Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

EXPLORING STATIC INTER-PROCEDURAL API MISUSE DETECTION USING GRAPH INLINING

Ruben Opdebeeck

2018–2019

Promotor: Prof. Dr. Coen de Roover
Advisor: Camilo Velazquez Rodriguez

Sciences and Bio-Engineering Sciences

Proef ingediend met het oog op het behalen van de graad van Master of Science
in Applied Sciences and Engineering: Computer Science

EXPLORING STATIC INTER-PROCEDURAL API MISUSE DETECTION USING GRAPH INLINING

Ruben Opdebeeck

2018–2019

Promotor: Prof. Dr. Coen de Roover
Advisor: Camilo Velazquez Rodriguez

Sciences and Bio-Engineering Sciences

Abstract

Programming against APIs is difficult. Application Programming Interfaces are often large, complicated, and specify implicit contracts which are not readily documented. Therefore, programmers often make mistakes when using API methods, which may lead to bugs and severe complications involving crashes, data loss, or security vulnerabilities.

To alleviate such difficulties, a lot of research has gone into automatically inferring API rules and usage patterns, and statically detect violations of such rules and patterns in a target project. Such tools are often called API misuse detectors. The state-of-the-art misuse detectors use pattern mining to infer correct usages of APIs, and identify misuses of the API as usages that deviate from the mined patterns. Recent work has explored graph-based representations to model such usages, but neglects inter-procedural analysis. This causes a significant number of false positives in their findings, due to missing graph elements that are present in transitively called methods. Such false positives decrease the precision of the tools, and is one of the reasons they are still difficult to use in practice.

In this dissertation, we present two main contributions to the field of static API misuse detection. As our first contribution, we incorporate inter-procedural analysis into graph-based pattern mining and violation detection through a process called graph inlining. This embeds the graph representation of a called method into the graph representation of its caller. To properly integrate the callee's graph, we must carefully consider code semantics, and thus, our approach considers many different types of control flow and data flow in its inlining algorithm. To customise this inlining process, we develop six inlining strategies that differ in the way they handle recursive calls, duplicate calls, and calls to methods that do not contain any direct API usage. These strategies lead to different patterns, and thus to different detected misuses.

Our second contribution is a general filtering algorithm to eliminate inter-procedural false positives. It is capable of removing incorrect violations due to missing pattern elements that are present in the call context surrounding the violating method, as well as removing duplicate violations that may occur due to inter-procedural analysis. This filtering algorithm inspects the callers and callees of a method to determine whether the violation reported in this method is likely a true misuse, and is based on the key insight that if a callee contains a violation, then all of its callers contain the same violation after inlining, and thus, the callers contain duplicated violations.

We evaluate our approach on a dataset of nine real Java projects containing actual misuses of APIs. We find that our approach effectively mines inter-procedural patterns, allowing it to identify new misuses, and also correctly eliminates inter-procedural false positives from its findings. We identify a number of important limitations of our work, including the prevalence of false positive misuses due to uncommon but correct alternative usages, as well as the increased cost of pattern mining and violation detection in the larger graphs produced by inlining. We investigate these limitations further, and pave the way to overcoming these issues in the future.

Acknowledgements

First and foremost, I would like to thank my promotor, Prof. Dr. Coen De Roover, whose positivism towards and during this thesis was of tremendous support. I would also like to thank my advisor, Camilo Velazquez Rodriguez, for the knowledge, insights, and remarks that he shared, which allowed me to complete this research successfully.

I would like to thank my family, for their continuous support, both through good times and, more importantly, through stressful times. Their confidence in me helped shape my own, and their reassurance during bleak times was incredibly powerful when I was overwhelmed.

Finally, I would like to thank my grandmother, who had been a constant source of love and support for the past 23 years of my life. She was always ready for a phone call and a chat when I was in dire need of a break, and although she would not understand a word written in this dissertation, she motivated me to go above and beyond. She regrettably passed away during the writing of this dissertation.

Rust zacht, moe'ke.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives and Contributions . . . 2
1.3 Overview of the Dissertation . . . 3

2 Background **5**
2.1 Mining Usage Patterns . . . 6
2.1.1 Frequent Itemset Mining . . . 6
2.1.2 Frequent Subsequence Mining . . . 9
2.1.3 Frequent Subgraph Mining . . . 10
2.2 Overview of Existing Bug Detectors . . . 12
2.2.1 Detecting Missing API Calls . . . 13
2.2.2 Detecting Call Order Violations . . . 14
2.2.3 Detecting Neglected Conditions and Error Handling . . . 17
2.2.4 Detecting General Violations . . . 19
2.3 Evaluating API Misuse Detectors . . . 22
2.4 Inter-Procedural Analyses in Misuse Detection . . . 25
2.5 Conclusion . . . 26

3 API Usage Graph Mining **29**
3.1 Structure of an AUG . . . 29
3.2 Constructing AUGs . . . 31
3.3 Pattern Mining and Violation Detection in AUGs . . . 33
3.4 MUDetect Evaluation . . . 35
3.5 Conclusion . . . 36

4 Inter-Procedural Analysis by Inlining Graphs **39**
4.1 Extended AUG Representation . . . 41
4.2 Graph Inlining . . . 43
4.2.1 Main Inlining Algorithm . . . 44
4.2.2 Inlining Strategies . . . 46
4.2.3 Inlining Individual Callees . . . 50
4.3 Inter-Procedural Filtering . . . 59
4.3.1 False Positives and Duplicate Violations . . . 59
4.3.2 Filtering Algorithm . . . 61
4.4 Conclusion . . . 64

5 Experimental Evaluation **67**
5.1 Recall Under Perfect Conditions . . . 68
5.1.1 Experimental Setup . . . 68
5.1.2 Results . . . 70
5.1.3 Discussion . . . 70

5.2 Precision of Top-20 Findings 72
5.2.1 Experimental Setup 72
5.2.2 Results 74
5.2.3 Discussion 78
5.3 Recall Under Realistic Conditions 83
5.3.1 Experimental Setup 84
5.3.2 Results 84
5.3.3 Discussion 86
5.4 Performance Evaluation 87
5.4.1 Experimental Setup 88
5.4.2 Results 88
5.4.3 Discussion 92
5.5 Threats To Validity 94
5.6 Conclusion 95

6 Conclusion **99**
6.1 Open Research Avenues 100
6.1.1 Extending Inter-Procedural Filtering 100
6.1.2 Behaviour-Oriented Representations 101
6.1.3 Relative Support Threshold 102
6.1.4 Performance Improvements 102
6.1.5 Extended Evaluation 102
6.1.6 General Applicability 103
6.2 Closing Statement 103

References **105**

List of Figures

2.1 Example source code to illustrate frequent itemset mining 7
2.2 Example of Figure 2.1 as a cross-table. 8
2.3 Example source code to illustrate partial call order 10
2.4 Examples illustrating frequent subgraph mining 12
2.5 Example source code to illustrate superfluous calls 16

3.1 Example of an API usage graph 30

4.1 High-level overview of our tool 39
4.2 Example illustrating the differences between AUGs and extended AUGs 41
4.3 Example of a call graph 43
4.4 Inlining order in a single method and a call tree 46
4.5 Overview of the inlining strategies 47
4.6 Examples illustrating undesired control flow changes in naive inlining . 53
4.7 Overview of data flow edge relinking during inlining 55
4.8 Overview of control flow edge relinking during inlining 57
4.9 Examples of an inter-procedural instance and violation of a pattern . . . 60

5.1 Empirical precision assessed in experiment P, as bar charts 75
5.2 Frequency of inter-procedural root causes of false positives 77
5.3 Example source code showing a real inter-procedural pattern instance and violation 80
5.4 Example source code showing a real self-usage 81
5.5 Example source code of a real false positive due to inlining depth 81
5.6 Recall for each detector, as a bar chart 85
5.7 Results of runtime measurements on depth-1 inlining 89
5.8 Results of runtime measurements on depth-2 inlining 90
5.9 Box plots of graph vertex set sizes 91

