A large-scale evaluation of tracing back log data to its origin with static analysis

Master's Thesis

Daan Schipper

A large-scale evaluation of tracing back log data to its origin with static analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Daan Schipper
born in Amersfoort, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen
Simon Carmiggeltstraat 6-50
Amsterdam, the Netherlands
www.adyen.com/

A large-scale evaluation of tracing back log data to its origin with static analysis

Author: Daan Schipper
Student id: 4155270
Email: d.c.schipper@student.tudelft.nl

Abstract

Logs are widely used as source of information to understand the activity of computer systems and to monitor their health and stability. As large-scale systems generate hundreds of millions of logs per hour reaching tens of terabytes, automated techniques exist to take advantage of the rich information present in logs. However, these techniques require the link to the event that generated the log, the log statement in the source code. Several solutions have been proposed to solve this non-trivial challenge, of these the approach based on static analysis reaches the highest accuracy. Log statements in the source code are statically analysed to extract templates and match log messages to these templates, creating a link between log messages and statements. However, no evaluation has been performed in large scale environments of various industries where log messages are versatile.

We perform a field study of the approach based on static analysis of source code to relate log messages to its log statement in a large-scale environment. The approach is evaluated on a rich and versatile dataset of logs produced by over thirty thousand log statements, reaching an accuracy of 97,6%. We provide an non-intrusive, adaptable to custom logging practices and easily extendable implementation that is ready to be used in large-scale system, which allows for automated log analysis techniques to be adopted.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Company supervisor:	Ir. N. Brenkman, Adyen
Committee Member:	Dr. A. Katsifodimos, Faculty EEMCS, TU Delft

Preface

This thesis has been submitted for the degree of Master of Science in Computer Science at the Delft University of Technology. Not only is this thesis the end result of the graduation project, but it also concludes my time as student. Throughout this time I have developed both scientifically and personally to cope with the challenges that I faced. This all would not have been possible without the support of the many people that stood alongside me, which I would like to thank in this preface.

Maurício Aniche, thank you for sharing my enthusiasm about this topic and guiding me throughout the past nine months. Your insights really helped me focus on what is important and your advice was greatly appreciated. At the start of the project you let me roam free to explore the research area, while later being available for any question I had.

Furthermore, I would like to thank the other members of my thesis committee, Arie van Deursen and Asterios Katsifodimos, for their efforts and time.

I would like to thank all my colleagues at Adyen for their help, always open to my random questions. Although they made perfectly sense to me, it must have been difficult to understand the context. Nevertheless I could expect an answer that helped me continue. You really helped shape the high-level explanation of this thesis. A special thanks to Nils Brenkman, who was always available for either questions or my frustrations.

Friends and family, thanks for your unconditional support. It must have been difficult to hear me out while talking about the most specific details and challenges I dealt with, but I could always rely on a sympathetic ear. Special thanks to those who provided feedback on my draft.

Papa, dit is voor jou.

Daan Schipper
Delft, the Netherlands
September 12, 2018

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Logs record runtime information of computer systems and produce timestamped documentation of events, states and interactions of components. The information gained from logging is used to identify problems and help to quickly solve them, consisting mostly of manual labour. Overall, a log entry is produced by a log printing statement in a system program's source code. Techniques have been developed to relieve this manual work and to take advantage of the rich information present in logs in an automated manner, such as process mining [11, 17, 37], anomaly detection [9, 15, 16, 40, 41], fault localisation [38, 46], invariant inference [10], performance diagnosis [21, 29, 33, 34, 45], online trace checking [8], and behaviour analysis [30, 42].

All these log analysis applications process the events corresponding to the entries contained in the log files, or more precisely, the invoked statements in the source code. However, tracing back the origin of log data to its log statement in the source code is a non-trivial challenge. First, determining the event during runtime affects the performance of the system. Frameworks like Log4j [24] allow developers to print the class name and line number of log statements. However, collecting this information at every log statement comes with a loss of performance[1]. Second, the log messages consist of free-form text to express semantics combined with parameters to record runtime information of interest. Because of this unstructured nature, the logs cannot be automatically analysed. Lastly, the sheer size of today's systems make it infeasible to employ heuristics, as it would be almost impossible to manually maintain a set of rules that process the logs [40].

As no direct connection between log messages and source code exists, it is necessary to parse the log entries automatically in order to retrieve the actual events recorded in the log [27]. Previous work proposes several approaches based on clustering [15, 35], heuristics [26, 36], longest common sequence method [12], textual similarities [19], evolutionary search [27], and static analysis [40] to solve this challenge.

In this thesis we perform a field study of tracing back the origin of log data to its origin, the log statement in the source code, at Adyen, a payment service provider operating globally and providing over 250 different payment methods which include local payment

[1] Behind the scenes, Log4j collects the log statement line by throwing an exception and capturing the generated stack trace. Log4j's developers have experimented with other alternatives, but so far, this is the most efficient one (see `https://issues.apache.org/jira/browse/LOG4J2-1029`).

methods and more services related to payments [1]. The large-scale systems produce approximately a billion log lines each day produced by tens of thousands of log statements. We apply an approach based on static analysis of the source code that achieves the highest accuracy [40]. Previous work lacks evaluation of its effectiveness in large-scale environments of various industries, as Xu et al. [40, 39] use logs from primarily storage systems. Related work depending on this approach only evaluates the end results, not the approach itself [32, 43, 44].

We locate logging statements by statically analysing the source code and extract templates from log printing statements. The templates are then used to match the log messages with to identify the log statement that generated it. We evaluate the approach on a log dataset taken directly from the production servers and show that approach is applicable to real world environments. All but two messages from the dataset of 100.000 log messages have been traced back to its log statement in the source code. Of those traces, 97,6% have proven to be correct with a confidence level of 95% and interval of 5%. We provide an non-intrusive, adaptable to custom logging practices and easily extendable implementation that is ready to be used in large-scale system, which allows for the aforementioned log analysis techniques to be adopted.

Chapter 2

Related Work

Logs capture the events, states and processes that happen during the execution of a software application. These events are stored in a log file, allowing developers of the application to analyse the behaviour of the system. Logs are widely employed and different types of logs exists for specific purposes. We characterise the following types of logs:

- **Application logs:** Logs that are produced by a software application. Developers introduce statements in the source code that will record a message upon invocation. The actual message of the log is determined by the developer.
- **System logs:** A standardised method of logging for operation systems, allowing for a separation between generating, storing and analysing the message.
- **Transaction logs:** Log that capture the communications between a system and the users of that system [31].

The information present in logs is of unmistakable value of all types of logs, but in this thesis we limit the scope to application logs. Logs are a commonly employed practice in the industry to monitor and understand the system's behaviour. The corresponding research field has been an active research area for decades. We survey only the approaches related to automatic analysis of events within application logs.

As outlined in the introduction, many approaches have been proposed to relate log messages to their events. The rich information present in logs are used for process mining [11, 17, 37], anomaly detection [9, 15, 16, 40, 41], fault localisation [38, 46], invariant inference [10], performance diagnosis [21, 29, 33, 34, 45], online trace checking [8], and behaviour analysis [30, 42]. However, these log analysis techniques require the event that generated a log, the invoked statements in the source code, to deduct any further information. The source of a log message is considered the corner stone of all log analysis techniques, as most of the data mining models used in these log analysis techniques require structured input [19]. Without the event, no structure can be obtained from the log message, thus rendering most log analysis techniques obsolete. The goal of this thesis is to provide this trace to the source code.

Tracing back a log message to the event that generated it is a non-trivial challenge. The logs originate from log statements that can be constructed in every way imaginable, as long as it follows the grammar of the language. This results in log messages that are clearly understandable for developers, but machines cannot process this unstructured format. Therefore, the logs need to be parsed to events that represent a log statement in an application [6]. Typically a log message consists of a constant part, which remains the same for every event occurrence, and a variable part containing dynamic information, which is determined during runtime. The goal of log parsing is to separate this structure of constant and variable parts within a log message. This is often referred to as "line pattern", "log event", "log key", "message signature" or "template". From here on we will use *template* to denote the constant part of a message.

As it is the basis for many log analysis techniques, log parsing is an active research area. Previous work proposes several approaches based on clustering combined with heuristics [15, 26, 35, 36], longest common sequence method [12], textual similarities [19], evolutionary search [27], and static analysis [40].

