasks in Section 3.1 with AOP. To explain our method, we introduce *AspectJ* [8]. AspectJ is the most prominent AOP implementation for Java and available as open-source software. We express cross-cutting concerns through *join points*, *pointcuts*, and *advices*. A join point is a candidate point where we can insert code. For example, every call to a method is a join point. A pointcut defines at which join points we want to insert code. We can formulate them by regular expressions. For example, we can choose all public methods or only methods within a certain class or package. Advices define the behavior we want to add to pointcuts.

First note that the syntax of a pointcut is `call(package.class.method(arguments))` and we can exclude packages, classes, and methods by regular expressions. We define the pointcuts as `call(* *.*(..))`, such that we catch all public and private method calls.

Secondly, we define an advice that outputs logging information about a called method. Java object `thisJoinPoint` of type `JoinPoint` provides static and dynamic information about join points, such as the arguments of the join point. We can use method `getSignature()` to get the method name and methods `getParameterNames()` and `getArgs()` to get the argument names and values, respectively. Note that we retrieve the return value and type after the method call.

Thirdly, we pass the existing program together with the aspects defined in the previous steps to the AspectJ compiler. The compiler provides three different stages to weave aspects together with target code. The most straightforward approach is compile-time weaving. When we have the source code available, the AspectJ compiler compiles from source code and produces woven class files as output. The JVM loads the woven class files as standard Java classes. If the source code is not available, we can use post-compile or binary weaving to weave existing class files. The last strategy, load-time weaving, is similar to post-compile weaving but defers binary weaving until the class loader loads a class file.

There are many different ways how to build an AspectJ project. Here we explain it using *Apache Maven* (referred as Maven) [4] for build automation. First, we define one of the three weaving stages in the project object model `pom.xml`. Then we build and execute the program with the following two commands, respectively. `mvn clean install` and `mvn exec:java`. Moreover, Eclipse provides the AspectJ Development Tools (AJDT) [10], where we can create projects with aspects. We can also add aspects to an existing Java project with the commands `Configure` and `Convert to AspectJProject`.

Note that we choose the classical AspectJ notation, where we define the aspects in a separate file. However, it is also possible to define the aspects using annotation-based style. Further information for AspectJ setup is available here [9] and here [5].

Limitations

- We cannot limit the depth of function calls.

We implement the method in `Autolog.aj` and release the implementation publicly on GitHub [13].

3.4 Python

In this section, we explain the least intrusive implementation for Python to achieve automated logging of function calls. Python is an interpreted and dynamically strongly typed language. Although we develop our solution for Python 3.6 and newer, it is straightforward to extend the support to all Python 3 versions. Python distinguishes between methods and functions, as methods are associated with a class, while functions are not. For simplicity, we mean both functions and methods by writing simply about functions, unless stated otherwise.

3.4.1 Method

Since Python is a dynamic programming language, we can apply monkey patching. That means we can implement automated logging in the source code without changing the interpreter. Therefore, monkey patching is the least intrusive method that we can apply for Python. We satisfy the three automation tasks in Section 3.1 by implementing `autolog.py`, a framework for automated logging in Python.

In the first task, we need to list all functions. Therefore, we recursively traverse a given list of modules. In this list, we define the modules that we want to log automatically. The traversal continues until we have traversed all functions in the modules or until we have reached a certain recursion depth. The recursion depth has default value one, but we can configure it to other values. After completion of the first step, we obtain a list of all functions and their signatures. To retrieve functions from a module, we use the library `inspect` [29]. This library allows us to inspect source code and get information about live objects such as modules, classes, methods, and functions. With the following command, we retrieve a list of functions inside a given module.

```
inspect.getmembers(module)
```

For a given function name, we retrieve the signature, also using the inspect module.

```
inspect.signature(function)
```

Note that we do not retrieve monkey functions to prevent reinspection. With monkey functions, we mean the functions that we introduced in this method.

In the second task, we want to create a new function that calls the original function and adds logging. Given the function signature and the argument and return values, we produce a monkey patch `f_monkey`, i.e., a new function definition that calls the original function and logs the original function call. Note that we implement the monkey patch as a string that we can execute as a program later.

