
Bachelor Thesis

# Systematic Identification of Java Reflection in Static Analysis

Luca Simon Stamen

July 17, 2024

Reviewer:
JProf. Dr.-Ing. Ben Hermann
Msc. Anemone Kampkötter


Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl V - Programmiersysteme
Fachgruppe Softwaretechnik sicherer Systeme
https://sse.cs.tu-dortmund.de

# Abstract

Static analysis is a powerful tool in finding and mitigating existing vulnerabilities in programs and real-world projects. Existing and future tools implementing a static analysis face the problem of unsoundness especially within their call graphs. In many cases unsoundness leads to false negatives within the result, which means hidden exploits exist although the analysis returns none. Java's reflection library, for inctance, offers methods causing these sources of unsoundness. To identify these sources and give an overview of their usage, a static analysis called ReflectDetect is created and presented in this thesis. ReflectDetect correctly finds invocations of those methods and differentiates them into dynamic feature categories. Additionally a study on real-world projects with ReflectDetect is done, which provides knowledge about the importance of the library for static analysis. The results are also evaluated for the existence of Reflection API patterns that are used on a regular basis providing more detail about the overall use cases and distribution of the reflection library in current software projects.

# Contents

# 1 Introduction

## 1.1 Motivation

Static analysis tools have certain requirements, like soundness and precision, ensuring that their result includes all vulnerabilities of the analysed project. A sound analysis considers every possible execution. A precise analysis considers only possible executions. In most cases the mentioned tools are unsound to a small extent, for instance, by using Java's reflection API. Therefore, dynamic analysis tools are used to further improve the soundness and precision of the overall result.
Future dynamic analysis tools inspecting reflection calls can improve their efficiency drastically by knowing at which program counter such calls are happening. Furthermore, excluding methods that are not being severe or relevant for unsoundness adds to that. To achieve that, ReflectDetect, a static analysis on Java projects, which searches for method invocations of the reflection library, needs to be developed. For the implementation of ReflectDetect I used the OPAL framework, which is written in the programming language Scala and offers a broad amount of functionality to create static analysis for Java projects.
The paper "Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study" by Davy Landman et. al.[9] is the groundwork for the whole implementation, study and overall thesis. Landman themselves do a study on the occurrences of these APIs. Most importantly they divide all methods inside the reflection library into two sets and further categorise them. One set consists of methods potentially causing unsoundness and the other set consists of definitely sound ones. Therefore, these categorisations perfectly fit the narrative of excluding findings for future efficiency of dynamic analysis tools.

## 1.2 Research questions

**Q1: What are the sources of unsoundness in call graphs caused by the reflection API?**
**Q1** aims at finding the reasons for unsoundness emerging during call graph computations. Following that the ability to resolve those sources needs to be evaluated to get additional information about the severity of the findings presented in this thesis. To obtain this data, scientific research, more specifically related work, is considered, chosen and summed up.

**Q2: Which reflection API calls and patterns are commonly used in real-world Java Projects?**
**Q2** aims at retrieving real-world data about different Java projects and to which degree they are affected by unsoundness through the reflection library. The data is collected with ReflectDetect, a static analysis tool designed and implemented for this thesis. The collected data is then evaluated and put into the context of this thesis. The resulting findings combined with the backing scientific information from **Q2** will give an overview of the severity and importance of the reflection framework.
To retrieve more semantic details about the found reflection calls, different categories are used as well as

the analysis for existing combinations, called patterns, to differentiate them from each other.

## 1.3 Thesis structure

The section *Basics* introduces and explains relevant terminology in the context of this thesis. That includes definitions important for the design and implementation of ReflectDetect as well as the resulting study.
The section *Related Work* sets the scientific environment in which the study takes place. Key information for the study is collected and presented from relevant papers talking about unsoundness and their often underlying reflection library.
The section *ReflectDetect* presents the design and implementation process of ReflectDetect, a static analysis tool for Java projects. Here all requirements on which the implementation is based on are defined and elaborated. Further a test of the functionality is presented followed by the limitations the tool faces at the moment.
Section *Study* analyses real-world Java projects with the tool presented in the previous section. Here the results are collected, processed and presented in regards to the existence of patterns and general metrics about the usage of the reflection library.
The section *Research Questions* combines the data presented in previous sections to answer the two research questions leading to the creation of this thesis.
The section *Conclusion* sums up the whole thesis and the most important key points.
The last section *Future Work* discusses possible research and implementations in the future based on this thesis. That includes additions to the implementation of ReflectDetect as well as the usage of the current version to detect reflection calls.

# 2 Basics

For the implementation, study and related scientific research important concepts and definitions are explained in this chapter.

## 2.1 Call graph

A (static) call graph, based on the invocation behaviour, contains vertices and edges abstracting and modeling the possible behavior of a certain project. Each vertex represents a reachable method and holds additional information about it. Each directed edge of a vertex represents a possible invocation of the method held by the destination vertex inside the body of the method held by the current vertex[13].

The construction of a call graph is higly dependent on the base algorithm and its implementation by a certain static-analysis (tool). "Class Hierarchy Analysis" (CHA), "Rapid Type Analysis" (RTA) and "Separate sets for methods and fields" (XTA) are well known implementations and available in the OPAL-Framework, which I use in this thesis for the study on sources of unsoundness. Also available are CTA, FTA, and MTA which are on the spectrum between RTA and XTA, which means that they are based on RTA and add or change design aspects until they get to XTA. Every implementation has different precision and scalability propositions. My implemented static analysis tool ReflectDetect, presented in this thesis, uses the RTA implementation given in OPAL.[14]

RTA is based on the class hierarchy and further takes into account class instantiation. First, it calculates the call graph of the CHA algorithm, then refactors the result to get to the RTA.
CHA constructs a node for each method. For every callsite that is a static dispatch inside the body of a method $M$ it adds a directed edge to the method $M'$ referenced by the callsite. If the callsite is a dynamic dispatch, CHA looks at the declared type of the object receiving the invoke and only adds a directed edge from $M$ to the method $M''$ of the given type.
After CHA finished RTA takes the resulting call graph and calculates a new one based on it. It takes all nodes from the CHA call graph and only adds the edges relevant inside the body of the entry-point (most likely the main method). Methods that are now reachable through these edges are temporarily stored and considered in the next iteration. This is repeated until the data structure, where the methods are stored, is empty. Therefore the resulting call graph is a subset of the CHA call graph containing less edges and equal or less nodes.
The RTA in general is more precise than the CHA algorithm. CHA on the other hand is sound in contrast to the possibly unsound RTA[6].

## 2.2 Soundness, Precision & Completeness

To compare the different static analysis tools, certain requirements have to be defined. To be efficient and effective the analysis should meet the following requirements:

1. Soundness: Every possible execution is considered and modelled
2. Precision: Only possible executions are considered and modelled

If both definitions are met, the analysis fulfills completeness and is seen as effective and efficient.
A call graph that does not reach a method through its edges, which is called in a certain execution of the program, is therefore considered unsound and the underlying analysis can never be sound[13].

Another approach of talking about soundness and precision are false positives (FPs) and false negatives (FNs). An unsound analysis does not consider every possible execution and thereby possibly results in less exploits (or none) than there really are. Those would be FNs.
Conversely an imprecise analysis could result in detected exploits which are not relevant for the given program, because they are not reachable by any execution. These would be FPs[13].

## 2.3 Control-Flow graph

A control-flow graph (CFG) consists, similarly to the call graph, of vertices and directed edges. For the CFG a vertex represents a statement or in other words an instruction. Each directed edge between two vertices models the flow of control between two statements.
It also consists of additional information. By tracing back the edges, certain constraints about data being relevant for loop- or conditional-statements can be derived. These vertices always have two outgoing edges representing the following control-flow on whether the condition was true or not.
To build a CFG all these constraints and conditions need to be tracked and temporarily saved. Although a CFG is more detailed than a call graph, the major trade-off is the performance of the calculation[8].
A CFG may be as unsound as any call graph for the same project. Any source of unsoundness is able to affect both computations in the same way because there is missing information about certain invocations. With potentially complex calculations, the CFG is able to resolve some of them by analysing its own current state, tracing back the needed information, and therefore staying less unsound than the call graph.

## 2.4 Java reflections

The Java reflection API or Library gives certain capabilities to a program during execution. A program using reflections is able to examine and manipulate itself during runtime, for example the internal properties[1].
Using reflections therefore represents a very powerful tool to enhance the possibilities of developers and simultaneously increase the complexity of the project. That results in more possible attack vectors for malicious actors and challenges for static analysis.