List of Tables

2.1 Precision, recall, and recall upper bound of four state-of-the-art misuse detectors . . . 23
2.2 Overview of existing tools that can be used to detect API misuses 28

3.1 Precision, recall, and recall upper bound of five state-of-the-art misuse detectors on an extended dataset . . . 35

4.1 Overview of the behaviour of the inlining strategies . . . 51

5.1 Overview of the datasets used in the experiments . . . 68
5.2 Hardware and software specifications of experiments . . . 69
5.3 Empirical recall upper bound assessed in experiment RUB . . . 70
5.4 Root causes of missed API misuses . . . 70
5.5 Selected Java projects for experiments P and R . . . 73
5.6 Empirical precision assessed in experiment P . . . 75
5.7 Detailed view of the number of true positives of three strategies on two projects . . . 76
5.8 Empirical recall assessed in experiment R . . . 85

List of Listings

3.1 AUG construction algorithm 32
4.1 Main inlining algorithm 45
4.2 Node inlining function 52
4.3 Inter-procedural filtering algorithm 63

1 Introduction

1.1 Motivation

DRY: Don't Repeat Yourself. This phrase is engraved in the mind of any competent programmer, and leads one to use abstractions, functions, modules, etc. in order not to repeat oneself. Some such abstractions are common across all projects of a certain language, such as data structures like linked lists, sets, or dictionaries; I/O operations, e.g. reading from an input stream or writing to a file; or date and time manipulations. To alleviate the need to re-implement each and every one of such abstractions for use in a new project, language designers and developers alike create, maintain, and provide libraries or frameworks, offering third-party Application Programming Interfaces (API) that can be used to implement the desired behaviour in a program. The availability of such libraries is enormous: Apache's Maven central artefact repository currently indexes over 180.000 packages for JVM-based programming languages, the Python Package Index (PyPI) contains over 200.000 third-party packages for the Python programming language at the time of writing, and the Node Package Manager (NPM) lists upwards of one million packages for use in the Node.js runtime environment for the JavaScript language.[1]

Developers make extensive use of such libraries, but often make mistakes in their implementation due to the difficulty of correctly using an API and integrating it into a project (Scaffidi, 2006). Root causes for such mistakes include a confusing API design, a lack of documentation, or pressing deadlines during development. Additionally, APIs often specify protocols that its users should follow for correct usage, yet these protocols are sometimes difficult to understand and keep track of (Sushine, Herbsleb & Aldrich, 2015). Failing to properly use an API may result in program crashes, data loss, or security vulnerabilities (Amann, Nadi, Nguyen, Nguyen & Mezini, 2016; Egele, Brumley, Fratantonio & Kruegel, 2013; Fahl et al., 2012; Georgiev et al., 2012).

Such problems led to an influx of research on development tools to warn programmers of potential bugs in their code. The earliest of such work (e.g., Hovemeyer & Pugh, 2004; Flanagan et al., 2002) uses a set of predetermined rules to statically detect violations in program code. However, continuously maintaining such rules for an ever-growing set of popular third-party APIs is practically infeasible. Thus, more recent research (e.g., T. T. Nguyen, Nguyen, Pham, Al-Kofahi & Nguyen, 2009; Monperrus & Mezini, 2013; Wasylkowski, Zeller & Lindig, 2007; Amann, Nguyen, Nadi, Nguyen & Mezini, 2019; Wasylkowski & Zeller, 2011; Acharya & Xie, 2009) instead focuses on mining patterns of API usage in program code, essentially relying on a programmer to break the DRY principle and repeat API usages sufficiently to be considered a pattern. A key insight is given by Engler, Chen, Hallem, Chou and Chelf (2001), who theorise

[1] Statistics gathered from `https://libraries.io/`.

that bugs are unlikely deviations from normal program behaviour, or, in other words, deviations from a pattern.

A recent study by Amann, Nguyen, Nadi, Nguyen and Mezini (2017) quantifies the effectiveness of four misuse detectors for Java, by running them on MUBench, their evaluation suite with a dataset of real projects containing actual misuses. Their work shows that these detectors have an extremely low precision and recall when run under non-perfect training conditions. The state-of-the-art misuse detector MUDetect (Amann et al., 2019) uses the insights gained from this study to improve upon these results, leading to a precision of 21.9% and recall of 20.9% when mining patterns from a single project. However, these results are still fairly low, which is in part caused by the absence of inter-procedural analysis: Over 15% of the false positives are due to intra-procedural analysis, where the detector for example reports missing program elements that are present in a transitively called method. Furthermore, MUDetect fails to find actual misuses when the misuse involves instance fields, since it deliberately removes such usages to prevent false positives due to intra-procedural analysis.

Bug detectors of the previous decade sometimes include inter-procedural checks during detection, or use inter-procedural pattern mining (Li & Zhou, 2005; Ramanathan, Grama & Jagannathan, 2007a; Lindig, 2016). In more recent years, this practice has dissolved, and most of the tools developed in the current decade employ intra-procedural analysis only. One exception is JSMiner (H. V. Nguyen, Nguyen, Nguyen & Nguyen, 2014), but their research focuses mostly on mining patterns, rather than detecting misuses. Thus, we need to devise new ways to incorporate inter-procedural analysis into the current API misuse detection techniques.

1.2 Objectives and Contributions

The main goal of our work is to alleviate the intra-procedural woes of the state-of-the-art misuse detectors. We divide this into two subgoals: First, we aim to prevent reporting false positives caused by missing program elements that are in fact present in the caller or callee of a program function. Second, we attempt to mine inter-procedural patterns of API usage which crosses method boundaries in a boundary-agnostic manner. This leads to patterns that capture more meaningful API usages, and allows implementations to be scattered across methods in arbitrary ways. To achieve our goal, we extend the state-of-the-art detector MUDetect by incorporating inter-procedural analyses. Concretely, our contributions are as follows:

- A graph inlining algorithm which embeds API usage graph representations of first-party callee methods into the graph of their caller. The resulting graphs, called *inlined API usage graphs*, capture larger views of API usage in a program as they encode the bodies of the caller, its callees, and potentially transitive callees. The graphs are method boundary-agnostic: They do not make any claims as to which program element should be present in which method body, thereby allowing arbitrary scattering of implementations, abstractions, method overloading, etc. The inlined API usage graph representation is general: Although we use it specifically to detect misuses, it can equally be used in pattern mining and API usage recommendation;
- Six unique inlining strategies that customise which method calls are inlined into a caller's graph, and up to which depth our algorithm should inline. These strategies differ in their handling of recursive calls, duplicate calls, and callees

that lack direct API usage. Different strategies may lead to graphs of different sizes, and may thus significantly impact the time taken to mine patterns and detect violations;

- A general, inter-procedural filtering strategy which removes both false positives in a method if the method's caller ensures that the pattern is fully instantiated, as well as duplicated reports that may arise due to our inlining;

- An evaluation of the effectiveness of our approach at removing inter-procedural false positives, as well as the precision and recall of the different inlining strategies, and the effect of the inlining and the choice of inlining strategy on the running time of our tool. We run our tool on a subset of the MUBench dataset and measure precision, recall, recall upper bound, and running time for each of our six strategies, and compare our results to MUDetect, our baseline, while providing an extensive discussion of the differences.

1.3 Overview of the Dissertation

The remainder of this dissertation is structured in the following manner. We start by introducing the reader to the field of misuse detection in Chapter 2. In Section 2.1, we first introduce the three main mining techniques employed by existing tools: Frequent itemset mining, frequent subsequence mining, and frequent subgraph mining. Section 2.2 continues by providing an extensive overview of previous work, categorised according to their capabilities. Afterwards, Section 2.3 describes the evaluation suite MUBench, which we use to evaluate our approach. Here, we additionally describe its dataset, the performance of four state-of-the-art tools on the suite, and insights gained from previous evaluations. Then, Section 2.4 reviews existing approaches to inter-procedural analysis in mining and misuse detection, and relates these to our work. Finally, Section 2.5 concludes by summary.