He et al. [18] performed an evaluation study on the most popular clustering based methods and found, despite achieving an high accuracy, that SLCT [36] and IPLoM [26] do not scale well with the volume of logs, since the clusters are constructed according to the difference in the messages. Furthermore, offline log parsing methods such as [15, 26] are limited by the memory of a single computer. Therefore, He et al. [19] propose an online method that parses raw log messages in a streaming manner, outperforming previous methods [12, 15, 28, 26]. However, the evaluation process samples only 2.000 log messages from each dataset, a following study shows that the method crashes on a dataset of 300.000 log lines [27]. Another finding of clustering based methods is that the overall accuracy is improved when log messages are preprocessed with some domain knowledge based rules to remove obvious numerical parameters, such as numbers, memory and IP addresses [18]. Although beneficial to the effectiveness of the log parsing, this is a manual process.

Messaoudi et al. [27] capture the template of a message by applying an evolutionary algorithm. This first of a kind approach significantly outperforming other approaches [19, 26]. However, it requires filtering of duplicates on top of pre-processing of log messages. Without filtering, the time to process the log messages is significantly longer.

Approaches based on static analysis have an additional source of information available, the source code itself, thereby reaching a significantly higher accuracy. Templates are extracted from the logging statements which are then used to match log messages with. This extra knowledge additionally allows them to be completely automated, thus not require any form of manual work. Log analysis techniques have been proposed that rely on this approach, Xu et al. [40] created features for machine learning techniques to detect anomalies, Yuan et al. and Zhao et al. [43, 44] reconstructed the execution flow and Shang et al. [32] provided development knowledge back to the developers that is present in various development repositories related to a specific log message. However, these works present evaluation results specifically suited for the end result of their application to show that the information that is mined is correctly obtained and useful. The fundamental process, linking the log messages to their event, is overlooked.

Only Xu et al. [39] do provide results of the accuracy, obtained by applying the approach

based on static analysis to log datasets of system that handle persistence. The number of variables present in log messages is on average much lower compared to the systems of Adyen, 1,42 and 1,14 to 1,92 respectively. Furthermore, the industry of the systems are completely different, thus with different coding and logging styles. This thesis provides an additional evaluation of the approach based on static analysis within in versatile system in a different environment and the challenges that arise when applying such an approach to a large-scale system.

Chapter 3

Approach

The goal of the approach is to trace back log data to its origin, the statement in the source code. We use static analysis to extract templates from log statements and process logs with these templates, similar to the method used in previous work [32, 40, 43, 44].

Once a log statement such as `log.info("average = " + average)` is invoked, a textual representation is generated of its argument and the log message `average = 4` is produced. The log message can be enriched with information like a timestamp, severity level or fully qualified name depending on the employed logging framework. However, the message itself is formatted according to the arguments in the source code. To map log messages back to log statements, we construct a *template* in the form of a regular expression based on the arguments of log statements in the source code that capture all possible log messages produced by that statement. For the above log statement we would generate `average = .*` as template. We then find for each log message the regular expressions that matches and select the template that has the highest similarity to the log message.

We present an approach to link log messages to source code via static analysis. The approach consists of two main phases: (1) creating the templates and (2) matching the log messages to the templates. The first phase can be prepared beforehand, while the second phase can be online executed to process the logs as they are produced.

We apply the approach to a complex system, therefore the following characteristics are required to hold for the implementation:

- **Non-intrusive:** No need to apply any change to the existing source code and has no performance impact on the run time at all. It is a completely separate process which requires no runtime information.
- **Adaptable:** Easy to extend with logging library of choice. You only have to configure the settings with the class name and the method signatures.
- **Extendable:** Complex objects such as `StringBuilder` and `String.format` as part of log statements can be easily added and analysed.

The overview of the approach is shown in Figure 3.1, in which a square represents a process within the approach, whereas a polygon represents input or output. We start by

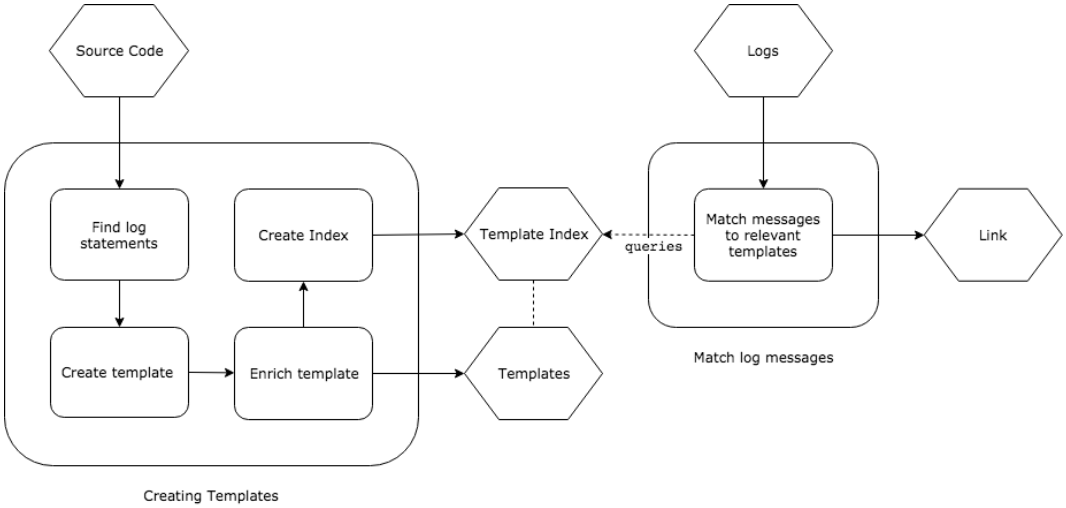

Figure 3.1: Overview of the approach.

finding log statements in the source code, for which we traverse the abstract syntax tree, a data structure corresponding to the source code [3], and analyse nodes related to log statements. Next, we collect a template from the statement along with the severity level and class name. We construct a template in the form of a regular expression that matches all possible log messages produced by it. We then enrich the templates with type analysis information such as the textual representation of objects and type hierarchies to make the templates more precise. To make the templates easily searchable, we conclude this phase by creating an index of the templates.

For each log entry, we we get candidate templates from the template index for each log based on the name, severity and message. Of the candidates we choose the template whose regular expression matches the log message. If such a template is successfully found, then we have connected the log to the source code and have successfully traced back the log to its source.

3.1 Find Log Statements

We search the source code for statements corresponding to log statements so we can programmatically evaluate its argument and generate a template. We parse the source code to an abstract syntax tree, a tree representation of the abstract syntactic structure of source code which follows the grammar of a programming language [3]. An example of such a tree is given in Figure 3.2, which represents the structure of the source code provided in Listing 3.1. The relations between the different expressions of the log statement are clearly visible. Multiple solutions already exist to generate an abstract syntax tree from source code, we opt to use JavaParser [20], a simple and lightweight library.

The first step takes the source code as input, programmatically evaluates the statements and searches for those corresponding to log statements.

```
1 logger.info("average = " + (min + max) / 2);
```

Listing 3.1: Example of a log statement.

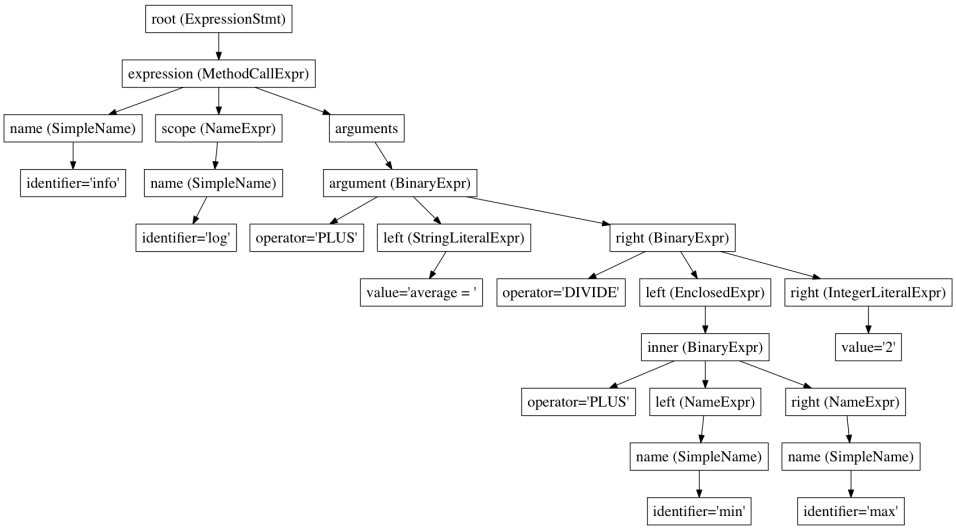

Figure 3.2: The abstract syntax tree of the source code in Listing 3.1.

```
1 logger.log(Level.INFO, "average = " + (min + max) / 2);
2
3 customLogFunction("average = " + (min + max) / 2)
4 customLogFunction(String message) {
5   logger.info(message);
6 }
```

Listing 3.2: Examples of peculiar logging methods.