[1] https://www.oracle.com/technical-resources/articles/Java/Javareflection.html

## 2.5 Definition: Reflect Pattern

A reflection pattern or just pattern will be used to describe the occurrence of reflect methods that are called or invoked very close to each other. A pattern consists of at least two reflection calls up to as many as possible. Reflection methods are considered close when either the line number or the program counter (pc) at any time is not farther away than 20 (which needs to be multiplied by four for the program counter, caused by byte addressing and 32-bit processors). That number is explicitly defined and used for this thesis. In general, the bigger the distance between two methods, the less likely a semantic correlation is.
A pattern also needs to fulfil semantic restrictions. Two reflection calls working on completely different things, for example method invokes and array manipulation, cannot be considered a pattern, because they cannot influence each other during execution.

# 3 Related Work

This chapter presents scientific research on unsoundness and the reflection library in the context of the study fitting. The data will be used in combination with the results found in the study to answer the research questions.

## 3.1 Reasons for unsoundness

Unsound static analysis tools have the drawback to include FNs wherever unsoundness appears. The root of the problem has its ground in the call graph construction. Every execution always has the requirement for a certain amount of efficiency. Talking about call graphs, that needs to be computed for thousands over thousands of classes and their method invocations, the construction faces one main problem: the trade-off between precision and scalability. This trade-off is the most prevalent reason for unsoundness[11].
A computation running exponentially longer to evaluate every possible calculation to fulfill precision is not scalable. A huge codebase or a lot of different projects cannot effectively be securely maintained in that way. Therefore to a varying degree, existing call graph algorithm implementations do sacrifice some soundness in their construction and precision to gain scalability[11].
Another reason for unsoundness on the developers side is the development costs for specific edge cases of the language features and different frameworks. Notably costs can also mean the time consumed to implement. These cases may have a not negligible impact on the soundness of the call graph but are still too expensive to check to be considered during implementation[11].
From the software side many language features and edge cases enabled through different libraries or frameworks can lead to unsoundness. For this thesis the summary is broken down to the aspects causing the reflection library to enable unsoundness. Other language features and edge cases are elaborated in the study by Michael Reif et. al.[11].
Reflection calls in general are unsound the moment certain information is missing. A reflection call invoking a method on an object without knowing the object cannot be correctly added to the call graph. Just adding every possible execution would fulfill soundness on the cost of precision, which would be way worse. Therefore not resolvable calls do not add edges and stay unsound.
The reflection calls are distinguishable into trivial, locally resolvable, context-sensitive and method handles after Reif et. al.[11]. **Trivial** reflection calls, as the name suggests, are easy to resolve because the call target is immediately available. For example a call where the String pointing to the loading of a class is a constant like it is in this example: *Class.forName("XYZ")*.
**Locally resolvable** reflection calls may not be trivial but still manageably resolvable. They are just appending on parameters, which are only intra-procedurally defined and manipulated. That means they exist only in the scope of the body of the method the reflection call is part of and are not influenced by the caller given parameters. Therefore a small analysis of the CFG is required to resolve these.
The worst and most complex kind of reflection invocations are **context-sensitive reflections**. These

have the same charactaristics as locally resovlable ones with the difference that they are inter-procedurally resolvable. Following that the resolution is based on the analysis of the CFG for the whole project which can be complicated, inefficent and time consuming.
**Method handles** appear on calling any reflection call on *java.lang.invoke.**. Here the reasons stated above lead to unsoundness but worse, because the invocation of a method is unknown and a bigger part of the call graph is affected.

Talking about the resolution of these, not every source of unsoundness, caused by the reflection library, is statically resolvable. It was shown that reflection calls are resolvable in 93% of all cases. The other 7% of cases cannot be resolved by any static analysis. This happens for example the moment any parameter of a reflection call is dependent on user input. User input cannot be predicted or calculated. Therefore these sources will never be resolved[1].

## 3.2 Exploits through reflections

The reflection library contains a variety of different attack vectors which can be used by a malicious actor to obtain hidden information. The most severe ones are ”insecure use of introspection”, ”type confusion” and ”buffer overflows”.

Although the reflection library calls have restrictions, they can lead to **insecure use of introspection** when system classes are involved.If trusted system classes are combined with untrusted input, certain reflection calls can quickly become a bypass around security managers. These bypasses open up access to members of the program that are normally restricted[7].

Similar to the restrictions, Java has a strong type system by design. **Type confusion** still manages to break that type system. By mistakenly handling an object of type A as type B (through reflective casts, for example), a malicious actor is able to perform actions on the object of type A that are only allowed for objects of type B. This ultimately also causes access to members of the object that are restricted[7].

**Buffer overflows** are read or write operations, which go beyond the bounds of the memory allocated for that specific data structure. The consequences differ to the respective location of the memory part. Affected parts can be inside the heap or the stack of the program.
Being in the heap, where sensitive variables could be stored, a **buffer overflow** yet again grants access to restricted members of the program breaking Java’s information hiding as well.
Contrary memory affected in the stack is able to overwrite the return address, which cleverly used, brings the ability to manipulate the control-flow of the program. While not breaking Java’s information hiding this is potentially way more severe[7].
In the worst case an attacker is able to open a shell and corrupt the running system. It should be mentioned here that there are already a lot of countermeasures for takeovers of the control-flow graph because these exploits are such a powerful tool.

Although reflection calls are inspected in this thesis as a reason for unsoundness, which possibly hides exploits through FNs, the reflection library is still open to other attack vectors. This adds more relevance

to the results and metrics found in this study, which also strengthens the importance for static analysis to take reflections into account.

## 3.3 Consideration of reflection in static analysis

Most reflection calls are resolvable as shown in the previous section. But only a few tools actually handle reflection calls to some extent[1]. The most mentioned static analysis tools in scientific research are DOOP[3], ELF[4] and SOOT[15]. In the study[9] done by Davy Landmann et. al. a total of 20 static analysis tools based on scientific research were analysed. Landmann's results show, contrary to the statement made before, that all of them handle reflection calls to a varying degree. DOOP being the best peforming tool only vulnerable to the analysis of Strings, which are the main reason for the occurrence of unresolvable calls. Therefore DOOP is at the moment of writing the best tool handling reflection calls. Sensitivity to fields and the fixedpoint analysis are the least respected features across all tools. The data ultimately suggests that the trade-off between scalability and precision is handled very differently between all analysed tools. None has the weakness or strengths of another. Every tool varies to some degree. To conclude the found statistics, reflection calls are present and taken into account in the most used static analysis tools. Other research suggests that there are tools not considering reflections and their unsoundness at all[1]. The tools treating reflection calls, besides DOOP, do ignore some of the definitely resolvable calls.

## 3.4 Consideration of reflection in dynamic analysis

TamiFlex[2] is a dynamic analysis tool to support any static analysis. The tool provides more soundness to the overall analysis in respect to a set of program executions. TamiFlex uses a "Play-out Agent", which logs all classes that are loaded during runtime of a specific run. The agent also updates that same log file on every run that is done, improving accuracy and precision. These runs are done until no changes to the log files appear. The suggested usages of those logs by the research[2] are to refactor the existing program by inserting regular method calls for the reflection invocations. Following that, the resulting program can be analysed by existing static analysis tools with a guaranteed higher precision and soundness.
The comparison[10] of different dynamic analysis tools, including TamiFlex, shows that they are effective but face challenges at the same time. Just as static analysis has a trade-off between scalability and precision, dynamic tools have one between code coverage and analysis overhead. These challenges need to be mitigated to be an effective support for static analysis.
An honorable mention is that some tools propose monitoring dynamic language features, which fits the narrative of the study presented in this thesis[10]. All in all, there are existing dynamic analysis tools that improve precision and support static analysis but do not lead to completeness.

# 4 ReflectDetect: Design & Implementation

To achieve a relevant inspection of reflection patterns and their underlying methods, real-world projects need to be analysed statically for appearing usages of the reflection library. Existing tools, as shown previously in related work, do consider reflection calls during computation but do not explicitly collect data about them. Therefore a new analysis, called ReflectDetect, is created for this upcoming study. ReflectDetect is written in scala using the static analysis tool chain OPAL[5].

## 4.1 Requirements

ReflectDetect is a static analysis tool that systematically searches for calls to the reflection library that are able to cause unsoundness inside other static analysis tools and their respective call graphs. ReflectDetect was specifically designed and created to analyse Java projects for the study presented in this thesis. Therefore, ReflectDetect must meet the following requirements:

| | |
|---|---|
| **Precision:** | ReflectDetect finds only relevant and reachable calls to the reflection library, which cause unsoundness. |
| **Soundness:** | ReflectDetect is sound and considers every possible reflection call that is able to cause unsoundness. |
| **Edge case:** | ReflectDetect is allowed to be unsound in a certain edge case. ReflectDetect uses a call graph to only take reachable reflection calls into account and fulfill precision. By using a call graph, in this case computed by the RTA algorithm, the call graph may not be sound. Following that, some sources of unsoundness are hidden through others already included in the call graph. Those calls not included in the set of reachable methods do not reduce the precision of ReflectDetect, since the root of the unsoundness is detectable. If the detectable sources of unsoundness are found and resolved in the future, ReflectDetect can be run again over the newly created call graph and find some of the previously hidden sources. That way every source of unsoundness can be iteratively found and resolved. |
| **Completeness:** | A static analysis fulfilling precision and soundness is considered complete. |
| **Semi-Completeness:** | ReflectDetect fulfills completeness excluding the unsound edge case. That requirement will be called semi-completeness in this thesis. |

## 4.2 Result production & representation

The results cover necessary data to be examined for the occurrence of reflection patterns and to be used for future work. The results are extracted from ReflectDetects data domain and represented as a String as follows:

Firstly, it contains three different values for the amount of classes inside the Java project. The first value shows the amount of classes in general inside the project. The second value gives information about the amount of classes having any unsound reflection call in any of their methods bodys, which has to be reachable. The third and last value represents the amount of classes reaching reflections only transitively, which means they are reaching classes described in the second value through any of their methods but do not have unsound reflection calls anywhere of itself.