In the third task, we want to replace the original functions with the attached function. After the definition of `f_monkey` from the previous task, we replace the original function by the new function `f_monkey`. To dynamically execute the string as a program, we can use built-in function `exec`.

```
exec(string_f)
```

The implementation of `autolog.py` comes with the following workarounds and limitations.

Workarounds and limitations

- We do not log inner functions, also called nested functions. Inner functions are functions defined within other functions. The reason is that module `inspect` does not retrieve inner functions since they are invisible outside of its immediately enclosing function. However, if we want to log inner functions, we outline a possible approach. Module `inspect` offers a method `getsource()` to retrieve source code. We can use this to retrieve the source code of function definitions. For example, with library `ast` [28], we can traverse the source code of function definitions recursively and retrieve inner functions. We show an example of an inner function in Figure 3.5. Note that inner functions are able to access variables of the enclosing scope.

- Similar to the previous limitation, we do not log lambda functions since `inspect` does not retrieve them. If we want to retrieve lambda functions, we can use a similar approach as for inner functions.

- We do not log functions that contain the following strings in the signature because they are not executable.
`<locals>`, `_get_kwargs`, `__repr__`, and
`argparse._ActionsContainer._get_handler`.

- We skip methods that have a non-executable signature, e.g., that contain an invalid path. More formally, when an object's built-in method `repr()`, which should return an executable string of the object, returns something with an invalid syntax (e.g., symbols `<, >, ', ", /,` \, ...), we want to skip it.

```
def print_people_asc(people):
    def comp(person1, person2):
        return person1.age<person2.age
    for p in sorted(people, comp):
        print(p)
```

Figure 3.5: example of an inner function

We release the implementation of `autolog.py` publicly on GitHub [14].

3.5 Go

In this section, we explain the least intrusive method for Go to achieve automated logging of function calls. Go, also known as Golang, is a statically typed and compiled programming language. Go distinguishes between methods and functions. Although Go has no classes, we can define methods on types. For simplicity, we mean both functions and methods by writing simply about functions, unless stated otherwise. Furthermore, Go has built-in support for multiple return values.

In Go, monkey patching and AOP are both more intrusive than for Java and Python since none of them is currently widely supported. Nevertheless, we can achieve automated logging of function calls with both methods, as we will describe in the next two sections.

3.5.1 First Method

In this section, we describe how to apply monkey patching. Note that even though Go is a compiled language, it is possible to apply monkey patching. Recall the three automation tasks for automated logging in Section 3.1.

In the first task, we want to list all functions and retrieve their fully qualified name. We can use libraries operating over abstract syntax trees to traverse function definitions and retrieve the function names. The according libraries in Go are `go/ast`, `go/parser`, and `go/token`. To retrieve the argument and return types, we can use module `reflect` [18]. However, we cannot retrieve the argument names of a function with `reflect`. The reason for this is that position and type determine the arguments, and the name is not essential. See a discussion on Stack Overflow on this topic [33].

In the second task, we want to create a new function that calls the original function and adds logging. We describe an implementation using monkey patching. First of all, it is not possible through regular language constructs. Nevertheless, we can achieve monkey patching by rewriting the running executable at runtime and inserting a jump to the function to call instead. Note that this approach can be unsafe in general. `GoMonkey` [35] is a Go library providing a monkey patching API. We can retrieve the argument values simply after the function is called by traversing the argument list. Similarly, the function computes the return values, and we can add them to the logging statement. We can replace functions using a `GoMonkey` API call.

```
monkey.Patch(<target function>, <replacement function>)
```

To replace an instance method inside a type, we need to add the type.

```
monkey.PatchInstanceMethod(<type>, <name>, <replacement>)
```

However, in the third task, we want to replace the original functions with the new function. The main challenge is to call the original function within the new function. To call the original function within the replacement, we use `monkey.PatchGuard`.

Limitations

- We cannot log argument names with the described approach. Further information are available here [33].