In order to analyse the log statements, we iterate over the individual nodes of the abstract syntax tree to find nodes that represent log statements. Previous work by Zhao et al. [44] identifies log statements by searching for method calls corresponding to the standard logging methods, such as those defined by Log4j [24]. In practice we need to extend this, companies like Adyen create their own logging libraries suited for their needs. Furthermore, log statements from other logging methods could be missed, of which examples are shown in Listing 3.2.

To identify log statements we look for all method invocations on objects of the logger class, which can be easily obtained by inspecting the source code [40]. We test if the signature of the invoked method is equal to that of the *logging methods*, methods that actually generate logs. This allows for developers to input custom logging methods which are then automatically analysed by the approach.

```
1  public void info(Transaction transaction, String message) {
2    innerlogger.info(generatePrefix(transaction) + message);
3  }
4
5  private String generatePrefix(Transaction transaction) {
6    if (transaction != null) {
7      return "{" + transaction.getId() + "} ";
8    }
9    return "";
10 }
```

Listing 3.3: Logging methods with an additional flow before generating the log message.

Furthermore, if a logging method contains additional steps to generate the log message, such as appending an identifier to each log message as shown in Listing 3.3, the approach can be extended to support such custom methods. A custom flow can be defined by the developer that specifies how to construct a template from the arguments of this custom log method. We follow the same operations as the logging method, in this example prepending the template obtained from the message argument with `(.* )?` to take the optional prefixed identifier into account. In conclusion, this method locates all log statements in the source code, including invocations to custom logging methods.

3.2 Create Template

The goal of the approach is to construct templates from the arguments of log statements that match all messages generated by it. These templates are then used to match the log messages with, providing the trace back to the log statement in the source code. Thus a more precise template will more accurately match the log messages.

However, the arguments of the log statements of which we will be generating a template have no restrictions to them. Developers can construct the argument in every way imaginable as long as it follows the language specification and a timestamped message will be generated of its arguments. Even non-primitive objects with a custom textual representation can be included. The arguments can vary from a simple literal expression to an interpolation of primitive types together with objects, which all are converted to a single line of text during runtime.

Static analysis is applied to create templates based on the arguments provided to the logging methods. We copy the literal expressions to the template and substitute all runtime variables with wildcards, while saving he type of the variable, which will be used in the following steps. We illustrate the approach according to Listing 3.4 where three types of arguments can be distinguished: literals, concatenations and object representatives. Although all log statements output the same message `average = 4`, the log statements are increasingly difficult to create templates for.

```
1  logger.info("average = 4");
2
3  logger.info("average" + " = " + "4");
4
5  logger.info("average = " + 4);
6  int min = 2, max = 6;
7  logger.info("average = " + (min + max) / 2);
8
9  String string = "average = ";
10 logger.info(string.concat((min + max) / 2)));
```

Listing 3.4: Examples of different types of log statements.

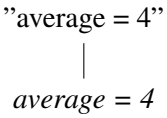

Figure 3.3: Template creation process of a log statement with literal variables.

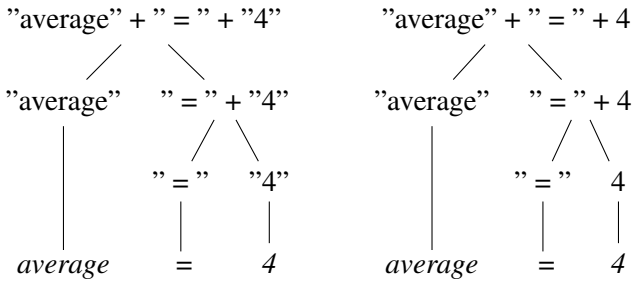

Figure 3.4: Template creation process of a log statement with concatenated variables.

Literals The first category are arguments which consist of a single literal expressions. We take the exact value as is present as argument and create a template of the argument. Figure 3.3 shows the template creation process of a log statement which consists of a literal. The resulting template is displayed in italics.

Concatenations The second category consists of log statements with an argument consisting of concatenated expressions, therefore we analyse the individual expressions. In case of the Java programming language, if the expressions are of the type `char`, `double`, `int`, `long`, `boolean`, `null` or `String`, we construct the template of the value of the expression. Figure 3.4 shows the template creation process of a log statement containing concatenated variables.

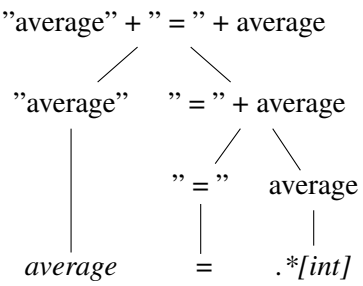

Figure 3.5: Template creation process of a log statement with a non-primitive variable.

However, if the expression is not of a literal type, it can have different values depending on the state of the system. Therefore we represent such an expression with a wildcard, but we also save the type of the expression which will be used in the following steps. Figure 3.5 shows the template creation process of a log statement containing non-primitive variables.

Object references Developers use helper classes to construct log messages, of which a few are listed on lines 10, 15, 20 and 22 of Listing 3.4 for Java. Although being created with different helper classes, all statements would result in the same template `average = .*[int]`. This poses an additional challenge to the template creation process, as the information to create the template is spread out in the code.

To cope with such variables, we (1) identify that the variable originates from a helper classes, (2) analyse the invocations on this variable within the scope of its declaration and (3) generate the template based on these statements. Figure 3.6 shows the template creation process of such a variable. When analysing the argument `builder.toString()`, we observe that this variable originates from a helper class. We search for all invocations on the variable `builder` and analyse, if necessary, the individual invocations the same as we would do the argument of a log statement.

In this work we provide support for `StringBuilder`, `String.format` and `JSON`, while leaving out support for others due to time constraints. At Adyen, we empirically observed these object references are of importance, as these classes are heavily used to construct log messages, therefore we prioritised on supporting these classes.

3.3 Enrich Template

We want to create as precise as possible templates to improve the overall accuracy of the approach. Some variables capture the state of the system, which can be used to investigate certain behaviour of the system. The actual values of these variables are determined during runtime, thus we cannot infer them with static analysis. However, in object oriented programming languages, these variables can be a reference to an object containing the value, not the actual value itself. Likewise, these objects also have a set of expressions defining their textual representation. We create an intermediate template representing the

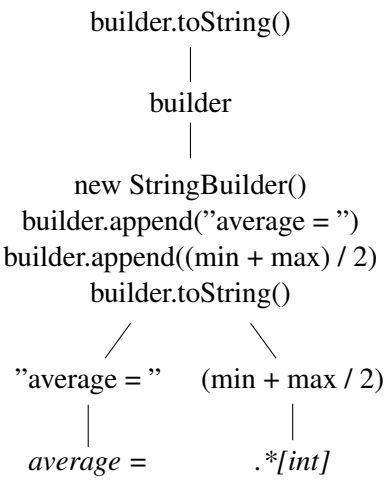

Figure 3.6: Template creation process of a log statement with variables referencing objects.

```
1  Average average = ...;
2  logger.info(average)
3
4  class Average {
5    int average;
6
7    String toString() {
8      return "average = " + average;
9    }
10 }
```

Listing 3.5: Log statement with a variable of non-primitive type.

textual output of these object. This allows us to infer the actual output of each variable of non-primitive type.

Consider the code listed in Listing 3.5, where the textual representation of the object `Average` is defined in the `toString` method. Following the previous step, we would have create the template `.*[Average]` for the log statement on line 2. However, we can substitute this wildcard with the intermediate template of its type, which is obtained in a similar manner as shown in Figure 3.5. This results in a more precise template, as shown in Figure 3.7.