Following that, three different values for certain counts of methods are added. Here the first value contains the general amount of methods inside the project. The second value contains the amount of methods that are reachable trough the entry point being equal to the amount of methods inside the call graph. Ending with the third value representing the amount of methods containing at least one reflection invocation inside their body.
These metrics are necessary and generally used to derive statistics about the usage of the reflection library across all projects analysed in the study. They also give an overview of how much a certain project is negatively affected by the reflection library and makes comparisons between different projects easier.
Following that every class that reaches a reflection call directly gets information added to the result. That includes general metrics about itself and more details about its methods, which have reachable reflection methods inside their body.
The insight about methods includes their name, the count of reflection calls inside their body, all dynamic reflect features appearing in their body and information about reflection calls.
The added information about each reflection call contains their class type, name, parameter types, program counter, line number and dynamic feature category. Before the found reflections are added to the result they are sorted by their program counter for easier pattern analysis. The program counter is also important for future work. More on that in its respective section.

## 4.3 Design

ReflectDetect consists of ten classes total. Three of those are inside the service domain handling the overall analysis, computation and result production. Six of them are inside the data domain, where all relevant data during computation is stored. After the computation has finished the final state is used to create the result. The final class, called **RootService**, connects both domains so the service domain can read and adjust the data domain at any time.

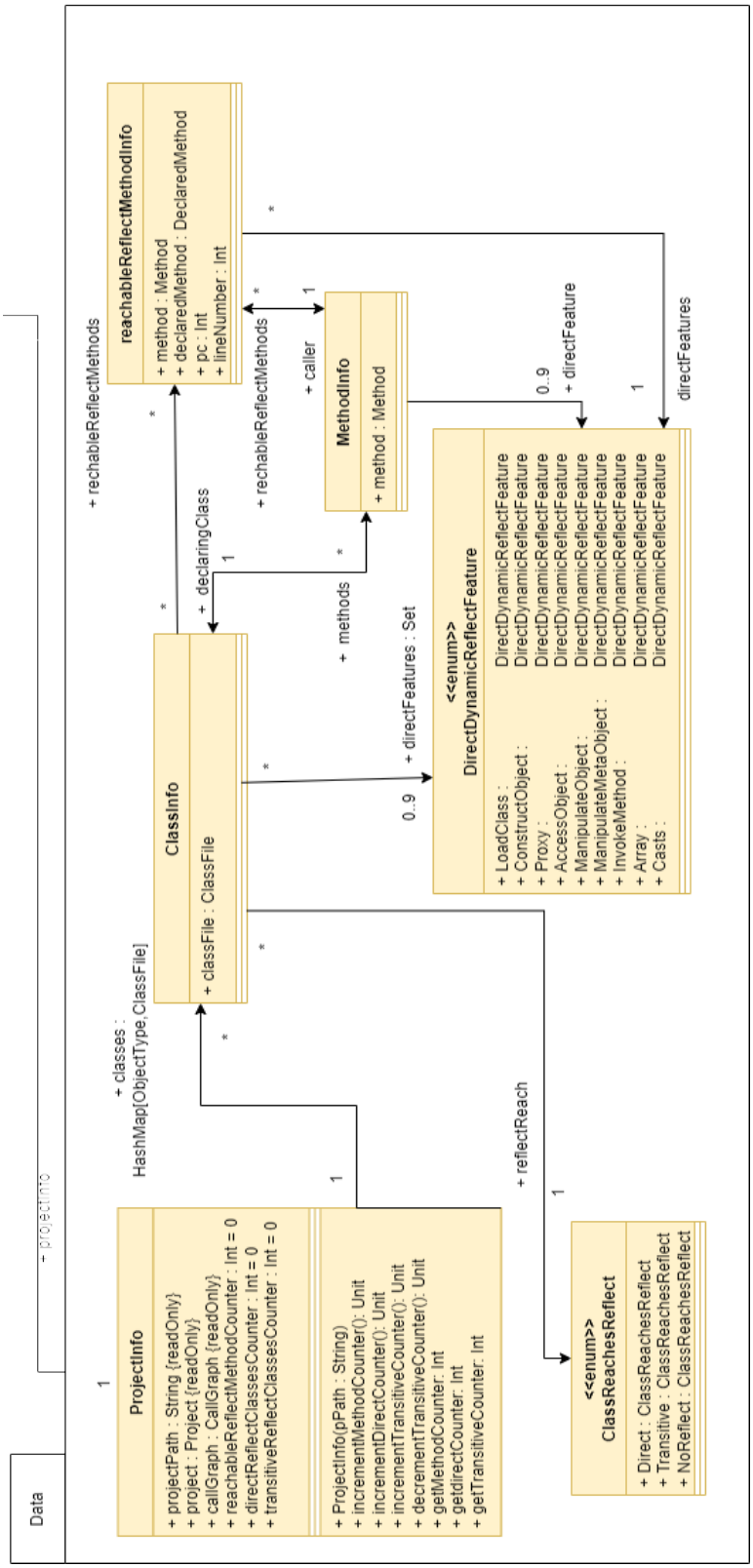


Figure 1: Data domain of ReflectDetect

The classes in Fig:1 store all necessary information during computation as already mentioned.
**ProjectInfo** stores basic information about the project, especially the call graph, a hash map containing ClassInfo objects for each class and the counters for different reflection semantics.
Each **ClassInfo** object holds the corresponding classfiles, a hash map containing MethodInfo objects for each related method, another hash map for every reachable reflect method, a set of reflect features which appear in any method of this class and an enumerator flagging how the class reaches any reflective call.
**MethodInfo** stores basically the same information as ClassInfo for its body but instead of holding the enumerator for the way it reaches a reflective call, it saves the class it is declared in. That is used to trace back the origin of its reachable reflections to add information to the correct ClassInfo object during analysis.
**ReachableReflectMethodInfo** is the key data structure of the whole analysis. Objects of this class contain every necessary data about each reachable reflection call. That includes the program counter, line number, Method and DeclaredMethod (OPAL differentiates here) as well as its related dynamic feature type.

For storing all these objects hash maps and sets are used, because of their advantageous properties. The hash map is very efficient in adding and searching elements, which is important for analysing very big projects. The only backlash are the keys needed to access the stored information, which are known at any moment in ReflectDetect by design. The set is used for the features because it does not allow duplicates. That way it is very easy to just add every found feature to the correct set without checking whether it is already part of it or not. That also improves performance and the complexity of ReflectDetect.
Lastly the two classes left are the mentioned enumerators. **ClassReachesReflect**, as the name suggests, flags whether a class reaches reflection directly, transitively or not at all. The last flag is considered the default state every ClassInfo object is initialised with. **DirectDynamicReflectFeature** contains the nine different feature types. More on those in its respective section.