Chapter 3 continues our literature review by describing MUDetect, the tool our research extends to incorporate inter-procedural analysis. To provide the reader with a thorough understanding of the API usage graph representation, which is extensively used in our tool, we first describe its structure, node types, and edge types, in Section 3.1, whereas Section 3.2 details the construction of such graphs. Next, Section 3.3 explains how these API usage graphs are used in pattern mining and violation detection. We then review the evaluation of MUDetect on the MUBench evaluation suite in Section 3.4, and describe the main root causes of false positives and false negatives. Finally, Section 3.5 concludes this chapter.

Afterwards, Chapter 4 describes our inlining approach, inlining strategies, and inter-procedural filter algorithm. First, we introduce our extensions to the API usage graph representation in Section 4.1, and describe the differences to the original API usage graph by example. Section 4.2 explains our inlining algorithm, the different strategies, and the process of inlining a single method. We continue in Section 4.3 by describing our filtering algorithm, by first providing examples of common inter-procedural false positives, and then describing our rules and heuristics for eliminating these. We conclude this chapter in Section 4.4.

We describe our experimental evaluation in Chapter 5. Sections 5.1–5.3 cover the three experiments we perform to measure our detector's efficiency at finding API misuses. In Section 5.4, we introduce and perform an experiment to gain insights into the performance costs of inlining and inter-procedural analysis. In each of these sections,

we first describe our experimental setup, then present the results of the experiment, compare the results to a baseline, and finally provide an elaborate discussion of our findings. We list the threats to the validity of our experiments in Section 5.5. Finally, Section 5.6 concludes this chapter.

We conclude this dissertation in Chapter 6, where we provide an overall summary of our work in closing. We additionally provide an overview of future research possibilities in Section 6.1, and end this dissertation in Section 6.2.

2 | Background

Since the late 1990s, a large amount of research has gone into detecting bugs in API usages. The earliest of this work used a-priori rules, patterns, or templates provided by the user to statically determine violations. For example, FindBugs, a static bug detector for Java by Hovemeyer and Pugh (2004), ships with over ten likely bug patterns, ranging from null-pointer dereference checked by a simple data flow analysis, to inconsistencies in synchronisation. Similarly, Flanagan et al. (2002) detect bugs in Java programs using static formal verification and theorem proving. Using a number of ground truth rules, such as "a null-pointer should never be dereferenced", and programmer-supplied annotations specifying contracts on methods and class invariants, their tool ESC/Java can statically infer likely bugs as program code that violates the contracts or rules.

Ultimately, however, such approaches increase the burden on either the user or the programmer of a software artefact providing an API. A user needs to be aware of likely bug patterns to generate rules, programmers need to add annotations to their source code. Hence, research often focuses more on automatically inferring rules from one or more projects, and validating these rules against a target project afterwards. The majority of these approaches follow the same high-level overview: First, they extract an abstract model of the source code, then, this model is mined for frequently occurring patterns, and finally, the mined patterns are utilised to find violations in the project's code base. However, note that some tools do not perform explicit mining, or use a radically different approach overall.

Although many tools follow the same general overview, there are many different properties according to which bug detectors can be classified, such as their model representation, mining technique, capabilities of finding different types of bugs, etc. The most important of these is the distinction between static and dynamic analysis. Dynamic analysis tools employ runtime instrumentation to monitor a program, its state and its flow. As an example, DynaMine (Livshits & Zimmermann, 2005) mines revision histories of a code base to find likely coding patterns, and validates these patterns by running the program and observing how frequently they occur at runtime. It can then find violations of the patterns very accurately, since a violation of a pattern is guaranteed to happen, as seen in the dynamic analysis. Another example of dynamic analysis tools is Eraser (Savage, Burrows, Nelson, Sobalvarro & Anderson, 1997), which detects race conditions by monitoring the state of locking on memory locations shared across threads, and ensuring that consistent locking behaviour is used.

In this dissertation, we will instead focus on tools relying solely on static analysis to detect usage violations and hence, likely bugs. We first introduce commonly-used mining techniques in Section 2.1, and afterwards describe a large range of existing tools representing the state-of-the-art in misuse detection and pattern mining, categorised

according to their capabilities, in Section 2.2. Section 2.3 covers a misuse detection benchmark, which has been used to evaluate a number of tools, and is also the benchmark on which our tool has been evaluated. In this section, we summarise the root causes of low precision and recall of existing detectors, and identify that a lack of inter-procedural analysis relates to a significant portion of the detectors' false positives and false negatives. Thus, in Section 2.4, we explore existing approaches to incorporating inter-procedural analysis into pattern mining and violation detection tools, forming the inspiration of our work. Finally, Section 2.5 concludes with a summary.

2.1 Mining Usage Patterns

There exist many methods of mining for patterns in large datasets, depending on the shape of the data and the purpose that needs to be fulfilled. In general, a ***pattern*** is a frequently occurring phenomenon in the dataset. The ***support*** of a pattern is the frequency of its occurrence, and hence the term "frequently" constitutes that the support of a phenomenon exceeds some user-provided minimum support value. Mathematically, we will refer to the support of a pattern P in relation to a dataset D as $s_D(P)$. The exact formula to calculate the support depends on the representation of the dataset, pattern, and the mining technique itself.

We can further generally define a pattern as ***closed*** when there exists no larger pattern that subsumes the pattern and retains the same support. Mathematically, we say a pattern P is closed in relation to a dataset D when

$$\left\{P' \mid P \circ P' \wedge s_D(P) = s_D(P')\right\} = \varnothing.$$

The operator $\circ$ denotes the subpattern relation, and is again defined in terms of the actual representation of patterns.

In the remainder of this section, we consider three different mining techniques: Frequent itemset mining, its related association rule mining, and a formalisation based on formal concept analysis; frequent subsequence mining and its related sequence association rule mining; and frequent subgraph mining. For each of the mining techniques, we will provide an actual example of its usage.

2.1.1 Frequent Itemset Mining

A lot of the previous research (such as Michail, 1999, 2000; Li & Zhou, 2005; Thummalapenta & Xie, 2011; Ramanathan, Grama & Jagannathan, 2007b) employs ***frequent itemset mining*** to detect sets of items which frequently occur simultaneously in the dataset. Frequent itemset mining is often applied in shopping basket analysis (Agrawal, Imieliński & Swami, 1993) to mine for ***association rules***, such as "If a customer buys bread, she is likely to buy butter as well."

To illustrate, consider three transactions of three unique customers:

- Customer 1 buys bread, milk, and butter;
- Customer 2 buys bread, butter, and eggs;
- Customer 3 buys butter, flour, and chocolate.

```
void foo() {
    lock(l);
    // ...
    unlock(l);
}
```

(a) An instance

```
void bar() {
    lock(l);
    // ...
    // no unlock
}
```

(b) A missing method call

```
void baz() {
    unlock(l);
    // ...
    lock(l);
}
```

(c) Wrong call order

Figure 2.1 – Contrived example snippets of API calls `lock` and `unlock` on a lock `l` in a program written in a C-like language.

We can consider the collection of items bought in a transaction as an individual itemset, and the itemsets of all transactions to be the dataset. In this dataset, there exist three itemsets that occur at least twice: $\{\mathtt{bread}\}$, with a support of two, $\{\mathtt{butter}\}$, with a support of three, and $\{\mathtt{bread}, \mathtt{butter}\}$, with a support of two. The latter two are closed, since they cannot be extended to retain the same support, while the first itemset can be extended to obtain the last itemset with the same support.

In this simple example, we can see that customers often buy bread and butter together. However, this does not provide us any causal information: Do customers buy bread because they buy butter, or do they buy butter because they buy bread? This is where association rules come in. We can see that when a customer buys bread, they always buy butter, and when a customer buys butter, they only buy bread in two thirds of the cases. Hence, with a confidence of 100% in relation to our small dataset, we can say that the purchase of bread leads to a purchase of butter, and with a confidence of 66.67%, we can say that the purchase of butter leads to a purchase of bread.