Another challenge of object oriented programming languages is inheritance, where classes can be derived from other classes, thereby inheriting fields and methods from those classes. This implies that the actual type of a variable is only known at runtime. To account for inheritance, we duplicate the template for each derivation and substitute the type of the variable for its subclass. If a template contains a non-primitive variable after substituting

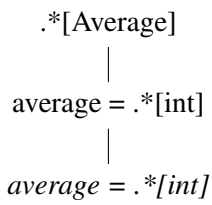

Figure 3.7: Type enriching of non-primitive values within a template.

```
1  class TwoNumberAverage extends Average {
2    int first;
3    int second;
4
5    String toString() {
6      return "average of two numbers = " + ((first + second) / 2);
7    }
8  }
9
10 class ArrayAverage extends Average {
11   int[] values;
12   int sum;
13
14   String toString() {
15     return "average of array = " + (sum / values.length);
16   }
17 }
```

Listing 3.6: Log statement with a variable of non-primitive type.

it once, we substitute it again with the textual representation of that type. We recursively substitute non-primitive variables in all templates until either no textual representation is known or all non-primitive variables are replaced with primitive variables, following the approach of Xu et al. [40].

We illustrate this step by extending the previous example from Listing 3.5 with the classes listed in Listing 3.6. We can see two classes extend the `Average` class with both their own textual representation. Same as before, we create intermediate templates of these classes, as shown in Figures 3.8 and 3.9. Next, we recursively check each non-primitive variable and substitute it with either its derivatives or its textual representation, as shown in Figure 3.10.

3.4 Create Index

Simply evaluating all templates in order to find the template that correctly parses a log message is infeasible, taking on average $O(n \cdot m)$, where n is the number of logs and m the

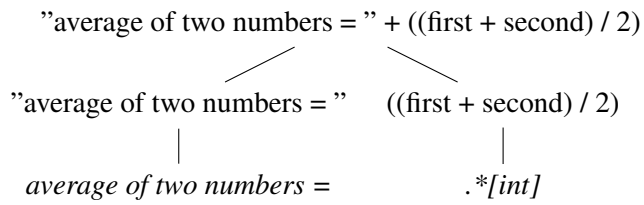

Figure 3.8: Creation process of the intermediate template of `TwoNumberAverage`.

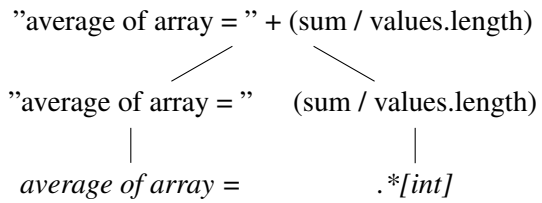

Figure 3.9: Creation process of the intermediate template of `ArrayAverage`.

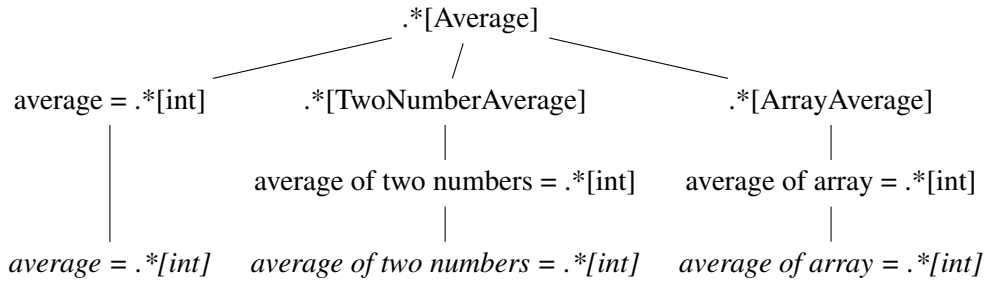

Figure 3.10: Recursive type enriching of non-primitive values within a template.

number of templates. We want to speed up this process and make the constructed templates in the previous steps easy searchable.

The templates consists of a constant part representing the semantic meaning of the state interleaved with variables consisting of variables which will be resolved during runtime. The constant part will added to the log message as it is, whereas the variable part will be substituted with its actual value during runtime, which alters based on the state of the system. We find templates that are more likely to match by searching for templates that are constructed with the same content compared to that of the log message. We then only have to evaluate these candidate templates instead of all of the templates.

We compile the constant part of all templates into a reverse index [7] to firstly evaluate templates with a higher possibility of matching. We use Apache Lucene [5] to index the templates, following the approach by Xu et al. [40]. The templates from the previous

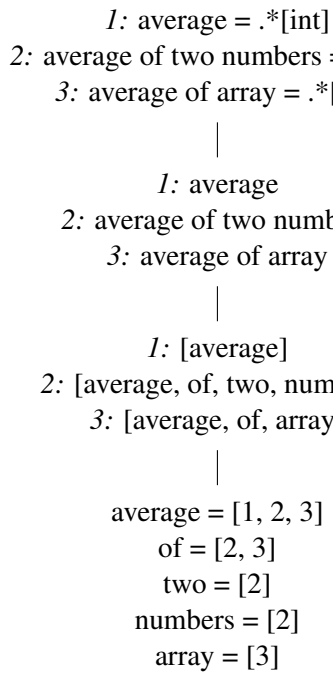

Figure 3.11: Creation process of the reverse index of two templates.

examples would be compiled to the index as shown in Figure 3.11. This allows us to reduce the number of templates to evaluate per log message.

3.5 Get Candidate Templates

We provide for each log message the trace to the log statement in the source code by finding the template that matches the message. As to not evaluate all templates for each log message, we collect a set that is likely to contain a matching template. We firstly filter the templates on the class name and the severity accompanying the log and then retrieve a list of candidate templates sorted by relevance according to the content of the message from the index. Then we select the first template from this list whose regular expressions matches the message as its origin.

We need to find the template corresponding to the log statement that is truly the source of the message. We query the index for the templates that are more likely to match based on the content of the message and receive a relevance ranked list based on TF-IDF [23]. Therefore it is required to have as precise templates as possible, the more the template corresponds to the messages the more likely we will try the template. We then evaluate whether the corresponding regular expression actually matches the message before selecting it as its origin.

```
1  class A {
2    Logger logger = LogManager.getLogger(A.class);
3    logger.info("average = " + (min + max) / 2);
4    logger.warn("average = " + (min + max) / 2);
5  }
6
7  class B {
8    Logger logger = LogManager.getLogger(B.class);
9    logger.info("average = " + (min + max) / 2);
10 }
```

Listing 3.7: Duplicate log statements.

When receiving the log message `average of two numbers = 4`, any non-alphabetic characters are stripped from the message. Next, we tokenise the message in the exact same manner when creating the index, so the resulting query to the index index would be `[average, of, two, number]`. The index returns any templates who have the same tokens and scores them according to relevance. If we would query the index from Figure 3.11, we would receive `[{2, 5.7}, {3, 2.4}, {1, 1.6}]`. We then iterate this list and check whether the regular expression matches the matches, resulting in the template `average of two numbers = .*[int]`.

We also restrict the number of possible templates to evaluate. A common practice is to supplement the log message with the name of the class and a severity level. We use this information to narrow down the location in the source code, since the class name is part of the location. Furthermore, the class name gives another feature to restrict the number of templates to review per log message. Consider the example listed in Listing 3.7 and the log `[A] INFO average = 4`, where `A` denotes the class name, `INFO` the severity level and `average = 4` the message. No distinction can be made between the log statements solely looking at the message. However, by considering the name of the class and the severity, we derive the log statement on line 3 as the source of the message.

Chapter 4

Empirical Study

In this chapter we evaluate the effectiveness of linking log data to its log statement in the source code with static analysis in terms of accuracy and performance. We assess how well the approach performs in a versatile environment, where a billion log messages are produced by approximately thirty thousand log statements. To evaluate the accuracy, we test whether the link provided by the approach is indeed correct. To evaluate the performance, we make a distinction between the creation of the templates and matching the log messages to the templates, since the former has to be performed once per release and the latter is a continuous process that has to match the speed at which the log messages are generated by the production systems.

In summary, we evaluate our approach according to the following research question:

RQ1 *What is the accuracy of the approach when dealing with extensive log data?*

RQ2 *What is the performance of creating the templates for the log statements in the source code?*

RQ3 *What is the performance of matching the log messages to its origin?*

4.1 Context of the Study

We evaluate the effectiveness of the approach in real life conditions and will use log data taken directly from the production servers of Adyen. We will first detail the currently employed techniques at Adyen and then discuss the dataset used to evaluate the approach.

4.1.1 Adyen

Adyen is a payment service provider offering a global solution for retail merchants. Over 250 different payment methods in more than 150 currencies can be used to make a payment by customers all over the world. Adyen offers an all-in-one solution for merchants to process payments extended with services for a better overall service.