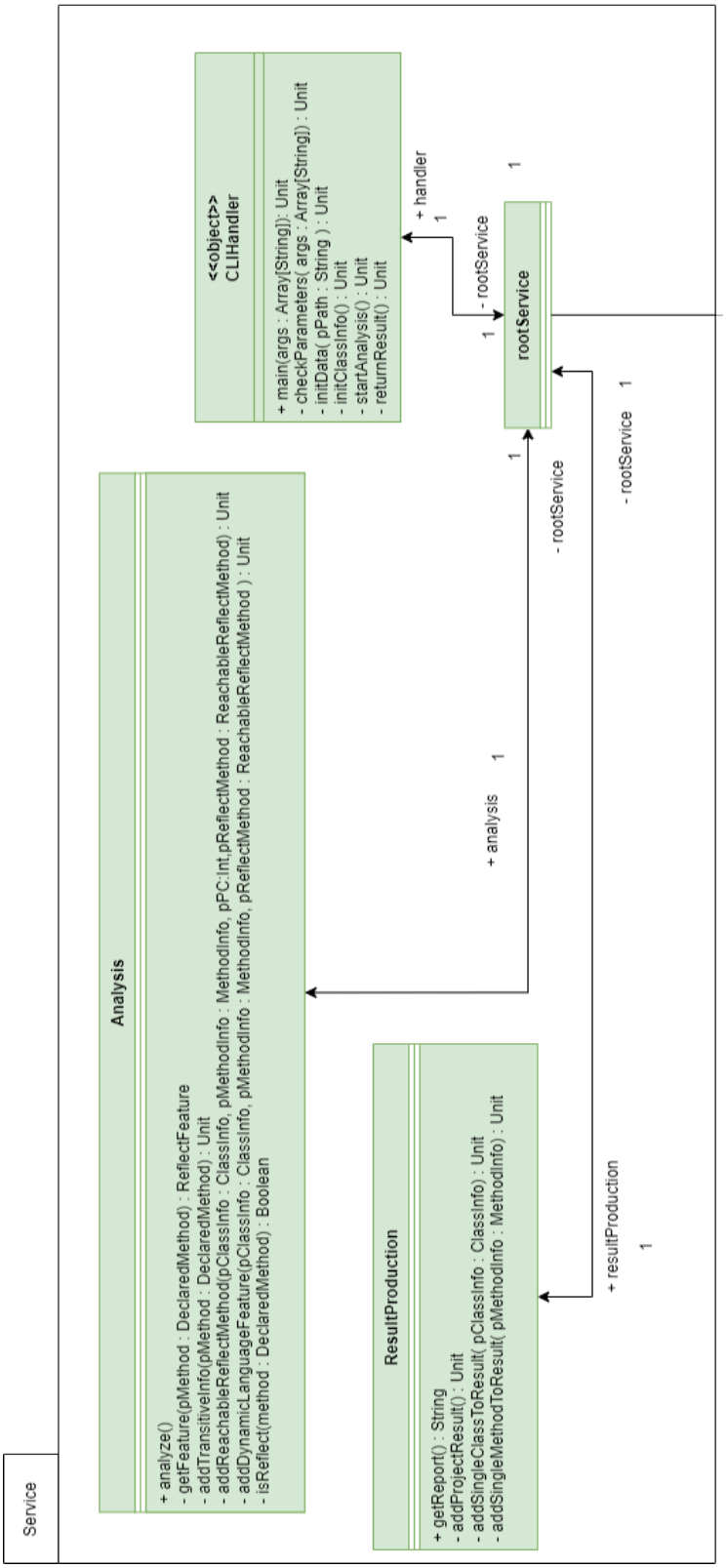


Figure 2: Service domain of ReflectDetect

The classes in Fig:2 only contain methods used for the analysis. Each class has its own semantic work space. **CLIHandler** contains the main method of ReflectDetect and therefore the entry point. Correctly started from any command line interface it checks the given program argument, which should be a path to the Java project being analysed, and initialises the data domain. After that initialization, the CLI handler kicks off the analysis and then the result calculation. The result is written into ”output/[ProjectName].txt”. If the folder ”output” does not exists, ReflectDetect creates one.

The class **Analysis** uses and manipulates the data domain until every reachable method is considered. This will be further elaborated in an upcoming section.

Finally the class **ResultProduction** creates the result string. That string will be returned to the CLI-Handler to be logged into a text file. First all basic information about the project is added to the string like the name of the project, as well as the mentioned metrics about classes and methods. Next the ResultProduction takes the final state of the data domain and iterates over every class. Classes that do not reach reflections at all are ignored. Classes that are reaching reflections only transitively are added and only mentioned in the result string for future work. Classes reaching reflections directly are added to the string with their name, amount of reachable reflections and all dynamic features appearing in any of its methods. To fully disclose every necessary information about the class, ResultProduction iterates over all methods and adds information about those containing at least one reflection. That data includes the method name, amount of reachable reflections and all appearing dynamic features. For each method ResultProduction lastly iterates over each appearing reflection inside the body and adds their respective information, which includes their method descriptor, program counter, line number and dynamic feature category. These are added in ascending order by sorting them by program counter. That is important to examine patterns and for future work. The result produced by ResultProduction is exemplified in Fig:3.

```
Project: jackson-databind-2.17.0.jar
Classes contained in this project: 30812
Classes containing reachable Reflections count: 24
Classes reaching reflection calls only transitively count: 100
Methods contained in this project: 252397
Reachable methods contained in this project: 11916
Methods containing reachable Reflections count: 49
----------------------------
Class: ObjectMapper
reachable Reflect Calls: 0
Class only reaches reflection calls transitively
----------------------------
Class: TypeFactory
reachable Reflect Calls: 3
Class reaches Reflection Calls directly
Direct Features:
Array which is Dynamic
LoadClass which is Dynamic
Methods of TypeFactory:

Method: rawClass
Reachable Reflect Methods Count: 1
Reflect Features:
Array which is Dynamic
Reachable Reflect Methods:

Call of java.lang.reflect.Array{ public static java.lang.Object newInstance(java.lang.Class,int) }
at PC = 47
at line 334
Feature Category: Array which is Dynamic
```

Figure 3: Excerpt of the result for the project ”jackson databind”

## 4.4 Analysis

The overall static analysis of a single project is the key part of ReflectDetect. A given project is inspected under fulfillment of semi-completeness and the data is collected. Therefore, if the detection of reflection calls is handled correctly, ReflectDetect will find all invocations significant in the context of unsoundness by design. The exact detection is presented in the following section.

### 4.4.1 Approach

The framework OPAL[5], written in scala, is used for the whole static analysis within ReflectDetect. OPAL is a ”highly configurable software product line”[5] for static analysis, meaning it is able to load and interpret any Java project, manipulate and examine its bytecode and every single statement of the program. It is important to note that the call graph being used for the analysis is computed by the RTA implementation of OPAL and therefore its soundness is dependent on that.

| Dynamic Language Features | | |
|---|---|---|
| **Category** | **Definition**<br>by Davy Landman et. al.[9] | **Logical expression for detection**<br>Method descriptor contains... |
| Load Class | Entry to the Reflection API, returns references to meta objects from a String. Considered harmful since it can execute static initializers. | ”forName(” ∨<br>”loadClass” |
| Construct Object | Create a new instance of an object, equivalent to the new <ClassName>() Java construct. | ”newInstance(java.lang.Object” ∨<br>”newInstance()” |
| Proxy | Proxies are fake implementations of interfaces, where every invoke is translated to a single callback method. Very harmful for static analysis, since there is no static equivalent for this feature. | ”getProxyClass” ∨<br>”getInvocationHandler” ∨<br>”newProxyInstance” |
| Access Object | Read the value of an Object’s field. Equivalent to the obj.field Java construct. | ”Field” ∧ ”get” ∧ ”java.lang.Object” |
| Manipulate Object | Change the value of a field. Equivalent Java construct: obj.field = newValue | ”set” ∧ ”(java.lang.Object,java.lang.Object)” |
| Manipulate Meta Object | The only mutable part of the API: changing access modifiers. | ”setAccessible(boolean)” |
| Invoke Method | Invoke an method. Equivalent Java construct: recv.method(args). | ”invoke(” |
| Array | Create, access, and manipulate arrays. | [”set” ∧ ”(java.lang.Object,int,java.lang.Object)” ] ∨<br>[”newInstance” ∧ ”java.lang.Class” ∧ ”int”] ∨<br>[”get” ∧ ”java.lang.Object,int”] |
| Casts | Cast to a dynamically Class meta object. Equivalent Java construct: (Class)obj | ”cast(” |

Figure 4: Dynamic Reflect Feature Categories potentially leading to unsoundness on usage [9]

```
private def isReflect(method: DeclaredMethod): Boolean = {
    method.declaringClassType.toString().contains("java/lang/reflect") ||
    method.declaringClassType.toString().contains("java/lang/Class") && (
        method.toJava.contains("forName(java.lang.String)") ||
        method.toJava.contains("forName(java.lang.String,boolean,java.lang.ClassLoader)")
        || method.toJava.contains("cast(java.lang.Object") ||
        method.toJava.contains("java.lang.Object newInstance()")
        ) ||
    method.declaringClassType.toString().contains("java/lang/ClassLoader") &&
    method.toJava.contains("loadClass(java.lang.String)")
}
```

Figure 5: Code snippet 1 from the class Analysis

To identify the appearance of reflection methods and ultimatively patterns consisting of at least two of those methods, the analysis first of all needs to find every reachable reflect method and then check for it to be a severe call in regards to unsoundness, belonging to one of the dynamic feature categories.

Ensuring to be as precise and sound as possible, ReflectDetect iterates through every reachable method, computed by the call graph, and invokes the private method "isReflect(method: DeclaredMethod)". Two cases return as true. In case one, it simply extracts the declaring class type as a string and if it contains "java/lang/reflect" it is one of many reflect methods. If that check fails, case two is considered. Here edge cases that lead to unsoundness by definition of the dynamic language features, which are outside the scope of the reflect library, are treated. That includes checking the declaring class type string to contain "java/lang/Class" or "java/lang/ClassLoader". The result is connected with a logical "and" to a comparison of the method descriptor string with relevant method descriptors leading to unsoundness. A snippet showing the implementation can be seen in Fig:5

On a true return value, all necessary and additional information is computed and added to the data domain for each found reachable reflection call. This is done by calling "addReachableReflectMethod(...)", which handles all necessary invokes and manipulates the data domain via "rootService".