3.5.2 Second Method

In this section, we briefly describe an AOP implementation to achieve automated logging of function calls. AspectGo [2] provides an aspect-oriented framework for Go. Note that there are several other projects on GitHub that provide aspect-oriented frameworks for Go, e.g., [20]. To achieve the three automation tasks in Section 3.1, we define a pointcut for all functions and methods. In the aspect, we implement the logging behavior. Finally, we build the project with the AOP compiler to weave the aspects into the methods.

Limitations

- We can only use a single aspect file. However, this is not relevant for our purpose since we have only one aspect.

We release the implementation of a prototype publicly on GitHub [12].

3.6 Deployment

3.6.1 Logging Structure

We provide the logs in a structured format. Independent of the logging structure, we want to log the timestamp and the function name. For the arguments, we want to log the types, names, and values. From the return statement, we want to log the value and type. We provide the log data in two formats. First a machine-readable format in JavaScript Object Notation (JSON) [23]. JSON enables automatic processing for, e.g., machine learning algorithms. We show an example in Figure 3.6.

```
{
    "time": 2019-09-20 09:20:19,
    "name": "package.class.foo",
    "arguments": [{
        "type": "int",
        "name": "a",
        "value": 1
    }, {
        "type": "String",
        "name": "b",
        "value": "hello"
    }],
    "return_type": "string",
    "return_value": "hello 1"
}
```

Figure 3.6: machine-readable JSON format

Secondly, for debugging and operations, we also implement a human-readable format that contains the data in one line. We show an example in Figure 3.7.

```
"2019-09-20 09:20:19, name = package.class.foo(int, String),
arguments: (a = 1, b = hello), returns = hello 1 (String)"
```

Figure 3.7: human-readable logline format

To select the format, we implement two functions `getLogLine()` and `getLogJson()` that return the format as a string.

3.6.2 Security

Logs can contain security-relevant information like user names, passwords, keys, and many more. Logging function calls automatically can leak such information. To address this concern, we provide an option to exclude libraries. For instance, we can exclude libraries that contain user passwords or keys from automated logging. Nevertheless, automated logging is still risky to use in production. Furthermore,

the ability to exclude libraries can also help to reduce performance issues due to automated logging.

Another concern is the use of another compiler or interpreter. Especially, when this compiler is not widely used and regularly updated, the compiler might increase the vulnerability of a system. E.g., due to missed security updates.

Depending on the intention of the log data, we could also consider to introduce a mechanism that anonymizes user data when creating log statements to provide privacy. For example, we could remove user names from the logs.

Chapter 4

Experiments

4.1 Java

In this section, we describe our measurement setup and present our results in terms of performance and storage.

4.1.1 Setup

To study the effect of automated logging on performance and storage, we use an open-source e-commerce application [1]. The application has a microservices architecture and uses *Kubernetes* [19] for automatic deployment of containerized applications. For container management, the application uses *Docker* [6]. We apply automated logging to all methods of the `orders` service. The `orders` service has about 2000 lines of code in Java. To apply automated logging, we extend the Apache Maven project object model and add the following dependencies to the `pom.xml` file. The dependency on the AspectJ runtime library `aspectjrt.jar`.

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.8.9</version>
</dependency>
```

To introduce an advice to classes at load time, we also need to include `aspectjweaver.jar`.

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.9</version>
</dependency>
```

To produce log statements in JSON format, we also add the JSON.simple [24] library to the dependencies.

```
<dependency>
    <groupId>com.googlecode.json-simple</groupId>
    <artifactId>json-simple</artifactId>
```

```
        <version>1.1.1</version>
    </dependency>
```

Since we have the source code of the aspect and the application, we enable compile-time weaving to weave aspects into the existing classes in the AspectJ compiler.

```
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <version>1.7</version>
        <configuration>
            <complianceLevel>1.8</complianceLevel>
            <source>1.8</source>
            <target>1.8</target>
            <showWeaveInfo>true</showWeaveInfo>
            <verbose>true</verbose>
            <Xlint>ignore</Xlint>
            <encoding>UTF-8 </encoding>
        </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>compile</goal>
                    <goal>test-compile</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
```

The dependencies are also available under Maven Central Repository [4]. Finally, we add the logging aspect defined in `Autolog.aj` to the project. The AspectJ compiler takes the aspects defined in `Autolog.aj` and produces woven class files as output.