Adyen deals with an enormous amount of payments, processing $122 billion in volume for merchants in 2017 [2]. To improve the quality of a transaction, such as issuing a payment

at a certain credit scheme, the information regarding that transaction is logged to ensure that it is completely and quickly processed. The information gained from logging is used to identify problems in the handling of certain payments and automated monitoring techniques help to quickly solve them.

The production systems of Adyen generate over a billion messages a day, requiring automated means of analysing the logs. The currently available log analysis processes consist of:

- Automated collecting, indexing, storing, and searching of logs. The logs generated by the individual servers are routed to a centralised processing pipeline consisting of LogStash [25], ElasticSearch [13] and Kibana [22], also known as the ELK Stack [14].
- Automatised indexing of JSON data. Log messages are appended with JSON data containing runtime variables, which are automatically identified and parsed by the ELK Stack. This allows developers to search and analyse specific runtime variables.
- Monitoring rules. Log messages that satisfy predefined rules are selected from the logs, processed to extract specific details and actions can be executed if the prerequisites hold. These rules are used to alarm developers of critical situations that may arise during operations.
- Clustering of messages. The log messages with severity levels of warning and error are clustered according to similarity of the message. If a message is not like the others, a new cluster is created. This is perceived by the system as a new bug and automatically notifies developers.

However, many manual hours have to be invested to investigate and monitor these logs to make sure the system runs as desired. Specifically, domain knowledge is almost always required to automatically process specific types of logs. For example, to automatically parse JSON data within log messages, a custom logging framework has been implemented. The JSON data is automatically parsed and its fields are stored using an index based on the keys. If the entered key is unique, the system is overloaded with creating the index. To prevent this, manual rules have to be constructed to prevent such keys.

Automated techniques to extract information from all logs are required to actively monitor all events occurring within the production servers of Adyen. The need exists to further automate the log analysis process within Adyen to minimise the amount of manual time and effort needed.

4.1.2 Dataset

The dataset used to evaluate the approach consists of logs taken directly from the live production servers of Adyen. These logs are produced during normal operations and no filter is applied on the messages. Table 4.1 shows the details of the dataset.

The logs are produced by source code which has the purpose to process payments from all over the world. At the moment of writing the code base consists of millions of lines

System	Messages	Log Size	Time of day
Adyen	100.000	70.1 MB	17:00:00

Table 4.1: Data set used in evaluation.

of code[1], written by over 150 developers. Of those lines, approximately tens of thousands of log statements generate log messages in the production servers. The percentage of lines of log statements is on the lower end compared to that of other systems, normally about 1%-5% [40].

On top of the standard methods provided by the popular logging library Log4j [24], Adyen uses a logging framework that allows for different sets of arguments which aid the developers. As previously stated, JSON data within a log message is automatically recognised and made easily searchable. To not have constructs of new JSON objects all over the code base, methods are provided which convert variadic arguments that represent the key value pairs of a field to an JSON object. This logging library provides additional means to log messages, mostly for convenience and to prevent code duplications. Table 4.2 details the usage of all the different methods used within Adyen. The custom convenience logging methods do not overshadow the old and standard way of logging, as 93,3% of logging methods are the same as common practice.

4.2 Procedure

To answer RQ1, we match the log messages from the dataset to the source code and evaluate whether the link provided by the approach is indeed correct. In the 100.000 messages in our sample, our approach generated 676 links (i.e., connected the log messages to 676 different log statements in the source code). To identify whether the link was correctly made, we manually analysed a statistically significant sample of 245 links, with a confidence level of 95% and confidence interval of 5%. For each link, we select one random matched message to evaluate the link. More specifically, we check whether the message could have been produced by the statement by taking into account the structure of message, the severity and the accompanying class name.

Furthermore, we evaluate the log messages of which the approach provides a link to an incorrect log statement. We manually inspect these log messages to inspect why the approach was unable to provide a correct link. We provide explanation and show the underlying cause for the misidentified messages.

To answer RQ_2, we execute the approach ten times on the dataset to eliminate any bias of external programs influencing the execution time. The machine the experiment runs on has 2 cores @ 3.1GHz from a Intel Core i7 CPU, and 16GB RAM. We analyse the execution time per individual step of the creation process and report the mean of the execution times of the ten runs. Furthermore we analyse the execution time of creating a template. We measure the time needed per file against the lines of code that reside in that file. Lastly, we

[1]Exact number is omitted due to confidentiality issues.

Method	Standard	Usage (%)
info(Object)	yes	37,77%
info(Object, Throwable)	yes	2,27%
info(Object, JSONObject)	no	0,04%
info(Object, Object...)	no	1,14%
info(Object, Throwable, Object...)	no	0,03%
info(Object, Throwable, JSONObject)	no	0,01%
warn(Object)	yes	27,64%
warn(Object, Throwable)	yes	18,69%
warn(Object, JSONObject)	no	0,05%
warn(Object, Object...)	no	0,80%
warn(Object, Throwable, Object...)	no	0,16%
warn(Object, Throwable, JSONObject)	no	0,01%
error(Object)	yes	3,42%
error(Object, Throwable)	yes	3,51%
error(Object, JSONObject)	no	0,01%
error(Object, Object...)	no	0,16%
error(Object, Throwable, Object...)	no	0,04%
error(Object, Throwable, JSONObject)	no	0,01%
info(Message<?,?>, String)	no	1,51%
info(Message<?,?>, Throwable)	no	0,07%
info(Message<?,?>, String, Throwable)	no	0,10%
info(Message<?,?>, String, Object...)	no	0,01%
warn(Message<?,?>, String)	no	1,80%
warn(Message<?,?>, Throwable)	no	0,10%
warn(Message<?,?>, String, Throwable)	no	0,24%
error(Message<?,?>, String)	no	0,33%
error(Message<?,?>, Throwable)	no	0,04%
error(Message<?,?>, String, Throwable)	no	0,01%

Table 4.2: Usage of logging methods within Adyen.

Total log messages	Log statements	Failed	Failed %
100.000	676	6	2,4%

Table 4.3: Accuracy results. The failed number and percentage indicate the log statements that were incorrectly identified as source.

analyse the distribution of time needed per log statement based on the median, quartiles, minimum and maximum.

To answer RQ_3, we link the log messages from the dataset to the templates and analyse the execution time needed per individual log message. We evaluate the distribution of the execution time according to the mean, quartiles and quantiles.

4.3 Results

4.3.1 RQ_1 - What is the accuracy of the approach when dealing with extensive log data?

Overall, the approach achieves 97,6% accuracy (239 out of 245 analysed log statements) on tracing back the origin of log data to its log statement in the source code, as shown in Table 4.3. All but two log messages have been linked to log statements in the source code. These two failures can be explained because they have been both produced by the same log statement, which logs a message which is too large[2]. This log statement has already been modified in a future release with the message "*Let's not log the entire serialised summary item*".

Out of the 676 log statement identified as source of all log messages, 6 of those have proven to be incorrect. The misidentifications occur due to the following underlying causes:

- **Exclusively JSON data present in the message:** Before querying the index, the JSON data is stripped from the message. Therefore, when message consist of only JSON data, as shown in Listing 4.1, the resulting query is an empty string along with filters on the severity and the name of the logging instance. As a result, a matching template corresponding to a wrong statement could be returned from the index, resulting in an incorrect link.

- **Unknown logging method:** A separate logging class is used to construct messages according to a specific format. Since it is not known that the methods in this class fall under the category of logging methods, no templates are created of statements calling this method, thus the approach cannot provide a correct link.

- **Inaccurate querying of the index:** The approach fails to query logs with a message following JSON data, as shown in Listing 4.2. The part at the end of the message is erroneously stripped from the message along with the JSON data. The approach

[2]The message consists of minimum 17.286 characters, the rest of the message is not present after being processed by the ELK Stack.

```
JSON: {
  "service": "Recurring",
  "messageFormat": "soap",
  "encoding": "utf-8",
  "requestData": "AAM...ig4",
  "responseCode": 200,
  "ms": 46,
  "responseData": "AAMB...Iw="
}
```

Listing 4.1: Message containing exclusively JSON data.

expects the JSON data to be the last part of the message, which is evidently not always true. This can be easily resolved by stripping the JSON data from the message in a more sophisticated manner.

- **Inaccuracies in creation process of templates:** The approach uses static analysis to create templates of log statements that predict what the messages will look like at runtime. Unfortunately, not all predictions are completely accurate. For example, the templates for the log statements listed in Listing 4.3 are not properly constructed.