### 4.4.2 Dynamic Language Features

To add more semantic details to found reflection calls, "dynamic language features"[9] as a categorisation are used. These categories also help differentiating two reflection methods as well as two patterns from each other. The scope of methods for these features also includes a few outside of the reflection library coming from *java.lang.{Class,ClassLoader}*, which will also be considered in ReflectDetect. For this thesis those are also meant while speaking of reflective calls, which come from the reflection library.

```
private def getFeature(pMethod: DeclaredMethod): Data.DirectDynamicReflectFeature = {
    val javaRep = pMethod.toJava
    ...
    } else if ((javaRep.contains("set") &&
                javaRep.contains("(java.lang.Object,int,java.lang.Object)"))
               ||
               (javaRep.contains("newInstance") && javaRep.contains("java.lang.Class") &&
                javaRep.contains("int"))
               ||
               (javaRep.contains("get") && javaRep.contains("java.lang.Object,int"))) {
        return eArray
    } else ...
    ...
    return default
}
```

Figure 6: Code snippet 2 from the class Analysis

To get to the dynamic language features, using the in Fig:4 mentioned logical expressions, every public method is evaluated and divided into sets.
One set consists of only statically resolvable methods being categorized as "support methods". These are not severe or relevant for unsoundness but important nonetheless for the library to work and be used correctly.
The other set consists of only dynamically resolvable methods, which are categorized as "dynamic language features". All of these methods are potentially only resolvable during runtime which directly impacts the call graph to be unsound and therefore also the analysis.
Every method, according to their set, gets one subcategory, which further describes their semantic meaning inside a program.
Because of the small relevance for unsoundness and due to time constraints, support methods are not considered in ReflectDetect.

During the process mentioned in 4.4.1, "getFeature(pMethod: DeclaredMethod)" is invoked to pair the currently analysed reflect method with the fitting dynamic feature category. The exact detection of every feature is shown in Fig:4 and a code snippet exemplifying the exact implementation in Fig:6. In simple terms, for each feature a logical expression must be fulfilled in which the method descriptor has to contain certain substrings.
These substrings inside the logical expressions of Fig:4 are based on the scope of all method substring defined in the paper "Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study" by Davy Landmann et. al. covering all methods inside the reflection library, which is shown in Fig:1 of their paper. The longest common substrings of all method substrings, belonging to the same category, were used to build these logical expression.

```
    ...
    callGraph.callersOf(pDeclaredMethod).iterator.foreach(context => {
          if (classMap(context._1.declaringClassType).reflectReach == NoReflect) {
            classMap(context._1.declaringClassType).reflectReach = Transitive
            rootService.projectInfo.incrementTransitiveCounter()
            list.addOne(context._1)
          }
    })
    while (list.nonEmpty) {
          val current = list.remove(0)
          callGraph.callersOf(current).iterator.foreach(context => {
            if (classMap(context._1.declaringClassType).reflectReach == NoReflect) {
              classMap(context._1.declaringClassType).reflectReach = Transitive
              rootService.projectInfo.incrementTransitiveCounter()
              list.addOne(context._1)
            }
          })
    }
}
```

Figure 7: Code snippet 3 from the class Analysis

### 4.4.3 Direct & transitive reachability

To derive the already discussed metrics and general information about a project, the number of classes reaching reflection calls directly and transitively needs to be tracked. The difference between direct and transitive reachability is shown in section 4.2. The tracking is done by adding specific enumerators to each classInfo object. Every classInfo object is initialized with the default enumerator representing that this class does not reach a reflection call at all. Upon changing an enumerator during computation, counters for the number of direct and transitive classes are either incremented or decremented.
In general every time a reflect method is detected, the fitting ClassInfo object declaring the caller gets its reachability enumerator set to direct. Following that, the callers are traced back through the call graph by calling the method ”addTransitiveInfo(pDeclaredMethod: DeclaredMethod)” with the current caller of the reflect method as the parameter.

The first version of that was a recursive solution. It took an iterator of all callers of the given declared method and called itself on each of them. In every invoke it checks the corresponding ClassInfo object to the declaring class type of the current given declared method and if it has the default reachability enumerator it is set to ”transitive”. In most projects analyzed this led to multiple stackoverflow errors.

The second and final solution used the recursive rules and changed the implementation to fit the concept of dynamic programming. Instead of calling itself on every caller, the callers are temporarily stored in a data structure. After one is analysed to completion and its callers are added to the data structure, the current one is removed. The moment the data structure is empty, the calculation of classes reaching the reflection transitively is finished. The implementation can be examined in Fig:7.

The second solution may take the same overhead of storage during computation as the first one, but it finishes smoothly in a very fast time because of the efficient design of the Scala language and the larger capacity of the heap compared to the stack.

## 4.5 Framework testing

A Java project[1] was created to benchmark ReflectDetect in finding and correctly evaluating every reflect feature discussed in this thesis with their different appearances. It also contains classes reaching these reflections directly, only transitively, or not at all to guarantee that the reachability categories are correctly set as well.
That is achieved by creating a class for each feature category. Those classes, which may use further classes, are instantiated one by one inside the main method representing the entry point of that project. By doing that the relevant reflect methods to the respective features are prepared and finally called. All of that is handled by the constructors of the feature testing classes. An example from the project is observable in Fig:8.
The print statements in the example and other classes were used to verify that reflect methods are invoked correctly during runtime. Finally, ReflectDetect ran on the test project and produced a result text file. That file contained every reflect method presented in Fig:4 with correct information such as feature category, declaring class, line number, and more. In addition, it correctly stated the reachability characteristics of each class. Therefore, it can be assumed that ReflectDetect correctly finds and evaluates all reflections, which may lead to unsoundness, under the previously discussed constraints and requirements.

## 4.6 Limitations

### 4.6.1 Observable patterns

All reflect patterns that can be derived from the result files are intra-procedural, which means inside the semantic space of the body of just one method. For inter-procedural patterns, this constraint does not exist. In the intra-procedural semantic space, patterns are able to appear all over the project between different methods and even classes. To still fulfill the definition of a pattern, either the program counter needs to be tracked, which requires a dynamic analysis, or the control-flow graph needs to be analysed for close nodes containing reflection calls. The second and only viable option for static analysis is very complex and time intensive to implement. Due to time constraints it was not implemented in ReflectDetect.

### 4.6.2 Parameter severity

To resolve unsoundness, parameters of the called reflection methods need to be known. To achieve this in static analysis, the control-flow graph needs to be analysed until instantiation of each parameter. Knowing the expected amount of work that needs to be done for each found reachable reflection to possibly having a chance of resolving their unsoundness gives an approximation on the complexity of unsoundness trough reflections. On a side note, even tracking a parameter back to its declaration does not guarantee

[1] `https://github.com/lu25ca09/ReflectDetectTestProject`

```java
package InvokeMethod;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class InvokeMethodFeatures {
    public InvokeMethodFeatures() {
        System.out.println("\n----------------------\nInvokeMethodFeatures: ");
        Method[] dummyMethods = DummyInvoke.class.getDeclaredMethods();
        DummyInvoke dummyInvoke = new DummyInvoke();
        for (Method method : dummyMethods) {
            method.setAccessible(true);
            System.out.println(method.invoke(dummyInvoke));
        }
    }
}
```

```java
package InvokeMethod;

public class DummyInvoke {
    public static String printPublicStatic(){return"Hello public Static Invoke";}
    private static String printPrivateStatic(){return "Hello private Static Invoke";}
    private String getPrivateString(){return "Hello private Invoke";}
    public String getPublicString(){return "Hello public Invoke";}
}
```

Figure 8: Code used for testing the category InvokeMethod

a resolution, for example if the parameter is assigned with user input. To give a general idea of that, parameter severity could be used.
It differentiates whether a parameter is resolvable directly, because it is a constant, locally, which means intra-procedurally resolvable, or globally, which means only inter-procedurally resolvable[11].
Because of its complexity and need of a whole static analysis for the control-flow graph, time constraints did not allow ReflectDetect to cover that although I tried implementing the detection.

### 4.6.3 Static reflect features

Covering static reflect features in detection would potentially lead to a lot more found and complex patterns. Reflect methods not causing unsoundness but being called to support those that do could also be useful to distinguish found patterns in more detail from each other. It also could lead to even more patterns being found, because single found reflection methods are now considered a pattern. Paired with methods inside the static feature set, some of them could be evaluated as patterns.
Although the groundwork for static features is already inside the codebase of ReflectDetect (enumerators are already declared) it was not possible to implement the detection due to time constraints. Nonetheless, because of ReflectDetect's modular design, this could easily be added in the future.