Let us define this more formally. Our definitions of itemsets and association rules is based on the definitions given by Agrawal et al. (1993), adapted to be more consistent with definitions given in other works, such as Michail (1999). Consider an alphabet of items $\mathbb{I} = \{i_1, i_2, \ldots\}$. In the example, $\mathbb{I} = \{\mathtt{bread}, \mathtt{butter}, \mathtt{eggs}, \mathtt{milk}, \mathtt{flour}, \mathtt{chocolate}\}$. Using this alphabet, we can define an ***itemset*** $I_k \subseteq \mathbb{I}$ as an unordered set of items of the alphabet. An itemset database ISD, equivalent to the set of transactions in the example, is a set of itemsets $\{I_1, I_2, \ldots, I_n\}$, i.e. $ISD \subseteq \mathbb{I}^2$. The support of an itemset $I_p \in \mathbb{I}^2$ is defined as follows:

$$s_{ISD}\left(I_p\right) = \left|\left\{I_k \in ISD \mid I_p \subseteq I_k\right\}\right|.$$

In other words, the support of an itemset is the number of itemsets in the database that contain this itemset.

Association rules are then propositional formulae of the form

$$x_1 \wedge \cdots \wedge x_i \Rightarrow y_1 \wedge \cdots \wedge y_j,$$

with $x \in X$, $y \in Y$, $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \varnothing$, which can be written more compactly as $X \Rightarrow Y$. Such rules state that if items $x_i \in X$ are present in an itemset I_k, the items $y_j \in Y$ must be present as well. Support is also defined for association rules: An association rule $X \Rightarrow Y$ has a support value of $s_{ISD}(X \cup Y)$. This association rule is said to hold with confidence c% if c% of the itemsets that contain X in the database ISD, also contain Y.

Mining such association rules can be decomposed into two steps: Mining frequent itemsets of minimum support s_{min}, and generating rules from these itemsets with a minimum confidence c_{min}% (Agrawal & Srikant, 1994). Many algorithms exist for efficiently mining frequent itemsets (e.g., Burdick, Calimlim & Gehrke, 2001; Grahne & Zhu, 2003) and association rules (e.g., Agrawal & Srikant, 1994).

As a second example of association rule mining, consider the function defini-

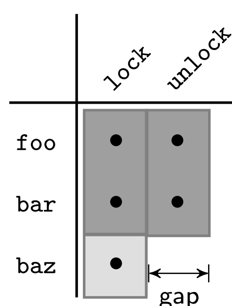


Figure 2.2 – Example of Figure 2.1 as a cross-table.

tions given in Figure 2.1. Assume `lock` and `unlock` are API functions which obtain and release a lock on their argument, respectively. Here, we define the alphabet $\mathbb{I} = \{\texttt{lock}, \texttt{unlock}\}$, and the itemsets $I_{\texttt{foo}} = \{\texttt{lock}, \texttt{unlock}\}$, $I_{\texttt{bar}} = \{\texttt{lock}\}$, and $I_{\texttt{baz}} = \{\texttt{lock}, \texttt{unlock}\}$. With closed frequent itemset mining with minimum support $s_{min} = 2$, we determine that `lock` and `unlock` are frequently called in the same function body, and `lock` is frequently called in general. Note that `unlock` is not present as a frequent call, since its frequent itemset is not closed. We can generate two association rules from these insights: $\texttt{lock} \Rightarrow \texttt{unlock}$ with confidence 66.67%, and $\texttt{unlock} \Rightarrow \texttt{lock}$, with confidence 100%. If we were to use these association rules for anomaly detection, we could report a likely bug in function `bar`, since it violates the first rule. The confidence of the rule can then be used to determine how likely the bug actually is. Note that function `baz` is not reported as a violation in any case, even though we implicitly know it is likely a bug because of the wrong order of calls. This is because we do not consider an order relation. However, we could change the representation of the items to include such information, called temporal properties (Wasylkowski et al., 2007). For example, it could state that a call to `unlock` is preceded by a call to `lock`.

An alternative mining technique is based on ***Formal Concept Analysis***, as used by Lindig (2016); Wasylkowski et al. (2007); Wasylkowski and Zeller (2011). A key insight is that itemsets represent a binary relation between the transaction (in our example, a function definition) and its features (the items in the itemset). These relations can graphically be modelled as a cross-table, which contains a dot whenever a transaction has a certain feature. We can then permute this table's rows and columns to group the dots into blocks that constitute patterns, which unifies patterns with their instances. The support of a pattern is then defined as the height of the block, while the width of a pattern is defined as the width of the block, i.e. the number of features involved in the pattern. Since a pattern is a set of features, with its instances being a set of transactions, there can be subpatterns which are subsets. These subpatterns are blocks with larger width, but smaller height. Likely violations of patterns can be identified as vertically-neighbouring blocks with lower width, leaving a gap in the larger block.

Figure 2.2 shows the example given in Figure 2.1 as such a cross-table. Using this graphical representation, we can clearly identify the pattern that a call to `lock` is paired with a call to `unlock`. This is marked in a dark gray shading in the illustration. The graphical representation also clearly shows the support of the pattern, i.e. the height of the shaded block. In the light gray shaded portion, we can see a violation of the pattern: A narrower, but taller block, which leaves a gap which if filled, would enlarge the pattern.

Since this technique is mostly a formalisation of frequent itemset mining, we do

not formally describe its representation or mining algorithm. We refer the interested reader to the work by Lindig (2016) for a formal description of the representation and algorithm, and to Ganter and Wille (1997) for the mathematical foundations of formal concept analysis.

2.1.2 Frequent Subsequence Mining

As we have seen in the previous section, frequent itemset mining can be used to detect patterns of frequently occurring sets of items, as well as causal relations between these items. However, frequent itemset mining cannot mine for temporal relations between items, and instead relies on its items encoding such temporal relations itself. To include such temporal information, existing tools (such as Xie & Pei, 2006; Zhong, Xie, Zhang, Pei & Mei, 2009; Thummalapenta & Xie, 2009; Acharya & Xie, 2009) instead mine for sequences of items, using ***frequent subsequence mining***.

Agrawal and Srikant (1995) provide an example of the applications of mining for sequential patterns in terms of a set of customer transactions in a video rental store. A pattern may consist of customers renting "Star Wars" movies, by first renting "Episode IV — A New Hope", then renting "Episode V — The Empire Strikes Back", and finally renting "Episode VI — Return of the Jedi". However, customers may rent other movies in between, and the same pattern would still be exhibited. Customers may also rent the same movie twice, for example, for an annual Star Wars marathon. The authors define item sequences as ordered sequences of itemsets, since a customer may rent both a Star Wars movie as well as another movie in the same transaction. In practice, however, the tools using frequent subsequence mining consider item sequences to simply be sequences of single items, which is equivalent to the original definition with singleton itemsets.

To formally define sequences and frequent subsequence mining, we generalise the definition given by Thummalapenta and Xie (2009) and include concepts from the definition given by Agrawal and Srikant (1995). ***Item sequences*** are sequences of events of the form $\langle e_1, e_2, \ldots, e_n \rangle$, where $e_i \in \mathbb{I}$, i.e. the event e_i is an item from the item alphabet, thus an item may occur multiple times in the sequence. Sequences are denoted more succinctly as $S = e_1 e_2 \ldots e_n$.

A sequence $S_i = e_1 e_2 \ldots e_n$ is a subsequence of $S_j = f_1 f_2 \ldots f_m$, denoted $S_i \sqsubseteq S_j$, if there exists integers $1 \leqslant k_1 < k_2 < \cdots < k_n \leqslant m$ such that $\forall 1 \leqslant l \leqslant n : e_l = f_{k_l}$. In other words, all events in S_i must be present in S_j, in the same order as they are present in S_i, but there may be gaps.