  The variable argument of the first log statement refers to a message that is prepared beforehand, however the approach does not check the scope for such statements. Therefore the resulting template for this statement consists solely of a wildcard, since the argument is interpreted as a variable. This can be mitigated by searching the scope of declarations for any of the variable arguments in the log statement.

  Likewise, the approach misses the constant string that is part of the initialisation of the StringBuilder object, which is crucial to the template of the second log statement. The resulting template for this specific statement is now only `.*:  .*`, whereas it should have been `Received request from .*:  .*`.

  Lastly, due to the complex way this single variable is declared, the method call corresponding to the third log statement is not reached. To clarify the complexity of this statement, it consists of a variable declaration with a method call as initialiser. However, the method call is a chain of method calls with as scope another method call, repeated six times. In one of those method calls a lambda expression is enclosed, where the log statement resides in.

RQ$_1$: Our approach correctly identifies 239 out of 245 links to the source code, achieving an accuracy of 97,6%. Support can be easily added to the approach to correctly identify the source of the misidentified messages.

```
JSON: {
  "serviceError": "Failed to retrieve account holder '***'",
  "encoding": "UTF-8",
  "serviceErrorCode": "10_035",
  "serviceErrorType": "validation",
  "messageFormat": "json",
} Binary IN: AAM..JM=
```

Listing 4.2: Message containing JSON data appended with some other part of the message.

```
String logMessage = "JSON: " + json.toString() + " Binary IN: " + binary;
log.info(logMessage);

StringBuilder sb = new StringBuilder("Received request from ");
sb.append(request.getRemoteAddr()).append(": ");
// populate StringBuilder
log.info(sb.toString());

Stream.of(...)
  .map(...)
  .filter(Objects::nonNull)
  .findFirst()
  .orElse(() -> {
    log.info("Selected default crypter",
      "crypter", (keyMap.get(DEFAULT_CRYPTER) == null) ? "null" :
        keyMap.get(DEFAULT_CRYPTER).getClass().getSimpleName(),
      "firstKey", firstKeyName,
      "secondKey", secondKeyName,
      "operationType", type);
    return keyMap.get(DEFAULT_CRYPTER);
  }).get());
```

Listing 4.3: Examples of log statements of which templates are incorrectly generated.

4.3.2 RQ_2 - What is the performance of creating the templates for the log statements in the source code?

Table 4.4 shows the execution time per individual process of the template creation process for the code base of Adyen. The total execution time took on average approximately 40 minutes across all ten runs. Most of this time is spend searching the code base for method calls that correspond to log statements, on average 92,1%. The actual execution time of creating and indexing templates only takes 3 minutes and 16 seconds, around 7,9% of the total execution time. As comparison, Zhao et al. [44] static analysis takes less than two minutes to run for systems ranging from 100.000 to 300.000 lines of code.

Figure 4.1 details the time spent per file based on the lines of code per file, each point in the figure denotes a single file. The line represents the linear regression between the two

Step	Time (minutes)	Percentage
Finding log statements	37:25.618	92,1%
Creating template	00:32.928	1,3%
Enriching template	02:27.375	5,9%
Creating index	00:16.582	0,7%
Total	40:42.502	100%

Table 4.4: Mean execution time per individual step of the approach in minutes, average of ten runs.

variables, which clearly indicates an increasing trend between the lines of code per file and the execution time needed for that file. This also strengthens the previous observation that most of the execution time is spent on scanning through the source code looking for log statements.

RQ_2: The approach processes millions of lines of source code in approximately 40 minutes, of which on average only 7,9% of the time is spend on creating templates of the log statements.

4.3.3 RQ_3 - What is the performance of matching the log messages to its origin?

Figure 4.2 shows the statistical distribution of the time needed to find a link to the source code per log message. The median execution time to process a single log message is only 4 milliseconds. Table 4.5 details the statistics, which shows that 99% of the log messages are processed in under 132ms.

However, the total execution time to process all log messages is 01:13:19.969 hours, which is much higher than expected. If all messages from the dataset would be processed with the median execution time of 4 milliseconds, it would take approximately 6:40 minutes. The maximum time to process a single log message is exceptionally high: 2:19.922 minutes, or 35 thousand times the median. The approach derives an incorrect template from the statement in question. Two wildcards are missing from the template, which results in a template not able to match the log messages produced by the statement. This results that *all* templates are evaluated to still try to find a log statement for this specific message, thus causing such a long execution time. Therefore it is most important to create as precise as possible templates which greatly reduces the time needed to match the log messages.

RQ_3: The approach links for 90% of the messages to the source code within 6 milliseconds. For less than 1% of the messages an exceptionally longer amount of time is required, since all templates are evaluated. Improving the creation process of the templates resolves this issue.

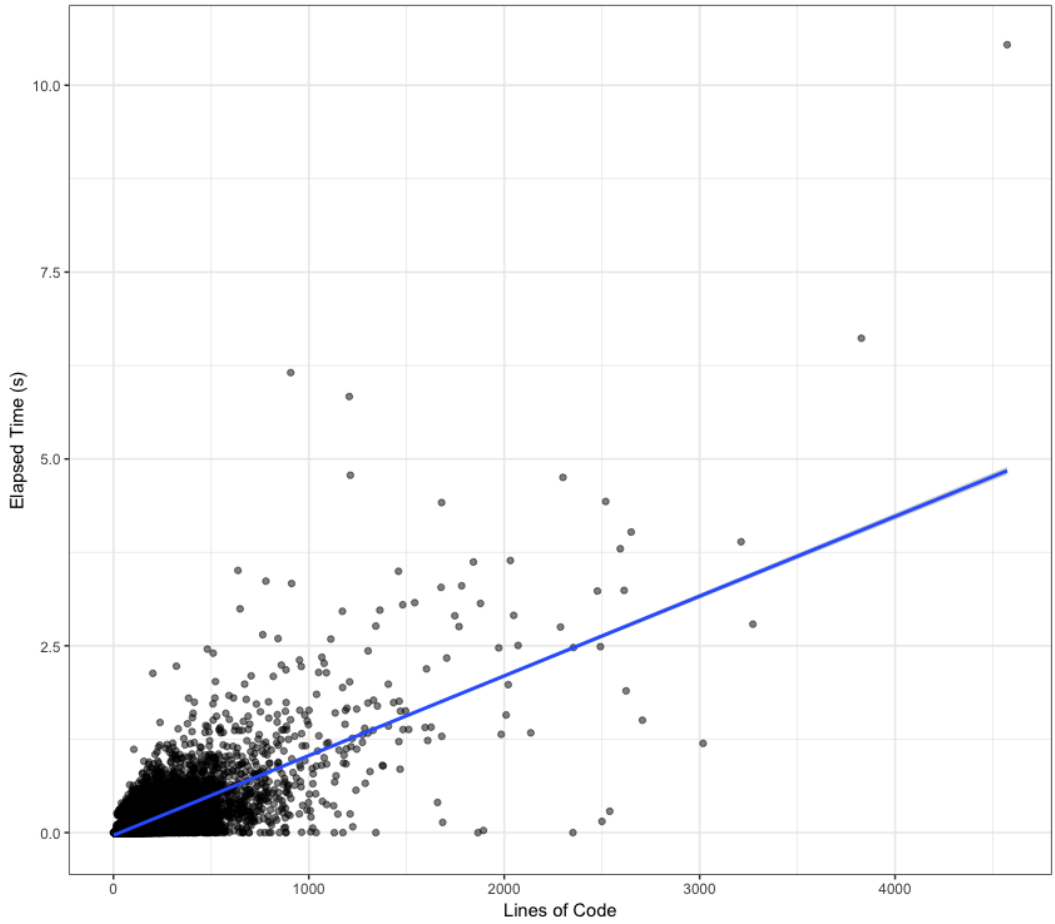

Figure 4.1: Execution time versus the lines of code per file.

Statistic	Time (ms)	Cumulative Time (%)
Minimum (0%)	3	0,00%
1st quartile (25%)	4	1,97%
Median (50%)	4	4,27%
3rd quartile (75%)	6	6,98%
90th quantile	6	9,61%
95th quantile	13	12,39%
98th quantile	59	15,22%
99th quantile	132	17,43%
Maximum (100%)	139922	100,00%

Table 4.5: Statistics of the execution time per log message in milliseconds.