4.1.2 Performance

We perform a load test to measure the impact of automated logging on the CPU performance. The application provides tests in an open-source load testing tool, *Locust* [25]. Locust supports simulation of end-users to run load tests distributed over multiple machines. The tests perform a high number of requests to ensure the requests run during the whole measuring time. We check this by measuring the CPU usage at the beginning and after the measurement. If the CPU usage after the measurement time is about as high as during the measurement, then we know that requests are still running. The details about the execution and parameters of the tests are in Table A.1.

We measure the CPU usage in two different settings for 70 minutes. First, with full automated logging, where we log every method call within all packages. Secondly, we measure the time without logging as a reference. To exclude the startup time from

the measurement, we compute the CPU load as follows. Where X denotes consumed CPU time after ten and Y after 70 minutes.

$$\text{CPU load} = \frac{X - Y}{60}$$

We show the measurement results in figure 4.1. Full logging increases the CPU load from about 0.5 to 1% in comparison without logging. This is about a factor of two. However, both CPU utilization's are low. Note that the low numbers for CPU utilization stem from the microservices architecture, where multiple services run together, and `orders` is one of them.

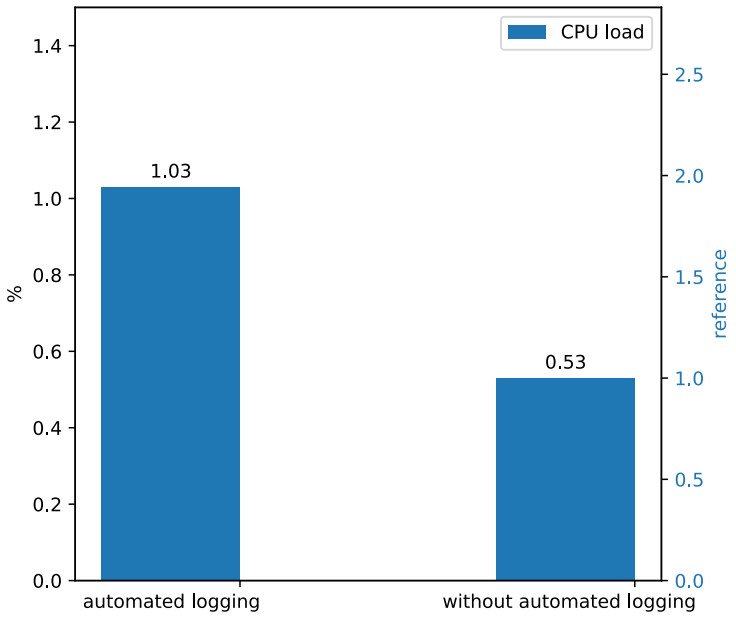

Figure 4.1: CPU utilization

4.1.3 Storage

During the load test, as described previously, we also store and measure the output in the form of logs to the console. In Table 4.1, we show the results of log output size with and without logging. We observe that automated logging produces about 1'300 times the amount of output than without automated logging in our setup.

	log size in MB
without automated logging	0.058
full logging	79.121

Table 4.1: log size during a run time of 70 minutes

Countermeasures Since automated logging produces large log files, we propose approaches to address this issue.

First of all, to reduce the required storage size, we can compress log files. For example, we can use GNU Tar [17] for data compression.

```
tar -czvf fullLogging.tar.gz fullLogging.log
```

In our case, the size reduces by a factor of about 20 from 79 MB to 4 MB.

Secondly, we can use existing services and frameworks to store and process log files. For example, IDrive [21] provides online data storage. The costs to store 2 TB for one year are 52 USD. If we regularly analyze data, we can delete the logs after analysis. For instance, we might store the data for one day and analyze them during the night time. Based on the measurement of the application for 70 minutes, we expect the following data size for automated logging for one day.

$$\text{log size in one day} = 79 \text{ MB} \cdot \frac{24 \cdot 60}{70} = 1.6 \text{ GB}$$

Note that the application for the measurements is a rather small application. For large applications, however, we can estimate the storage based on this measurement. For instance, e-commerce application eBay had a codebase of 50 million lines of code in 2011, according to this article on eBay [7]. The order application from the measured e-commerce application has about 2000 lines of code. Since they are both e-commerce applications, we estimate the data size for eBay by interpolation.

$$\text{log size in one day} = 1.6 \text{ GB} \cdot \frac{5 \cdot 10^7}{3 \cdot 10^3} = 26.67 \text{ TB}$$

The costs to store 30 TB on IDrive for one year are 448 USD. We note that we want to store the data for more than one day, the costs will multiply with the number of days. As an alternative, Splunk [32] is a tool that provides comprehensive log management with storing, searching, and analyzing for big data sets.