# 5 Study

The study aims at classifying the importance of the reflection library for existing static analysis tools. It collects metrics about how strongly the projects are affected by reflection invocations in general. In detail the same found calls to the library are evaluated for the existence of certain patterns. Dependent on the results, patterns may imply a more complex handling of the reflection library in the future.

## 5.1 Method

A set of real-world Java projects are analysed using the static analysis ReflectDetect, which is introduced in this thesis. The results of that analysis are used for general statistics about the quantity of calls to the reflection library, which could cause unsoundness in call graphs. These results are additionally evaluated for the occurrences of method call combinations, called patterns, which may be a common use case of the reflect library. The results in this study are representative for real-world Java projects because ReflectDetects fulfills semi-completeness.
ReflectDetect was run manually on every project inside the IDE IntelliJ without an automation. An automation could be done in future work by either using a shell or extending the current implementation. The produced text files containing the results were also analysed manually. Additional code in ResultProduction could reduce the number of lines in the results as well. Excluding methods that only have one reachable reflection call would be one possibility to achieve that. These methods are still included on purpose because that gives a good overview of the reflection methods used most of the time. Nonetheless the results need to be evaluated manually to correctly exclude ”patterns” where the methods have no semantic connection, mostly depending on the pcs, line numbers and the methods themselves.
The results are collected in an Excel spreadsheet. Each project has its own table connected with formulas in an overview table. The spreadsheet also contains a template table with already found patterns. Therefore the spreadsheet can be reused or filled with more projects in the future. To improve accuracy, a broader amount of projects can be added at any time.
The mentioned files can be found on the github page of ReflectDetect[12].

## 5.2 Chosen Java Projects

A total of 16 projects were analysed. In Fig:9 all analysed projects can be seen. Five of which are taken from github[1] sorted by projects that are implemented in Java and listed with a decent number of stars, implying that they are used enough to be relevant. If a JAR file was not easily extractable, the project was not considered. The other eleven projects are taken from maven[2] sorted by the highest usages.
The version of each project used for the analysis was selected by the highest usages as well. In general,

[1] https://github.com
[2] https://mvnrepository.com/popular

| Project | Version | Source |
|---|---|---|
| dubbo | 3.2.12 | github |
| mybatis | 2.5.11 | github |
| okhttp | 4.12.0 | github |
| retrofit | 2.9.0 | github |
| rxjava | 3.1.8 | github |
| org.assertj.assertj-core | 3.25.3 | maven |
| com.google.code.gson | 2.10.1 | maven |
| com.google.guava | 33.1.0-jre | maven |
| org.apache.hadoop.hadoop-common | 3.4.0 | maven |
| com.fasterxml.jackson.core.jackson-databind | 2.17.0 | maven |
| junit | 4.13.2 | maven |
| org.apache.logging.log4j.log4j-core | 2.23.1 | maven |
| org.projectlombok.lombok | 1.18.30 | maven |
| org.mockito.mockito-core | 5.11.0 | maven |
| org.slf4j.slf4j-api | 2.0.13 | maven |
| gov.nasa.worldwind | 2.0.0 | maven |

Figure 9: Java Projects included in this study

filtering by usages gives the analysis more relevance. The more dependencies to an affected project exist, the higher are the chances their unsoundness disrupts future static analysis by throwing FNs.

## 5.3 Results

ReflectDetect collects data about the amount of classes and methods having any reflection invocation inside of them. Details about these invocations are collected as well. With the gathered amounts for each project metrics, like an average, are calculated. These metrics are used to give an overview of the relevance of the reflection library in real-world projects. The details about each appearance of an invocation are evaluated to determine the complexity for usages of the reflection library. In the following these results are presented.

### 5.3.1 Project Statistics

Overall, the parts of projects affected by reflections seem very little. On average only 0.42% of classes and 0.81% of reachable methods are affected by reflection calls belonging to the dynamic feature category, which can be seen in Fig:10. Although that number may appear very low, in absolute numbers it is still quite relevant. Every project has classes in the five-digit range starting from 30,079 to 33,758. Reachable methods on average are in the five-digit range as well starting from 631 to 36,492, where only one project was outside that range explaining the minimum.
Altogether that breaks the number of affected classes down to 131 and affected reachable methods down to 93 on average.

| | Min | Max | Median | Average |
|---|---|---|---|---|
| Amount of classes in each project: | 30,079 | 33,758 | 30,982 | 31,199 |
| Classes affected by reflection directly: | 0.01% | 0.26% | 0.09% | 0.10% |
| Classes affected by reflection generally: | 0.01% | 2.55% | 0.21% | 0.42% |
| Amount of methods in each project: | 243,596 | 271,254 | 251,728 | 252,669 |
| Amount of reachable methods in each project: | 631 | 36,492 | 10,343 | 11,520 |
| Methods affected by reflection in general: | 0.00% | 0.06% | 0.02% | 0.02% |
| Reachable methods affected by reflection in general: | 0.04% | 3.65% | 0.51% | 0.81% |

Figure 10: Statistics for the 16 analysed Java projects

### 5.3.2 Common Patterns

The 16 analysed Java projects in this study have a much higher appearance of reflection methods outside the definition of a pattern. That means most affected method bodies contain only one reflection call or certain edge cases. These edge cases describe the appearance of multiple reflection invocations inside a body without any semantic connection. They can be found multiple times across the results of all projects. Most of the time these edge cases consist of calling the same reflection method multiple times, which is exemplified in Fig:11.

```
Method: initStartTime
Reachable Reflect Methods Count: 2
Reflect Features:
InvokeMethod which is Dynamic
Reachable Reflect Methods:

Call of java.lang.reflect.Method{ public java.lang.Object invoke(java.lang.Object,java.lang.Object[]) }
at PC = 23
at line 56
Feature Category: InvokeMethod which is Dynamic

Call of java.lang.reflect.Method{ public java.lang.Object invoke(java.lang.Object,java.lang.Object[]) }
at PC = 52
at line 60
Feature Category: InvokeMethod which is Dynamic
```

Figure 11: An example of close reflections calls not considered to be a pattern appearing in log4j

That happened for example with the *Method.invoke(Object, Object[])* method, which invokes the method of the object in the parameter.
One case describes the amount of different calls between two reflection invocations as being too high. That is distinguishable by a very big difference in their pc values or line numbers. The higher the space between them, the less likely is a semantic correlation and therefore the existence of a pattern. To make sure these are not correlated at all, either the CFG or the code itself needs to be inspected, which I leave for future work.

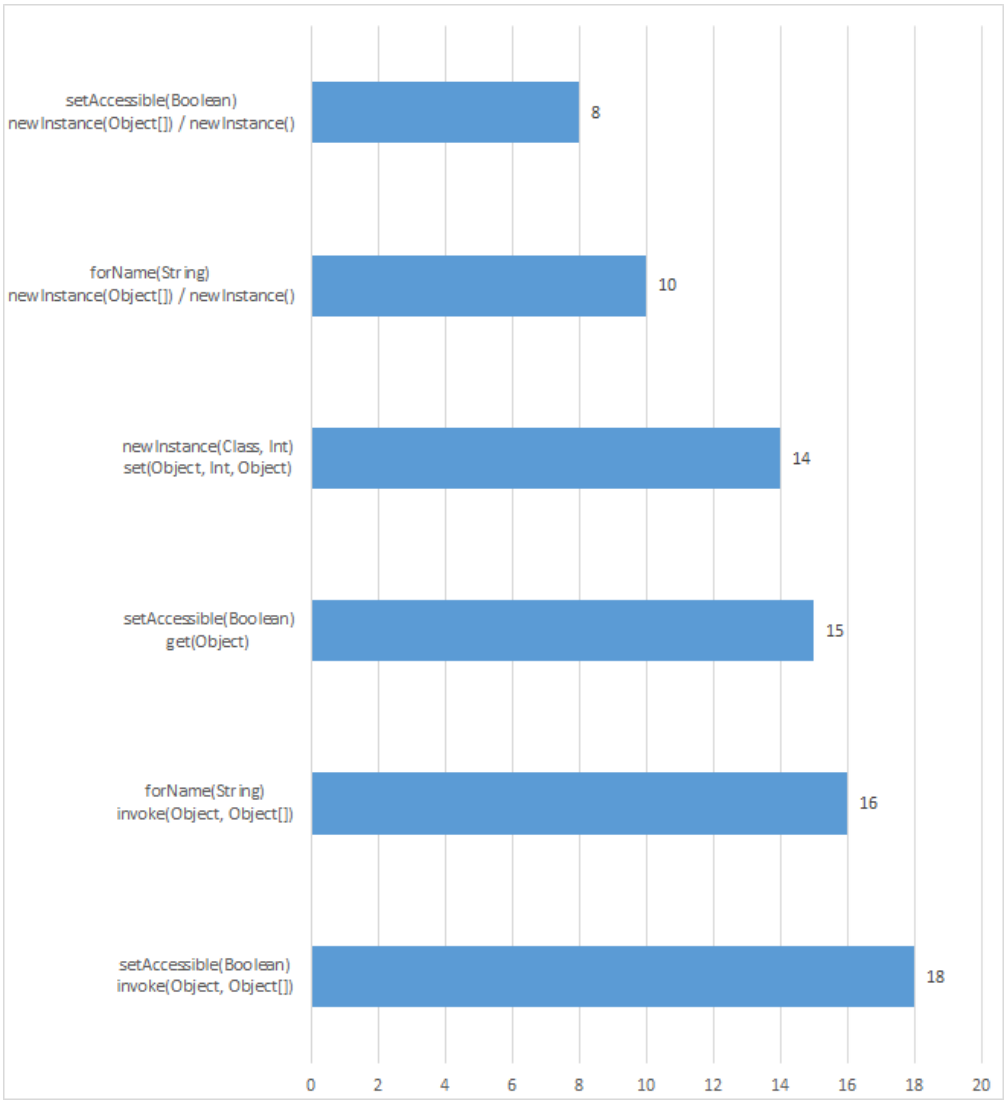


Figure 12: Patterns of specific methods appearing more than 7 times across all projects