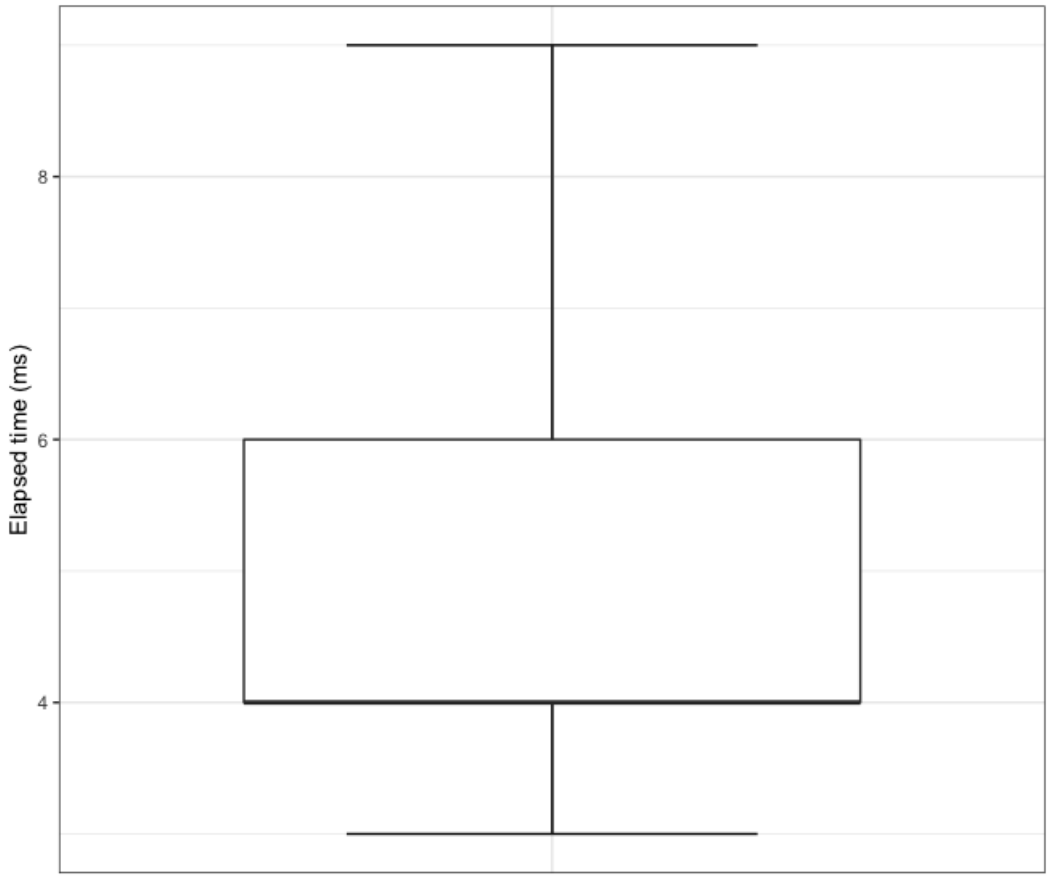

Figure 4.2: Time per log message to find template, with a median of 4 milliseconds.

Chapter 5

Discussion

5.1 Threats to validity

In this section we discuss the threats to the validity of our study and the course of action to mitigate them. We differentiate the validity into three categories: (1) internal, what could have affected the research itself, (2) construct, whether the evaluation method contains any form of bias and (3) external, the extent to which the results of a study can be generalised.

5.1.1 Internal validity

Manual sampling The decision of correctness of the individual samples were reviewed manually and could therefore have been subject to bias. We check whether the message could have been produced by the statement by taking into account the structure of the message, the severity and the accompanying name. We do this by first checking if the statement could logically have produced the message. To further reduce this threat, we reviewed the code base for log statements that could have generated the message to conclude the origin of the message with confidence.

5.1.2 Construct validity

Sampling links Sampling the messages according to their link could influence the accuracy score, since we choose to sample the sources of the messages to validate the accuracy of the approach. The number of log messages produced per log statement fluctuates significantly, as one log statement is responsible for 18,6% of all messages. If we would sample the log messages directly, we would risk evaluating the same messages repeatedly. To prevent this, we sample the origin of the messages, thereby reviewing each type of message once.

5.1.3 External validity

Single code base We have shown the results of applying the log parsing technique on the production systems at Adyen, however the logging statements might not capture all

logging behaviour expressed by developers of other systems. As we are evaluating the log statements of a single system, the developers could converge on a common style of writing logging statements only used within Adyen. The generation of log templates is directly influenced by this style, which could mean that the generation is in some kind of local optimum. At Adyen over 150 developers from all over the world with different backgrounds and ethnicities contribute to the source code, which shows that a diverse group has written and is using the information from log messages. Furthermore, table 4.2 shows that the standard logging methods are mostly used. Future work could include an evaluation on different sources to confirm the effectiveness of the approach.

5.2 Future Work

We split the future work up into improving the effectiveness and extending the usability. We first detail areas of improvement to further develop the approach and then discuss potential future directions for the approach.

5.2.1 Improving effectiveness

The approach uses static analysis to create templates of log statement that predict what the messages will look like at runtime. Unfortunately, not all templates completely capture all different messages that can be produced by the log statements. This section discusses the several improvements that can be made to the approach.

Unknown logging methods Currently methods are present in the source code that are not identified as logging methods. For example, developers introduce local logging methods for repeated operations within classes, as shown in Listing 5.1. The method adds additional information to the message before it is outputted as log. Many of these logging methods could exist in the source code which all have a specific way of formatting messages, unfortunately these are currently overseen by the approach. Such custom logging methods can be automatically recognised.

Evaluate entire log Different log statements use exactly the same arguments to construct a message, making it impossible to differentiate the exact source of a log solely based on the message. For example, consider the source code in Listing 5.2. When receiving the message `Error while inserting new mapping`, no distinction can be made between the log statements. However, when taking into account the throwable provided as second argument, we can decide exactly which statement is the origin of the message.

5.2.2 Extending usability

The approach shows a great potential for future applications, as it can be considered the foundation of log analysis techniques. This section discusses possible future directions for the approach.

```
protected void logInfo(String info) {
   log.info(getLogMessage(info));
}
protected void logInfo(String info, Exception e) {
   log.info(getLogMessage(info), e);
}
protected void logWarn(String info) {
   log.warn(getLogMessage(info));
}
protected void logWarn(String info, Exception e) {
   log.warn(getLogMessage(info), e);
}

private String getLogMessage(String info) {
   StringBuilder sb = new StringBuilder();
   sb.append(merchantAccount).append(":");
   sb.append(merchantReference).append(":");
   sb.append(skinCode).append(" - ");
   sb.append(info);
   return sb.toString();
}
```

Listing 5.1: Examples of unknown logging methods.

```
try {
    ...
} catch(IllegalStateException e) {
    log.warn("Error while inserting new mapping", e);
    addActionError("Duplicated key " + e.getMessage());
} catch (IllegalArgumentException e) {
    log.warn("Error while inserting new mapping", e);
    addActionError("Invalid value for field " + e.getMessage());
} catch (DbException e) {
    log.warn("Error while inserting new mapping", e);
    addActionError("There was an error while creating mapping");
}
```

Listing 5.2: Source code with duplicate log statements.

Higher throughput of log messages The approach is able to process an individual log message on average in 4 milliseconds. However, Adyen's production systems generate approximately 15 log messages per millisecond, 60 times as fast as the approach can currently handle. The approach can be extended to operate in parallel, achieving near linear speedup [40].

Other language support The approach uses the abstract syntax tree to navigate the source code to find log statements and create templates of their arguments. Currently the approach is focused on only Java, but this could be extended to include multiple languages. For this ANTLR [4] would be a viable solution, which generates a parser that can build and walk parse trees from a language grammar. This allows for great extensibility to other languages without having to write a separate implementation for each individual language.

Chapter 6

Conclusion

Log analysis techniques take advantage of the rich information present in logs, which are used for a broad range of purposes. In order to take advantage of this information, the log entries need to be traced back to its origin, the log statement in the source code that generated it. Several solutions based on clustering, heuristics, longest common sequence method, textual similarities and static analysis have been proposed to solve this non-trivial challenge, with the last category achieving the highest accuracy. However, it lacks evaluation in large scale environments for various industries.

To this end, we performed a field study of the approach that statically analyses the source code to relate log messages to its log statement. We extracted templates from the arguments of log statements and matched log messages to these templates, creating a link between log messages and statements.

The approach is evaluated on a rich and versatile dataset of logs produced by over thirty thousand log statements. We evaluated the approach according to the following research questions and answered them accordingly:

RQ$_1$ *What is the accuracy of the approach when dealing with extensive log data?* Our approach correctly identifies 239 out of 245 links to the source code, achieving an accuracy of 97,6%. Support can be easily added to the approach to correctly identify the source of the misidentified messages.