Sequences are stored in a sequence database SDB, containing tuples (sid, S), where sid is a unique sequence identifier and S is the sequence. The support of a sequence S with regards to the sequence database SDB is defined as

$$s_{SDB}(S) = \left| \{(sid, S_i) | S \sqsubseteq S_i\} \right|,$$

i.e. the number of super-sequences of S in the database. Definitions of frequent sequences and closed frequent sequences follow the same definitions as given in the beginning of the section, where the subpattern operator $\circ$ is the subsequence operator $\sqsubseteq$.

Using item sequences, we can define sequence association rules (Thummalapenta & Xie, 2009) in a similar manner to association rules in itemsets. A ***sequence association***

```
void qux() {
    foo();
    if (condition) {
        bar();
    } else {
        baz();
    }
}
```

Figure 2.3 – An example function body in a C-like language with branches in control flow, leading to its call order being a partial order.

rule is a propositional formula of the form

$$(e_1 \dots e_m) \wedge (f_1 \dots f_n) \Rightarrow (g_1 \dots g_k) \wedge (h_1 \dots h_l),$$

meaning that if an item sequence contains the subsequences $e_1 \dots e_m$ and $f_1 \dots f_n$, it should also contain the subsequences $g_1 \dots g_k$ and $h_1 \dots h_l$.

Recall the example given in Figure 2.1 of the previous section. From these examples, we can generate three item sequences representing call orders: `lock; unlock` for function `foo`, `lock` for function `bar`, and `unlock; lock` for function `baz`. Assume that the item sequence of `foo` is frequent in the dataset because of many other functions containing the same subsequence, while the other two sequences are not. Using frequent subsequence mining, we could then identify anomalies in both `bar`, because of the missing call, and `baz`, because of the wrong call order.

Note that frequent subsequence mining requires a total order on the items in a sequence. For call order, this can be problematic, since function bodies often only provide a partial call order of its callees in the presence of branches in control flow. Consider for example the function body given in Figure 2.3, which contains a branch based on a boolean variable `condition`. We cannot provide a total call order for the callees here, since `bar` is not called before `baz`, nor is `baz` called before `bar`. Hence, we can only define a partial order, where `foo` is called before both `bar` and `baz`, and `bar` and `baz` are incomparable. To provide a total order to frequent subsequence mining, one could instead opt for a path-sensitive analysis, which will provide two item sequences for the function `qux`: `foo; bar` and `foo; baz`. Alternatively, one can opt for frequent subgraph mining instead, as shown in the next subsection.

2.1.3 Frequent Subgraph Mining

If we only have a partial order on items, we can opt to represent our collection of items as a labelled directed acyclic graph. Furthermore, frequent itemset mining and frequent subsequence mining are often applied to collections of homogeneous items, resulting in these techniques only being able to represent patterns of one kind of phenomena, e.g., call order, or caller-callee relations. If we instead want to represent more complex patterns, for example, patterns incorporating both syntactical constructs such as loops, if-then-else statements, or try-catch constructs, as well as API calls and call order, we need a richer representation. This can be achieved using labelled graphs, where a label on a vertex represents the called API function or syntactic construct, and edges can represent temporal orders, data dependencies, or any other relation that is interesting to mine for.

Discovering patterns in graph-based representations is performed through *frequent subgraph mining*, and is implemented in a number of previous works (e.g., T. T. Nguyen

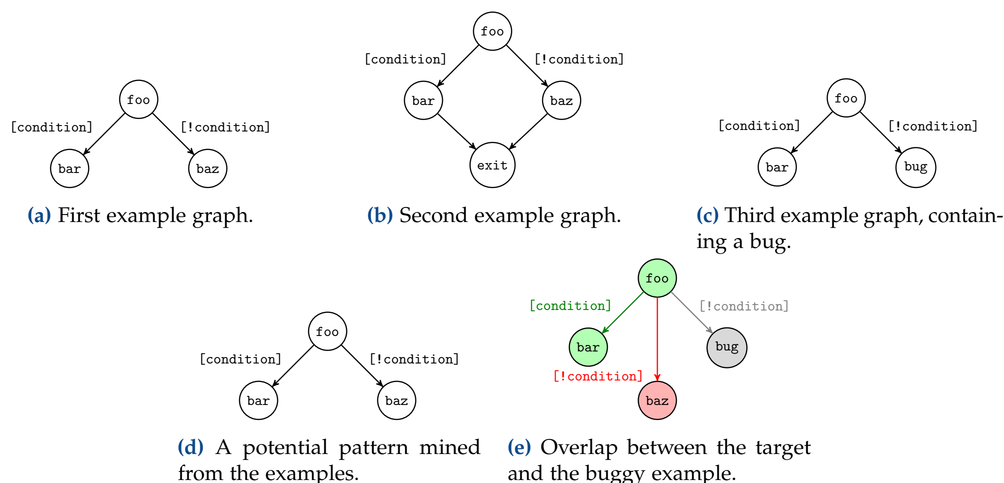
et al., 2009; Chang, Podgurski & Yang, 2007; Amann et al., 2019). Our definition of the problem is a generalisation of the definition given by T. T. Nguyen et al. (2009). We are given a graph database $GDB = \{G_1, G_2, \ldots\}$, where G_i is a labelled directed multigraph with vertex set $V(G_i)$ and edge set $E(G_i)$ with labels for both nodes and edges. Note that we keep our definition general, allowing graphs to contain multiple edges between two nodes, and being assigned labels for both its nodes and edges. This generality is necessary to cover all possible graph representations used in the tools.

Consider a candidate pattern graph G_c. An *occurrence* of G_c in a graph $G_i \in GDB$ is an induced subgraph SG_i of G_i which is label-isomorphic to G_c. We additionally say that two occurrences of SG_{i_1} and SG_{i_2} of G_c in G_i are independent if their vertex sets are disjoint sets, i.e. the two occurrences share no nodes. The support $s_{GDB}(G_c)$ of candidate pattern graph G_c in relation to the database GDB is the number of independent occurrences of G_c in the graphs of the database. This means that a candidate pattern can be supported by an instance multiple times, if the instance contains multiple occurrences of the pattern. Graph patterns can be closed, following the same definition as given in the beginning of this section, where the $\circ$ operator now denotes that a graph is an independent occurrence of another graph.

Since subgraph mining needs to solve the subgraph isomorphism problem, which is NP-complete, it follows that frequent subgraph mining is expensive. Mining is often performed through a-priori pattern growth techniques (Inokuchi, Washio & Motoda, 2000), which involves a number of steps. First, given a number of frequent subgraphs (initially containing no nodes), it generates possible extensions to these candidates, leading to new candidates. This is the growth step. Afterwards, candidate patterns are pruned based on viability: If it can already be determined that a candidate pattern will not lead to a meaningful final pattern, it can be removed from consideration in order to reduce the search space. This pruning phase is vital to make frequent subgraph mining feasible: The number of possible extensions can increase exponentially at each step of candidate generation.

The a-priori approach is based on the key insight that if a candidate pattern is infrequent, none of its extensions can be frequent, and algorithms thus prune candidates when they drop below the minimum support value. To determine the frequency of candidate patterns, the patterns first need to be clustered into isomorphic groups, after which the frequency can be determined for a single group. Note that candidates in the same cluster may be obtained from different graphs, or through different extensions of the same graph. Hence, afterwards, candidates are split up again for the next round of pattern growth. When none of the candidates can be extended further to larger frequent subgraphs, a final frequency calculation is performed, and the patterns are obtained.

As an example, consider the graphs given in Figure 2.4. In these graphs, nodes represent calls to a function, where the target function is given as a node label. Edges represent the partial order in calls. A branching point in the graph represents a branch in the control flow of the body, and the condition for taking this branch is annotated as an edge label. Figure 2.4a is a representation of the partial call order of the function body given in Figure 2.3. Figure 2.4b is the representation of a similar function that additionally calls `exit` after the conditional. Figure 2.4c is the representation of a similar function that calls `bug` instead of `baz`. We can identify a frequent subgraph with a support of 2, as represented in Figure 2.4d. The first two example graphs contain an instance of this pattern, while the third example contains a partial overlap. This overlap is shown graphically in Figure 2.4e, where overlapping nodes and edges between the pattern and the target example are shown in green. The green portion represents



(a) First example graph.