The last case is an obvious semantic divergence. This happens when two or more reflection calls are close, meaning their pc values are close to each other, but cannot affect each other in any way. For example one method call manipulates an array and the other looks up a class name afterwards. The probability that those affect each other is very low.

Those edge cases are very much present, but the amount of reflection calls appearing alone still dominates. That suggests the reflection library is mostly used for small workarounds to avoid refactoring of the codebase. Further it suggests that the library is not used for complex constructs or big parts of a project. That benefits the resolution of unsoundness in the future, because the computation will be less

complex based on fewer dependencies between sources of unsoundness.

Talking about the complexity of future resolution of unsoundness, patterns will be a more severe case. Every mentioned pattern is ordered. Ordered means that every line placed above another inside one pattern is called first when it comes to execution. In Fig:12 the most common combinations of reflection calls that potentially correlate are shown. Half of the found patterns there start with manipulating access rights on a meta object, by calling *setAccessible(Boolean)*, followed by invoking methods, manipulating objects, their fields and data structures as well as creating or inspecting them. This also fits the narrative of small workarounds instead of big constructs because it is way easier to quickly change access rights on a tiny bit of a meta object than to refactor the codebase to achieve the same. The two most used patterns both contain the call of the method *Method.invoke(Object, Object[])*. One pattern loads a class first, probably where the to be invoked method comes from, and the other manipulates access rights beforehand, most likely to invoke methods the objects access is restricted on.
Those, in Fig:12 shown, most frequent patterns come down to 98 patterns total in all 16 projects. That is a very small number compared to the roughly 1500 affected reachable methods across all of them. That number is an approximate sum of affected reachable methods in all projects based on the previously shown averages:

$$0.81\% \cdot 11,520 \cdot 16 = 93.312 \cdot 16 = 1,492.992$$

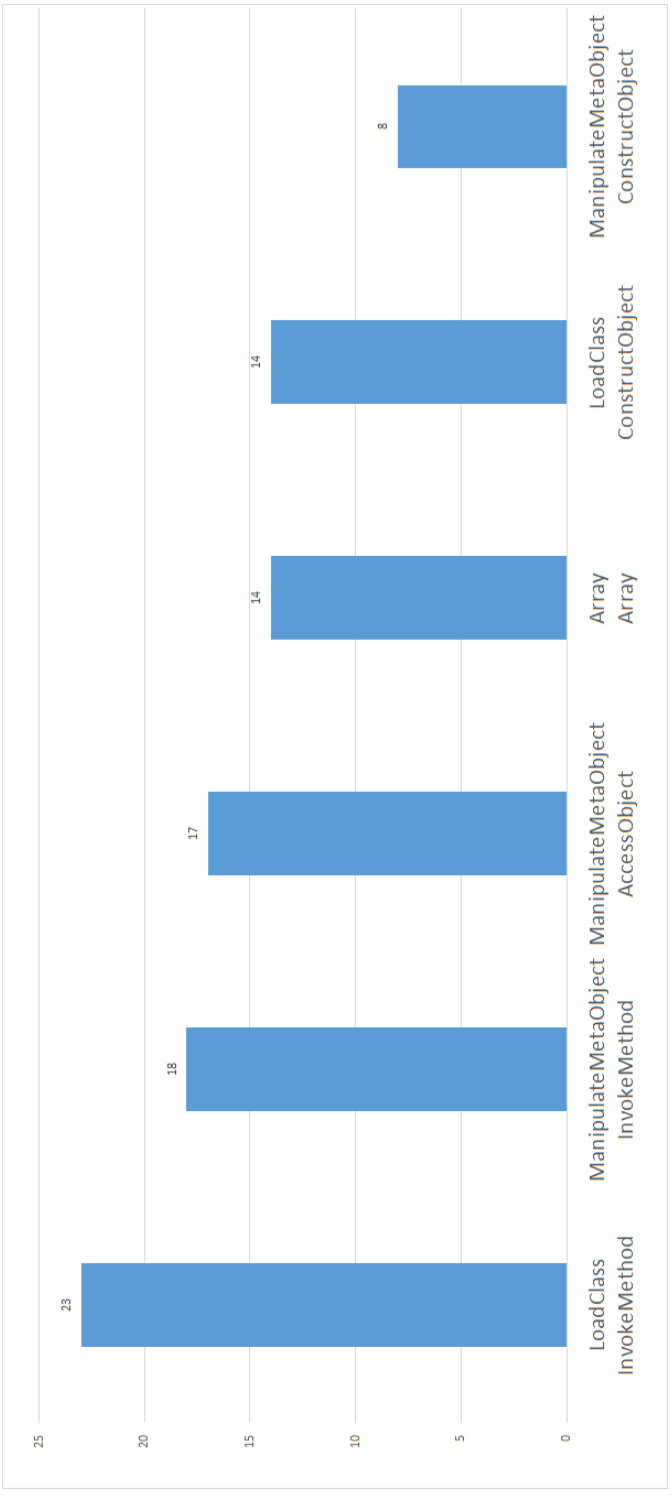


Figure 13: Patterns of specific dynamic features appearing more than seven times across all projects

The patterns shown in Fig:13 are based on the method patterns discussed before. For every semantic fitting method combination, their features categories were tracked in the same order. Further the feature pattern of LoadClass and InvokeMethod has an appearance of 23 across all projects, where as 16 out of these 23 are the method combination of Invoke*forName(String)* and *invoke(...)*. These dynamic feature category patterns therefore flatten the amount of found methods patterns to their core semantic meaning. The total amount of patterns shown in Fig:13 is 120, which is close to the number of total method patterns but still bigger. The reason for that is the definition of patterns fitting methods combination appearing only once or twice in mostly one or two projects. But through the feature patterns they are not neglected despite being not as common as the others.
The most common semantic combination is to load classes and invoke their methods although an object could easily be instantiated instead. The reason for still relying on the library may be that the instantiation is not necessary or wanted in certain cases. Another reason to use that work-around are restricted access rights for objects of that class to those method invocations.
The second most common combination is also an already discussed pattern, where the meta object is manipulated, most likely the access right as Fig:12 suggests, and then one of its methods is invoked. The third most common semantic pattern describes the manipulation of a meta object and then accessing data of it. The accessed data can be anything from fields, parameters, methods and so on. These occurrences may lead to data leaks if a malicious actor can take over the control of these instructions.
The most complex found feature pattern, in regards to its resolution, appeared four times, which is not shown in any figure. The pattern combines the manipulation of a meta object, the access to the information of its underlying object, and the manipulation of it. That pattern is most likely used to inspect data that is restricted and to change the same data, if it does not fit the current state of the program. In general the complexity of a pattern increases with the number of invocations included. The number of parameters, which is not known when talking about feature patterns, especially inter-procedurally resolvable ones, is also an important metric for complexity. Therefore the patterns with the highest amount of invocations, in this case 3, are the most complex ones for future resolution algorithms.
The feature pattern ”array, array” may appear trivial, because the same feature is paired with each other, which is misleading. The category array is used for a broad amount of methods coming down to manipulating, accessing and creating arrays. The same use cases for objects are split into different categories contrary to arrays. Therefore the appearance of the pattern array after array 14 times across all projects does not give much information about their semantic but the usage of that category in general.
The proxy feature category did not appear in any pattern and even on its own only a very small number of times across all projects. The proxy feature category is the most severe one and very hard to resolve [9].