Thirdly, we can exclude libraries to reduce the data size. We can do this by specifying paths in the source code that we do not want to log. Furthermore, we can limit the depth of function calls. For example, we could only log function calls on the first level.

4.2 Python

In this section, we describe our measurement setup and present our results in terms of performance and storage.

4.2.1 Setup

We use *Cartridge*, an e-commerce shopping cart application [30] built on the *Django* [22] web framework. To activate automated logging, we add `autolog.py` to directory `project_name`. In the same directory, we activate automated logging in file `manage.py` by adding the following code.

```
import django.core.management
from autolog import Autolog
```

```
if __name__ == "__main__":
    autolog = Autolog([django.core.management])
    autolog.run()
```

We follow the instructions from the manual. First, to install Cartridge with the required dependencies we run.

`pip install -U cartridge` or `python setup.py install`

Next, we create a new project and change to this project

```
mezzanine-project a cartridge project_name
cd project_name
```

Then we create some content for demonstration.

```
python3 manage.py createdb noinput
```

Finally, we run the application.

```
python3 manage.py runserver
```

Now we are able to browse to `http://127.0.0.1:8000/admin` and can login with username `admin` and password `default`.

4.2.2 Performance

To perform a load test, we write a small script in Python that simulates user requests to the Cartridge application. The script sends HTTP requests periodically to the web application. The script is available in Appendix A.2. We set the rate of the requests high, such that the CPU has a utilization between 50-80%. We note that we run this setup in another environment than for Java. We run the script for 60 minutes and measure the CPU utilization. To find the process ID (PID) we run.

```
ps aux | grep runserver
```

To measure the time we run. Note that `time` denotes the CPU utilization of a process.

```
ps -eo pid,time,etime,args | grep <PID>
```

We show the results in Table 4.2. Note that the format of `time` is `mm:ss`.

	starttime	10 minutes minutes
without automated logging	00:02	07:11
automated logging	00:02	07:20

Table 4.2: CPU utilization time

4.2.3 Storage

During the load test, we store the output during 60 minutes into a file and compare the resulting file sizes with automated logging and without automated logging. The results are in Table 4.3.

	log size
without automated logging	1 KB
automated logging	1.4 GB

Table 4.3: log size during 60 minutes

Countermeasures to handle large log files, we can apply the same mechanisms as described for Java.

4.3 Testing

How can we ensure that we log every function call? Ideally, we would formally verify that the three automation tasks we described in Section 3.1. are correctly implemented. This verification is left for future work, as developing the frameworks required substantial effort. We explain next how we implemented automated logging for Java and Python and show how we could implement it for Go.

In Java, we log all method calls since an AspectJ pointcut retrieves all methods [11]. The AspectJ compiler weaves the aspects into all methods and produces code with new methods that contain the original methods with added logging statements. As a result, we log every method call in Java automatically.

In Python, however, we implemented our own framework. Therefore, we tested the functionality empirically. We provide several test modules using different function and method calls and external libraries. We tested various arguments and return types to produce log statements. We provide test for numeric, boolean, string, and reference inputs and check the corresponding log output.

Chapter 5

Discussion

5.1 Conclusion

In this thesis, we studied four different methods to enable automated logging of function calls for three major programming languages. We analyzed and implemented different methods for automated logging. We showed for every programming language the least intrusive method. In Java, we implemented automated logging with aspect-oriented programming as the least intrusive method. In Python, we implemented a framework that uses monkey patching to achieve automated logging. However, there are some functions that we do not log. In Python, for example, we do not log inner functions with our approach. However, we can achieve this by traversing the source code of functions and retrieving inner function definitions. In Go, we propose a method with monkey patching and aspect-oriented programming to achieve automated logging. In the experiments, we measured the runtime and the storage for Java. The measurements showed that the utilization is low with automated logging. In our measurements, however, automated logging creates a significant raise of the log size.