(b) Second example graph.

(c) Third example graph, containing a bug.

(d) A potential pattern mined from the examples.

(e) Overlap between the target and the buggy example.

Figure 2.4 – Example of graph representations of method bodies derived from the body in Figure 2.3, a related graph pattern, and a violation.

a common label-isomorphic subgraph of the pattern and target. Note that the call to `baz` in the pattern is absent in the target, which is represented in red. The gray portion represents nodes and edges in the target that cannot be found in the pattern. The violation can be found in the red portion, since features that are required by the pattern are not present in the target. The gray portion is not necessarily a violation, however, since the call to `exit` in the second example would similarly not be found in the pattern. Having superfluous elements in the graph cannot be regarded as a violation of a pattern, as patterns are obtained as frequent subgraphs of the methods' graphs themselves. Thus, if we were to report a violation because of a superfluous element, we would essentially require all of a project's methods to be identical to other methods to not be considered a violation.

2.2 Overview of Existing Bug Detectors

We now introduce a number of tools related to detecting bugs in API usages and classify them according to their capabilities. Note that detecting violations in API usage is closely related to the fields of mining and recommending API usage patterns. Indeed, these three fields share a common goal of finding meaningful similarities in API usage across one or multiple projects. Pattern mining focuses on identifying these similarities and representing them as patterns, which can then later be used for analysis, recommendation or violation detection. API usage recommendation uses these similarities to infer likely code completions given a partial implementation, while API usage violation detection attempts to flag likely bugs in the usages.

Hence, these three fields often share the same techniques to reach their goal, many of which have been covered in the previous section. For example, given a number of mined API usage patterns, one can attempt to detect instances of these patterns in a target code base. Imperfect instances, i.e. code snippets that are missing one or more elements of the pattern, may be API usage violations. However, not every imperfect instance is a real violation, leading to the possibility of false positives. Thus, existing tools use a range of different ranking strategies, based on e.g. the support of the pattern or the distance between the pattern and target snippet.

Patterns can also be used in usage recommendation, where tools can suggest a pattern if the partially implemented code snippet shares enough similarity with it. However, usage recommendation often relies on statistical methods to rank recommendations, such that the recommendations the programmer would likely find more useful end up ranking higher. To this end, an often used representation is a probabilistic model (T. T. Nguyen, Pham, Vu & Nguyen, 2015; Bielik, Raychev & Vechev, 2016; A. T. Nguyen & Nguyen, 2015) which may, for example, assign probabilities to the occurrence of a certain call sequence and can then rank its code completion recommendations according to these probabilities. Such models can be used for bug detection as well, such as by querying the model to retrieve the probability of a certain call sequence, and flagging a likely violation in the sequence if the probability of its occurrence is lower than a certain threshold.

Because of the relations between these three research fields, we will additionally describe certain tools that could be capable of detecting API usage violations, even though they are not designed to do so. We categorise these tools according to their general capabilities: Some tools are able to detect missing method calls, some other tools are able to detect malformed call sequences, containing both missing and misplaced calls. A number of tools specialise in detecting missing and malformed semantic constructs, such as missing conditions or faulty exception handling. They may also be able to detect missing or misplaced method calls if these method calls are involved in their targeted semantic constructs. The final category contains tools that are able to detect all of these: Missing or misplaced calls, as well as missing or malformed semantic constructs.

2.2.1 Detecting Missing API Calls

A common cause of bugs in a program is a missing call to a certain API method, such as the absence of a call to the `hasNext` method of a Java `Iterator` to ensure a subsequent element exists before a calling the `next` method to retrieve this element. Several detectors specialise in identifying such missing calls in a method body.

PR-Miner (Li & Zhou, 2005) detects missing API calls in C function bodies. It generates association rules using frequent itemset mining on a function body's program elements, namely variables, function calls, and data types. These rules then specify that when a set of such program elements is used in a function body, another set of program elements must be used as well. Violations are flagged when a method body uses the program elements in the antecedent of the rule, but not all program elements in the consequent, and the rule has a sufficiently high confidence. PR-Miner additionally performs an inter-procedural detection by searching a method's callers or callees if a program element is missing, in order to reduce false positives. Note that since PR-Miner considers calls to any function in its itemsets, it is able to mine usage patterns relating to both first-party and third-party APIs.

Colibri/ML (Lindig, 2016) detects missing calls using formal concept analysis. It considers the caller-callee relation of the whole program as its input relation, which is then transformed into a cross-table, as described in Section 2.1.1. Missing calls can then easily be identified as "gaps" in this cross-table. Notably, Colibri/ML performs inlining on its input relation, by adding, for each function, the cross-table entries of its callees as well. In other words, the resulting cross-table contains entries for each direct and indirect callee of this function. Additionally, if every caller of the function calls the same other function, such an entry is added as well. In the end, the cross-table then contains all pairs of functions that are always called together in a program. Hence,

COLIBRI/ML is able to mine inter-procedural patterns involving both first-party and third-party functions, and naturally detect violations inter-procedurally as well, since it unifies patterns, its instances, and violations of the pattern.

DMMC (Monperrus & Mezini, 2013) detects missing method calls in Java as violations of the ***majority rule***. It extracts ***type usages*** from every method body in a source program. A type usage for variable x is a triple (T, C, M), where T is the type of the variable, C is the context, i.e. the signature of the method in which x is present, and M is the set of methods called on x. Hence, type usages do not include call order or control structure information. Type usages participate in two binary relations. Two type usages for variables x and y are ***exactly-similar*** if they have the same type, the same set of methods, and are used in a similar context. They are ***almost-similar*** if they have the same type, are used in a similar context, and y contains one extra method, i.e. $M_x \subset M_y$ and $|M_x| = |M_y| + 1$.

Each type usage is assigned a ***strangeness score*** or S-score, which is based on the number of exactly-similar and almost-similar type usages. Following the majority rule, if a type usage x has many almost-similar type usages, but a low number of exactly-similar type usages, it is a strange usage, as it diverges from the majority. Such strange type usages are flagged as potential misuses. DMMC also recommends which method call is likely missing from the type usage. It ranks the potential fixes using the majority rule again: The more almost-similar type usages contain the extra method, the higher the rank as a potential fix.

CODEWEB (Michail, 1999, 2000) is an API usage pattern miner which defines items as reuse relationships in object-oriented class hierarchies. Concretely, it adds items for class inheritance, class instantiation, method overriding, and method calling. Association rules can then state that when a class inherits a certain base class, it should also override a certain method of the base class, or when it calls a certain method, it should also call another method. Although CODEWEB does not specifically detect misuses in class code, their patterns could potentially be used to detect missing method calls. Not only could their rules identify missing calls in the presence of other calls in a method body, it could also identify missing method definitions in the presence of inheritance, for example, when a framework specifies that inheriting classes should always override a specific instance method.

PARSEWEB (Thummalapenta & Xie, 2007) is an API usage pattern miner and recommender that answers queries of the form "Source object type → Destination object type" by providing a sequence of method calls that lead from an object of the source type to an object of the destination type. Such patterns can potentially be used to identify missing calls as well. The tool extracts call sequences by transforming an AST to a directed acyclic graph, then extracting the shortest path in the DAG that connects an object of the source type to an object of the target type. Sequences are clustered according to their similarities, and ranked according to their frequency (size of the cluster) and sequence length. Although method invocations are returned as a sequence, the clustering considers sequences with a different order to be similar, and hence ordering information is lost in the end.