## 5.4 Discussion

The results show that the library is mostly used for very small adjustments to the program. An overwhelming number of usages are single invocations not appearing in a pattern. Generally the metrics suggest a little significance of the reflection library usage in relative numbers. Absolute numbers, on the other hand, show that the reflection library does in fact have the capability to influence the soundness of call graphs being used in static analysis to an extent that is not negligible.
Most found patterns are not much more complex than the invocations outside of the definition of patterns. Patterns containing more than two invocations are a rarity. There are a lot of patterns that are not further examined because they appeared once or twice. Others are used in nearly every project implying there are certain use cases of the library which are very common. Most of these cases do manipulate access rights, which means changing properties that are set with ”private”, ”protected”, ”public” and so on. Also the invocation of methods is a very common use case for the library, which definitely proposes difficulties for the creation of call graphs.

# 6 Research Questions

Here the results found in the study and related work are summarised in order to answer the original research question that led to the creation of the entire thesis. **Q1: What are the sources of unsoundness in call graphs caused by the reflection API?**
In general unsoundness, caused by reflection calls, lacks information about the call. That information includes the state of the parameters at the moment of invocation. To gather and resolve these parameters a static CFG analysis has to be done. In general around 93% of those calls are statically resolvable. Most implementations do resolve less than those 93% because of the trade-off between precision and scalability. Tools like DOOP, that do resolve that 93%, still include 7% of unresolved calls statistically. These calls are not resolvable because the missing information is not retrievable in static analysis. These 7% can be reduced by using dynamic analysis looking at a subset of possible executions. Overall the sources of unsoundness can never be reduced to zero with certainty.
**Q2: Which reflection API calls and patterns are commonly used in real-world Java Projects?**
The found metrics show that the reflection library is omnipresent in real-world applications. Every project includes at least one call to the reflection library. The general amount of found calls is relatively low but in absolute numbers quite a lot considering all of them can lead to unsoundness.
The most used calls to the reflection library are methods invoking other methods and methods manipulating access rights of meta objects. These and the other results strengthen the narrative that the reflection library is mostly used for small work-arounds rather than complex structures and implementations.
Patterns are far less dominant in real-world projects than single calls. Every pattern appearing multiple times across all analysed projects consists of only two method invocations. That also suggests the usages of small work-arounds because two methods being called in combination do not offer much semantic depth, which means the idea behind those lines of code is always easy to understand in a relatively short time. These findings also show that the reflection calls have a very small dependency between each other. Therefore the complexity of resolving the resulting unsoundness is relatively easy compared to hypothetical patterns consisting of many more methods being called after one another.

# 7 Conclusion

After setting the environment for the study by presenting the basics and defining the necessary concepts, the static analysis tool ReflectDetect was created and presented in this thesis. After analysing real-world Java projects with that tool and gathering data out of scientific research on the topic, interesting insights into calls out of the reflection library, which lead to unsoundness, were the result. Recalling the number of affected classes and methods, 131 classes and 93 reachable methods are affected on average. These numbers are definitely not negligible talking about static analysis. Any static analysis tool not resolving 93 possible sources of unsoundness could lead to a way higher number of FNs and therefore hidden exploits than the amount of sources. Simultaneously developers trusting those results could choose affected projects in the future over less severe ones because the comparison lacks correct information.
Surprisingly the amount of manipulations of the meta object followed by the manipulation of the object is relatively low compared to the same pattern with the difference in accessing the data in the second step. During the process of the study I expected that to be the biggest use case of the reflection API because it feels very handy in many situations.
It is also a very good finding that the proxy feature category is not as prevalent as other features. That category is the most severe one talking about unsoundness. Methods of that category are very hard to resolve in general. A high appearance of those methods, especially in patterns, would be very bad for future work on unsoundness enabled through the reflection library.
Summing that up reflect methods are always relevant for static analysis and cannot be ignored.
Luckily related work has shown that the reflection library is considered to a certain extent in static analysis. The tools being used the most have gotten good results on resolving sources for unsoundness caused by the reflection library. Absolutely winning that comparison was DOOP which resolves everything besides the mentioned 7% statically. That 7% are resolvable using dynamic analysis supporting the static ones. Dynamic analysis can never be sound because the total number of all possible executions is just not scalable at the moment. Therefore the sources of unsoundness inside the 7% cannot be found to completion with definitive certainty.
In conclusion unsoundness caused by the reflection library is omnipresent in real-world Java projects. These sources are considered and some tools are effectively resolving them. The chances that a small number of sources persist is still always greater than zero percent.

# 8 Future work

Many aspects of the study can be improved in the future, first and foremost with regard to the limitations of ReflectDetect. The mitigation of these will bring more precision to the results and overall analysis. Besides that ReflectDetect can already be used to detect dynamic reflection calls, most likely causing unsoundness. These sources can be considered in future work or support tools in resolving them.

## 8.1 Improvements to ReflectDetect

### 8.1.1 Automation

First of all the already mentioned automation could be added to ReflectDetect. ReflectDetect is publicly available on Github[12] and therefore easily changeable and optimisable. Refactoring the CLIHandler class to run iteratively over the program arguments, ReflectDetect would be able to analyse as many projects as possible in a single execution. That would also benefit changes to the ResultProduction, because in the current state every project needs to be re-analysed manually on any change, which is exhausting and time consuming. To achieve that the CLIHandler also needs to kick off a new analysis between each iteration, which includes a reset of the data domain and a re-initiation based on the next project coming from the program arguments. Because of the modular design of ReflectDetect, that addition will be easy to implement.

### 8.1.2 CFG analysis

A more complex optimization would be a CFG analysis. By analysing the CFG of the project the parameter severity of found reflection calls could be added to the result giving more details about whether a trivial or more complex resolvable reflection occurred. On one hand the analysis of the CFG potentially results in a broader amount of patterns found by adding inter-procedural patterns to the scope. But on the other hand, the metrics found in the study also suggest that the probability of reflection calls being close enough inside the CFG to be considered a pattern by pc values converges against zero. Probabilities do not take the semantics of a program into account and therefore it is still possible more patterns could be found.
A CFG analysis would at least be a difficult task to do. While working on the implementation I actually tried to include such an analysis without success. There are many cases for the different kinds of statements that need to be considered, which would have delayed the finalisation of ReflectDetect and this thesis beyond the given time constraints.

## 8.2 Dynamic analysis

Coming to the usage of the current state of ReflectDetect in the future. One reason to add the pc values to the result was to detect dependencies in between reflection calls for the study, which the line numbers also do. The other reason is in regards to future work, especially dynamic analysis research on reflection calls. By knowing the pc values of each reflection invocation, it is possible for dynamic analysis tools to just track the stack of the current execution and activate the actual analysis once hitting a pc from such an invocation. By doing so, the overhead and efficiency of the dynamic analysis improves by a lot. Comparing the amount of reachable methods and such methods containing a reflection call, that should be strongly noticeable. The faster a dynamic analysis can run without taking the projects execution into account, the more often it can be rerun. The related works show that the precision and therefore accuracy improves with every iteration.

# Bibliography


[1] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 669–679. https://doi.org/10.1109/ASE.2015.69

[2] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*. 241–250. https://doi.org/10.1145/1985793.1985827

[3] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.* 44, 10 (oct 2009), 243–262. https://doi.org/10.1145/1639949.1640108

[5] Michael Eichberg and Ben Hermann. 2014. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614630

[6] Ben Holland. 2016. Call Graph Construction Algorithms Explained. https://ben-holland.com/call-graph-construction-algorithms-explained/. Accessed: 16-05-2024.

[7] Philipp Holzinger and Eric Bodden. 2021. A Systematic Hardening of Java's Information Hiding. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems* (Virtual Event, Hong Kong) *(ASSS '21)*. Association for Computing Machinery, New York, NY, USA, 11–22. https://doi.org/10.1145/3457340.3458300

[8] Jang-Wu Jo and Byeong-Mo Chang. 2003. Constructing control flow graph that accounts for exception induced control flows for Java. In *7th Korea-Russia International Symposium on Science and Technology, Proceedings KORUS 2003. (IEEE Cat. No.03EX737)*, Vol. 2. 160–165 vol.2.

[9] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 507–518. https://doi.org/10.1109/ICSE.2017.53

[10] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. 2017. Reflection Analysis for Java: Uncovering More Reflective Targets Precisely. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 12–23. https://doi.org/10.1109/ISSRE.2017.36

[11] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. https://doi.org/10.1145/3293882.3330555

[12] Luca Simon Stamen. 2024. ReflectDetect codebase & results. https://github.com/lu25ca09/ReflectDetect. Accessed: 04.07.2024.

[13] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88.

[14] Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. *SIGPLAN Not.* 35, 10 (oct 2000), 281–293. https://doi.org/10.1145/354222.353190

[15] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

# Eidesstattliche Versicherung

# (Affidavit)

Stamen, Luca Simon
Name, Vorname
(surname, first name)

217826
Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Systematic Identification of Java Reflection in Static Analysis

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Werl, 17.07.2024
Ort, Datum
(place, date)

Stamen
Unterschrift
(signature)

**Belehrung:**
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG - ).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

**Official notification:**
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Werl, 17.07.2024
Ort, Datum
(place, date)

Stamen
Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**