5.2 Future Work

The goal of this thesis is to log all function calls. Experiments showed that the resulting logs are large. For future work, however, we could provide a mechanism that distinguishes important function calls from others that we do not need to log. This would reduce the required storage to process the log data and also enhance the quality of features.

In Go, we could implement the two methods described in sections 3.5.1 and 3.5.2. However, we would suggest to start with an aspect-oriented implementation since Go is a compiled programming language. Nonetheless, it would be interesting to compare the two implementations in terms of performance and storage. Also, we can apply and implement methods for other programming languages.

In Python, the implemented framework does not log inner functions and lambda functions. To address this, first of all, we would suggest to evaluate if these functions contain valuable information to log. For example, we could perform a source code analysis of a large software system, where we evaluate if inner functions contain im-

portant information for our purpose. Alternatively, we could perform a questionnaire survey among developers to find out more about the behavior of inner functions. However, if we want to log them, we could implement the approach outlined in Section 3.4.1.

For all three programming languages, we could evaluate automated logging on larger software systems. In Section 4.1.3, we estimate the log size for larger software systems. However, we could apply and evaluate automated logging to larger software systems. We could measure the CPU performance and storage.

To enable user privacy, we could implement an anonymization mechanism for log data, such that a user's identity is hidden. This can be done after the generation of the logs. For example, we could aim to achieve differential privacy, such that it is not observable if a user appears in the data or not.

Bibliography

[1] A. Acosta, I. Dmitrichenko, I. Crosby, J. Smith, P. Winder, V. Lal. Java e-commerce application. https://github.com/microservices-demo. Accessed: September 24, 2019.

[2] A. Suda. AOP framework for Go. https://github.com/AkihiroSuda/aspectgo. Accessed: October 10, 2019.

[3] Apache Software Foundation. Apache Thrift framework. https://thrift.apache.org. Accessed: September 28, 2019.

[4] Apache Software Foundation. AspectJ Maven. https://mvnrepository.com/artifact/aspectj. Accessed: October 1, 2019.

[5] Baeldung. AspectJ tutorial. https://www.baeldung.com/aspectj. Accessed: October 1, 2019.

[6] Docker, Inc. Docker. https://www.docker.com. Accessed: September 24, 2019.

[7] eBay Inc. eBay lines of code. https://tech.ebayinc.com/engineering/onboarding/. Accessed: October 14, 2019.

[8] Eclipse Foundation. AspectJ. https://www.eclipse.org/aspectj. Accessed: September 24, 2019.

[9] Eclipse Foundation. AspectJ annotation. https://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj.html. Accessed: October 1, 2019.

[10] Eclipse Foundation. Aspectj development tools. https://www.eclipse.org/ajdt. Accessed: October 2, 2019.

[11] Eclipse Foundation. AspectJ pointcut. https://www.eclipse.org/aspectj/doc/next/progguide/language-joinPoints.html. Accessed: October 1, 2019.

[12] F. Engler, K. Kubicek. Automated logging in Go. https://github.com/englerfa/Logging_Go. Accessed: October 1, 2019.

[13] F. Engler, K. Kubicek. Automated logging in Java. https://github.com/englerfa/Logging_Java. Accessed: October 1, 2019.

[14] F. Engler, K. Kubicek. Automated logging in Python. https://github.com/englerfa/Logging_Python. Accessed: October 1, 2019.

[15] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33. ACM, 2014.

[16] Geeksforgeeks community. Monkey patching in Python. https://www.geeksforgeeks.org/monkey-patching-in-python-dynamic-behavior. Accessed: October 3, 2019.

[17] GNU community. GNU Tar. https://www.gnu.org/software/tar. Accessed: October 10, 2019.

[18] Go authors. Go reflections. https://golang.org/pkg/reflect. Accessed: October 12, 2019.