RQ$_2$ *What is the performance of creating the templates for the log statements in the source code?* The approach processes three million lines of source code in approximately 40 minutes, of which on average only 7,9% of the time is spend on creating templates of the log statements. The other time is used to search the source code for statements corresponding to log statements.

RQ$_3$ *What is the performance of matching the log messages to its origin?* The approach links for 90% of the messages to the source code within 6 milliseconds. For less than 1% of the messages an exceptionally longer amount of time is required, since all templates are evaluated. Improving the creation process of the templates resolves this issue.

We conclude that our approach is highly effective in tracing back the origin of a log data to its log statement in the source code, on average in 4 milliseconds and with an accuracy of 97,6% with a confidence level of 95% and interval of 5%. We provide a non-intrusive, adaptable to custom logging practices and easily extendable implementation that is ready to be used in large-scale system, which allows for automated log analysis techniques to be adopted. Since the approach relies on static analysis to predict the log messages, not all edge cases are completely and perfectly covered. As part of future work, we propose some concrete areas of improvement. We also asses possible directions for future applications for the approach, such as making it available for other languages and extending it to operate in parallel to achieve near linear speedup.

Bibliography

[1] *Adyen - Wherever people pay*. https://www.adyen.com/. Accessed: 2018-08-20.

[2] *Adyen 2017 revenue crosses $1 billion mark; increase of over $400m from 2016 - Adyen*. https://www.adyen.com/press-and-media/2018/adyen-2017-revenue-crosses-1-billion-mark-increase-of-over-400m-from-2016. Accessed: 2018-09-04.

[3] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[4] *ANTLR*. http://www.antlr.org/. Accessed: 2018-08-28.

[5] *Apache Lucene - Apache Lucene Core*. http://lucene.apache.org/core/. Accessed: 2018-08-31.

[6] Joop Aué et al. "An Exploratory Study on Faults in Web API Integration in a Large-Scale Payment Company". In: *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track*. 2018. DOI: 10.1145/3183519.3183537.

[7] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Vol. 463. ACM press New York, 1999.

[8] David Basin et al. "Scalable offline monitoring". In: *International Conference on Runtime Verification*. Springer. 2014, pp. 31–47.

[9] Christophe Bertero et al. "Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection". In: *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE. 2017, pp. 351–360.

[10] Ivan Beschastnikh et al. "Leveraging existing instrumentation to automatically infer invariantconstrained models". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 267–277.

[11] Hsin-Jung Cheng and Akhil Kumar. "Process mining on noisy logs—Can log sanitization help to improve performance?" In: *Decision Support Systems* 79 (2015), pp. 138–149.

[12] Min Du and Feifei Li. "Spell: Streaming parsing of system event logs". In: *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE. 2016, pp. 859–864.

[13] *Elasticsearch: RESTful, Distributed Search & Analytics — Elastic*. https://www.elastic.co/products/elasticsearch. Accessed: 2018-08-21.

[14] *ELK Stack: Elasticsearch, Logstash, Kibana — Elastic*. https://www.elastic.co/elk-stack. Accessed: 2018-08-21.

[15] Qiang Fu et al. "Execution anomaly detection in distributed systems through unstructured log analysis". In: *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE. 2009, pp. 149–158.

[16] Maayan Goldstein, Danny Raz, and Itai Segall. "Experience Report: Log-Based Behavioral Differencing". In: *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE. 2017, pp. 282–293.

[17] C.W. Günther and W.M.P. Aalst, van der. "Fuzzy mining - adaptive process simplification based on multi-perspective metrics". English. In: *Proceedings of the 5th International Conference on Business Process Management (BPM 2007) 24-28 September 2007, Brisbane, Australia*. Ed. by G. Alonso, P. Dadam, and M. Rosemann. Lecture Notes in Computer Science. Germany: Springer, 2007, pp. 328–343. ISBN: 978-3-540-75182-3. DOI: 10.1007/978-3-540-75183-0_24.

[18] Pinjia He et al. "An evaluation study on log parsing and its use in log mining". In: *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE. 2016, pp. 654–661.

[19] Pinjia He et al. "Drain: An online log parsing approach with fixed depth tree". In: *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE. 2017, pp. 33–40.

[20] *JavaParser - For processing Java code*. http://javaparser.org/. Accessed: 2018-08-27.

[21] Zhen Ming Jiang et al. "Automated performance analysis of load tests". In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE. 2009, pp. 125–134.

[22] *Kibana: Explore, Visualize, Discover Data — Elastic*. https://www.elastic.co/products/kibana. Accessed: 2018-08-21.

[23] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[24] *Log4j - Apache Log4j 2*. https://logging.apache.org/log4j/2.x/. Accessed: 2018-08-20.

[25] *Logstash: Collect, Parse, Transform Logs — Elastic*. `https://www.elastic.co/products/logstash`. Accessed: 2018-08-21.

[26] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. "A lightweight algorithm for message type extraction in system application logs". In: *IEEE Transactions on Knowledge and Data Engineering* 24.11 (2012), pp. 1921–1936.

[27] Salma Messaoudi et al. "A Search-based Approach for Accurate Identification of Log Message Formats". In: *International Conference on Program Comprehension (ICPC), 2018 IEEE/ACM International Conference on*. IEEE/ACM. 2018.

[28] Masayoshi Mizutani. "Incremental mining of system log format". In: *Services Computing (SCC), 2013 IEEE International Conference on*. IEEE. 2013, pp. 595–602.

[29] Karthik Nagaraj, Charles Killian, and Jennifer Neville. "Structured comparative analysis of systems logs to diagnose performance problems". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 26–26.

[30] Nicolas Poggi et al. "Business process mining from e-commerce web logs". In: *Business process management*. Springer, 2013, pp. 65–80.

[31] Ronald E Rice and Christine L Borgman. "The use of computer-monitored data in information science and communication research". In: *Journal of the Association for Information Science and Technology* 34.4 (1983), pp. 247–256.

[32] Weiyi Shang et al. "Understanding log lines using development knowledge". In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE. 2014, pp. 21–30.

[33] Yang Sun et al. "Personalized ranking for digital libraries based on log analysis". In: *Proceedings of the 10th ACM workshop on Web information and data management*. ACM. 2008, pp. 133–140.

[34] Mark D Syer et al. "Leveraging performance counters and execution logs to diagnose memory-related performance issues". In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE. 2013, pp. 110–119.

[35] Liang Tang, Tao Li, and Chang-Shing Perng. "LogSig: Generating system events from raw textual logs". In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM. 2011, pp. 785–794.

[36] Risto Vaarandi. "A data clustering algorithm for mining patterns from event logs". In: *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*. IEEE. 2003, pp. 119–126.

[37] Jan Martijn EM Van der Werf et al. "Process discovery using integer linear programming". In: *International conference on applications and theory of petri nets*. Springer. 2008, pp. 368–387.

[38] W Eric Wong et al. "Effective software fault localization using an RBF neural network". In: *IEEE Transactions on Reliability* 61.1 (2012), pp. 149–169.

[39] Wei Xu. “System problem detection by mining console logs”. PhD thesis. UC Berkeley, 2010.

[40] Wei Xu et al. “Detecting large-scale system problems by mining console logs”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 117–132.

[41] Wei Xu et al. “Online system problem detection by mining patterns of console logs”. In: *Data Mining, 2009. ICDM’09. Ninth IEEE International Conference on*. IEEE. 2009, pp. 588–597.

[42] Xiuming Yu et al. “Prediction of web user behavior by discovering temporal relational rules from web log data”. In: *International Conference on Database and Expert Systems Applications*. Springer. 2012, pp. 31–38.

[43] Ding Yuan et al. “SherLog: error diagnosis by connecting clues from run-time logs”. In: *ACM SIGARCH computer architecture news*. Vol. 38. ACM. 2010, pp. 143–154.

[44] Xu Zhao et al. “lprof: A non-intrusive request flow profiler for distributed systems”. In: *OSDI*. Vol. 14. 2014, pp. 629–644.

[45] Jieming Zhu et al. “Learning to log: Helping developers make informed logging decisions”. In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*. Vol. 1. IEEE. 2015, pp. 415–425.

[46] De-Qing Zou, Hao Qin, and Hai Jin. “Uilog: Improving log-based fault diagnosis by log analysis”. In: *Journal of Computer Science and Technology* 31.5 (2016), pp. 1038–1052.