2.2.2 Detecting Call Order Violations

The previously covered detectors are able to identify missing API calls, but are unable to identify API call order violations as they do not specifically encode temporal order relations among calls. For example, even though they are able to detect a

missing call to `hasNext` of a Java `Iterator`, they are unable to identify the call order constraint that `hasNext` must be called before `next` in order for a program to be safe. For instance, if a code snippet calls both of these methods in a loop, but fails to call `hasNext` on the first iteration to handle the case where an iterator is empty, the previously covered detectors do not identify a violation. To detect call order violations, detectors need to encode relations that specify that a call is always executed before or after another call. This is often done through frequent itemset mining by encoding temporal properties, or frequent subsequence mining which already encodes ordering information. Additionally, one tool is able to detect call order violations using a probabilistic model that represents likely call sequences.

Call Order Violation Detection Using Itemsets

JADET (Wasylkowski et al., 2007) builds further on top of Colibri/ML to detect such violations in call order. To do this, it first generates finite state machines to model method bodies, where states are locations in a method body and transitions are labelled with the program statement, such as a method call or variable assignment, that takes place between the two locations. From these models, it extracts temporal properties in the form of $m \prec n$, stating that m may be called before n. It then mines for patterns among these temporal properties using formal concept analysis, with an input relation that relates a program method to such a property extracted from the method. Because of formal concept analysis, this mining immediately provides violations of the patterns as well. Violations are flagged if there is sufficient confidence, which is based on the ratio of potential violations to pattern instances.

Tikanga (Wasylkowski & Zeller, 2011) further extends JADET by adopting *Computation Tree Logic* (Clarke, Emerson & Sistla, 1986) formulae as the representation of features, rather than simple call order properties. This allows it to capture how objects are used before being passed to a method call, and generate more complicated constraints when compared to JADET. As a result, Tikanga can generate more expressive call order constraints and therefore detect more complicated violations, but its base capabilities are the same as JADET's.

Call Order Violation Detection Using Item Sequences

Chronicler (Ramanathan et al., 2007a) detects missing and misplaced calls in C function bodies. Through path-sensitive analysis, it generates call precedence constraints between two calls in a single function body, for example, $a \overset{f}{\leftarrow} b$ indicates that a call to b is always preceded by a call to a in the function f. Compositions of these rules are memoised as *function summaries* for each function declaration, allowing inter-procedural analysis by extending precedence constraints at call sites using the function summary of the callee. Chronicler then uses frequent subsequence mining to detect patterns in these control-flow constraints. As such, it can find both missing calls and call order violations when the mined constraints are unsatisfied.

MAPO (Xie & Pei, 2006; Zhong et al., 2009) mines for frequent call sequences as patterns in API usage. For each method in the project, it extracts sequences of method calls, incorporating inter-procedural mining by inlining call sequences. During extraction of call sequences from a method body, when MAPO encounters a first-party method call, it performs a lookup of the method, scans this new method's body for call sequences, and extends the call sequence with this new method's calls rather than

```java
void m(Iterator<String> it) {
    if (it.hasNext()) {
        String first = it.next();
        String second = it.next();
    }
}
```

Figure 2.5 – Example of a superfluous call in a Java method.

the original call itself. This inlining process is recursive, and call sequences therefore incorporate API call sequences of transitively called methods as well. The end condition for this recursive process differs for the two versions of MAPO, and is discussed in more detail in Section 2.4. The resulting API call sequences are mined using frequent subsequence mining. Although MAPO makes no attempt at detecting misuses of these frequent call sequences, a detector based on its patterns may be able to detect both missing and misplaced method calls.

The first version of MAPO (Xie & Pei, 2006) is path-insensitive: It extracts sequences in call-site order, ignoring conditional statements and control flow, thereby inducing a total order on API calls, which can easily be fed into the frequent subsequence miner. The second version of MAPO (Zhong et al., 2009) performs a path-sensitive extraction, extracting a sequence for each path through a method body, accounting for control flow structures such as conditions and loops. Furthermore, before frequent subsequence mining, the second version clusters API call sequences according to the fully qualified name of the method from which the sequence was extracted, and the methods called in the snippet. This clusters snippets that likely exhibit the same behaviour, based on similarities in their method name, class name, and involved API methods. Using this clustering, the second version of MAPO can mine different patterns for different programming contexts involving the same methods. For each of the clusters, it runs frequent subsequence mining separately to generate patterns, which are later fed into a recommender system that can be queried for relevant patterns.

Call Order Violation Detection Using Probabilistic Models

DroidAssist (T. T. Nguyen et al., 2015) is an API usage recommender for Android projects. It trains a *Hidden Markov Model of API usage* (HAPI) using call sequences extracted from graph-based representations called groums (see Section 2.2.4) in a path-sensitive manner. The HAPI can then be used to recommend the next method call in a partial implementation of an API usage. The authors note that this HAPI can be used to detect misuses, by querying the probability of a certain method call sequence and flagging the sequence as a violation if the probability is very low. However, this detection strategy has not been evaluated. An interesting property of DroidAssist is that, in addition to detecting missing and misplaced method calls, it is able to detect superfluous calls. Consider for example the duplicated call to the `next` method of an iterator, shown in Figure 2.5. None of the previously covered detectors, nor any of the detectors that are yet to be covered, can identify a violation: Both calls to `next` are preceded by a call to `hasNext`, and the call to `hasNext` is always followed by a call to `next`. If the HAPI is properly trained, it will assign a very low probability to a duplicated call to `next`, and can therefore identify a violation.

2.2.3 Detecting Neglected Conditions and Error Handling

APIs often require certain conditions to be satisfied before a method may be called. Examples include certain arguments that may not be `null`, or receiver objects that must be in a certain state. Failure to adhere to these conditions may result in undefined behaviour or crashes, leading to bugs in the program. Furthermore, API methods can fail due to external conditions, such as non-existent files or timeouts in network connections, and signal the caller of these failures either by returning specific values to the caller (e.g., non-zero return values in C), or raising an exception. Failure to handle such errors correctly may lead to serious bugs, such as memory leaks by neglecting to free memory on a program path, inconsistent states in external resources by e.g. forgetting to close a file handle or improper rollback of transactions in a database, or even program crashes due to uncaught exceptions or continued use of data in undefined states.

Thus, proper condition checks and exception handling are vital for the correctness and stability of a program, yet are often a source of bugs. Various tools have been developed to detect anomalies in these semantic constructs. Note that these tools often are also able to detect missing or misplaced calls, but in a less general context compared to the previous tools, and only identify such misuses when pertaining to their specifically targeted semantic construct.

Detecting Missing and Faulty Conditions

Alattin (Thummalapenta & Xie, 2011) detects missing conditions surrounding API call sites, accounting for the many different ways in which a condition can be specified. For example, when iterating over a Java collection, one can either use the `hasNext` method on an iterator, or maintain a separate counter and check its value against the size of the collection itself. To account for these alternatives, Alattin extracts for each call site to an API method F_k, the conditions surrounding the call site. As an example, for the method `Iterator.next`, such condition check may be "`boolean` check on return value of `Iterator.hasNext` before call to `Iterator.next`". It then mines these extracted conditions for rules of the form

$$P_1 \vee \cdots \vee P_i \vee A_1 \vee \cdots \vee A_j.$$

In this formula, $P_1, \ldots, P_i$ are frequent rules, and $A_1, \ldots, A_j$ are infrequent alternative rules.

The tool mines such rules in two phases for each API method F_k. For each distinct call site of F_k, the extracted conditions are stored as an itemset in the itemset database ISD_{F_k}. In the first phase, Alattin mines this database using frequent itemset mining, which generates the frequent patterns $P_1, \ldots, P_i$. In the second phase, it partitions the database into two disjunct databases based on the previously-mined patterns. The positive itemset database PSD_{F_k} contains the itemsets that contain a frequent pattern, the negative itemset database NSD_{F_k} contains the itemsets that do not contain a frequent pattern. Then, it performs another pass of frequent itemset mining on the negative itemset database, leading to the alternative patterns $A_1, \ldots, A_j$, which are frequent in NSD_{F_k}, but infrequent in ISD_{F_k}. Finally, this leads to a list of frequent patterns and infrequent alternative patterns, which can be regarded as an association rule of the form $F_k \Rightarrow P_1 \vee \cdots \vee P_i \vee A_1 \vee \cdots \vee A_j$, stating that for a call to F_k, one of the patterns must hold. A violation is flagged if a call site of F_k matches none of the frequent patterns or infrequent alternative patterns.