[19] Google LLC. Kubernetes. https://kubernetes.io. Accessed: September 24, 2019.

[20] H. Huang, X. Zheng. AOP framework for Go. https://github.com/gogap/aop. Accessed: October 10, 2019.

[21] IDrive Inc. Data storage. https://www.idrive.com/idrive/signupBusiness. Accessed: October 14, 2019.

[22] J. Rief, R. Fleschenberg. Django framework. https://www.djangoproject.com. Accessed: October 1, 2019.

[23] JSON. JSON. https://www.json.org. Accessed: October 2, 2019.

[24] JSON simple contributors. JSON library for Java. https://github.com/fangyidong/json-simple. Accessed: October 10, 2019.

[25] Locust contributors. Locust. https://locust.io. Accessed: September 24, 2019.

[26] Pivotal Software. Spring framework. https://https://spring.io. Accessed: September 28, 2019.

[27] Pivotal Software. Spring Sleuth framework. https://spring.io/projects/spring-cloud-sleuth. Accessed: September 28, 2019.

[28] Python Software Foundation. Abstract syntax tree library. https://docs.python.org/3/library/ast.html. Accessed: October 3, 2019.

[29] Python Software Foundation. Inspect library. https://docs.python.org/3/library/inspect.html. Accessed: October 3, 2019.

[30] S. McDonald. E-commerce shop. https://github.com/stephenmcd/cartridge. Accessed: October 14, 2019.

[31] W. Shang. Bridging the divide between software developers and operators using logs. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1583–1586. IEEE Press, 2012.

[32] Splunk, Inc. Splunk. https://www.splunk.com. Accessed: October 10, 2019.

[33] Stack Exchange, Inc. Reflect argument name in Go. https://stackoverflow.com/questions/31377433/getting-method-parameter-names. Accessed: October 7, 2019.

[34] Stack Exchange, Inc. What is aspect-oriented programming? https://stackoverflow.com/questions/242177/what-is-aspect-oriented-programming. Accessed: October 3, 2019.

[35] B. van der Bijl. Monkey patching in Go. https://github.com/bouk/monkey. Accessed: October 3, 2019.

[36] F. Wang, R. Ko, and J. Mickens. Riverbed: Enforcing user-defined privacy constraints in distributed web services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[37] Wikipedia contributors. Aspect-oriented programming — Wikipedia, the free encyclopedia, 2019. [Online; accessed 15-October-2019].

[38] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 415–425. IEEE Press, 2015.

Appendix A

A.1 Java Experiment

We show the configuration parameters of the load test in Table A.1. We run the experiments on a test server.

parameter	value
replicas	8
clients	5'000'000
requests per client	10'000
hatch rate	50

Table A.1: Java load test parameters

The hatch rate describes the number of clients added per second. We set the parameters in the configuration file.

```
deploy/kubernets/manifests/loadtest-dep.yaml
```

Next, we start a Minikube.

```
minikube start --memory 16284 --cpus 8
```

Then we create a resource from file.

```
kubectl create -f deploy/kubernetes/manifests/sock-shop-ns.yaml
-f deploy/kubernetes/manifests
```

We measure the time the application has used the CPU with the following command.

```
kubectl exec -it --namespace sock-shop orders-765f7dfb6b-hmq4r
-- ps -eo pid,comm,etime,time,args
```

A.2 Python Experiment

In Figure A.1, we show the script to perform load tests. We perform ten different HTTP requests to simulate user action periodically.

```
import requests
import time

url = 'http://127.0.0.1:8000/admin/login'
payload = {'username': 'admin', 'password': 'default'}
requests.post(url, data=payload)

sleeptime = 0.01
while True:
    resp = requests.get('http://127.0.0.1:8000/admin/shop/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/product/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/product/1/change/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/productoption/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/discountcode/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/sale/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/sale/?o=3')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/shop/order/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/media-library/browse/')
    time.sleep(sleeptime)
    resp = requests.get('http://127.0.0.1:8000/admin/media-library/browse/
                                        ?o=date&ot=desc&dir=gallery')
    time.sleep(sleeptime)
```

Figure A.1: load test script

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.