Ramanathan et al. (2007b) extend the CHRONICLER tool, covered in Section 2.2.2, and focus on detecting missing and faulty conditions. As in the original tool, their work extracts precedence constraints among API calls. Additionally, they extract data flow properties for data values involved in API calls, such as comparisons involving the variable, which they translate into data constraints. As in CHRONICLER, frequent subsequence mining is applied on the precedence constraints, but an additional frequent itemset mining pass is performed on the data constraints. As a result, their tool can detect both patterns in call order as well as common constraints on the call arguments. Using these patterns, they detect neglected and flawed conditions on variables involved in an API call.

Chang et al. (2007) detect missing and malformed conditions by modelling a program as a *Program Dependence Graph* (PDG) (Ferrante, Ottenstein & Warren, 1987). A program dependence graph is a graph that represents two types of dependencies between program statements: Data dependencies, as identified using definition-use analysis, and control dependencies, where a statement is control-dependent on another statement if the latter controls the branch in which the former resides. Conditional rules are represented as graph minors of PDGs, and mined using a combination of frequent itemset mining and frequent subgraph mining. Frequent itemset mining is used to determine the node labels that are frequent enough to occur in a frequent subgraph, thereby reducing the search space. After graph minor patterns are mined, they are presented to the user for validation, and confirmed patterns are used in violation detection by identifying subgraphs of the PDG that are similar, but not isomorphic to, the pattern.

Detecting Missing and Malformed Exception Handling

CAR-MINER (Thummalapenta & Xie, 2009) specialises in mining patterns in exception-handling code and subsequently detecting violations. The authors identify that certain exception-handling routines are only necessary under certain normal control flow scenarios. For example, when interacting with a database, the code for setting up the database connection may be identical for different types of database operations. Transaction rollback is only necessary if the operation modifies the database, whereas merely querying the database does not require a transaction rollback. To properly distinguish such scenarios, CAR-MINER generates sequence association rules of the form

$$\left(C_c^1 \dots C_c^n\right) \wedge C_a \Rightarrow C_e^1 \dots C_e^m.$$

Such rules specify that a function call C_a should be followed by a sequence of function calls $C_e^1 \dots C_e^m$ when C_a is preceded by a sequence of function calls $C_c^1 \dots C_c^n$. In our database example, the sequence $C_c^1 \dots C_c^n$ corresponds to the database connection setup, C_a corresponds to a database update call, and $C_e^1 \dots C_e^m$ corresponds to the transaction rollback in exception-handling code.

To mine such rules, the tool first generates an exception flow graph, an extended form of control flow graph which includes edges to represent exceptional flow. Using this exception flow graph, CAR-MINER generates normal and exceptional control flow traces. It then extracts the call sequences $C_a^1 \dots C_a^n$ from normal control flow traces, and the exception-handling call sequences $C_e^1 \dots C_e^m$ from exceptional control flow traces. Using frequent subsequence mining on the extracted normal and exceptional call sequences, they can generate the previously described sequence association rules. The patterns are then used to detect violations, which are detected in a similar manner to detectors using normal association rules.

Acharya and Xie (2009) also detect missing and faulty error-handling code. Using push-down model checking, they extract inter-procedural program traces for normal control flow and exceptional control flow as finite state machines. From these traces, they extract usage scenarios by considering definition-use data flow information in order to generate data dependent call sequences. The data dependence information is used to separate potentially interlaced traces, so that exceptional flow stemming from API usage can be isolated. They then mine these call sequences using frequent subsequence mining, and use them to detect violations using another round of push-down model checking. False positives are removed through reachability analysis, to exclude error cases that cannot occur.

Weimer and Necula (2005) mine patterns in exception-handling code using finite state automata to represent programming rules. They limit themselves to two-state machines, to generate rules that remain simple, and can be accepted or rejected by users, rather than requiring debugging or editing. The states of these finite state machines are events, i.e. calls, leading to rules that specify that e.g. `close` can only be called when a resource is open, and `open` can only be called if it is closed. Hence, it can also identify redundant calls as violations. They require their mined finite state machines to have their second event occur in at least one, but not all exception-handling traces. This allows them to mine rules that are important to the program, since it occurs at least once, but has errors, since there are cases where it does not occur.

2.2.4 Detecting General Violations

The previous approaches focus on detecting one type of violations, and limit their representations to consider only one kind of facts. One exception is the work by Ramanathan et al. (2007b), which uses two separate representations and two separate mining techniques, but limits itself to detecting malformed conditions. Using richer representations that capture both temporal order properties, data dependencies, and multiple kinds of program elements, a tool can detect both missing and misplaced calls, missing or malformed conditions, and missing or malformed exception handling. The representation of choice for many of the tools described in this section is a graph, which is often mined with frequent subgraph mining.

Detecting Violations Using Graphs

GrouMiner (T. T. Nguyen et al., 2009) represents method bodies of a code base as a ***graph-based object usage model*** (groum). A groum is a directed acyclic graph with labelled nodes and unlabelled edges, generated for a single method in the program. Nodes represent actions, such as method calls or constructor calls, and control structures, such as if or while. Node labels then convey information about the node itself: An action node carries the name of the called method as its label, while control nodes are labelled according to the control structure they represent. Edges represent a temporal usage order and data dependency between two nodes: Two nodes A and B are connected by a directed edge $A \to B$ if A is used before B, and there exists a data dependency between A and B.

GrouMiner uses frequent subgraph mining using the a-priori mining algorithm to detect patterns in the groums of method bodies. In order to check for graph isomorphism in the algorithm, it uses Exas feature vectors (H. A. Nguyen, Nguyen, Pham, Al-Kofahi & Nguyen, 2009) as an approximate metric of graph similarity. If

two graphs have the same Exas feature vector, they are very likely to be isomorphic. An Exas feature vector can be thought of as a signature of a graph: It generates hash values for each node and each path up to a certain length in the graph, and counts the occurrences of these hash values in the graph. Since hash values can collide, so can feature vectors, which is why it is an approximation.

The tool detects violations of patterns as infrequent inextensible subgraphs of a pattern graph. A graph G is said contain a usage anomaly of pattern P if it contains an occurrence G' which is a subgraph of P, but cannot further be extended to be isomorphic to P, and G' rarely occurs as a pattern on its own. It can detect such violations efficiently because infrequent inextensible subgraphs are encountered during frequent subgraph mining and retained for the purpose of violation detection.

MUDetect (Amann et al., 2019) builds further on GrouMiner by enriching the representation to include data values, data flow relations, synchronisation information, and exception-handling information, leveraging insights gained from a study on the effectiveness of existing misuse detectors (Amann et al., 2017), on which we go into more detail in Section 2.3. Their resulting representation, called an ***API Usage Graph*** (AUG), is a labelled directed acyclic multi-graph, where both nodes and edges are labelled, generated intra-procedurally for each method body in a project. Nodes can either be action nodes, which may be method calls, construction invocations, variable assignments, etc.; or data nodes, representing data entities such as variables, objects and constants. Edges can be of multiple classes, representing either control flow or data flow.

AUGs are mined for patterns using frequent subgraph mining, in a near-identical way to GrouMiner. MUDetect defines misuses of APIs as strict subgraphs of patterns, and detects such misuses by computing overlaps between a target graph T and a pattern graph P. Violation detection is done in an independent phase so that the tool can find all alternative overlaps between a pattern and a target, thereby being able to filter out false positives caused by alternative patterns. Additionally, considering detection as a separate phase allows MUDetect to mine patterns from multiple independent projects, thus increasing the accuracy and likelihood of correctness of patterns. Detected violations are further filtered according to heuristics to remove common false positives, and are finally ranked according to one of multiple possible ranking strategies. We go into more detail on MUDetect, its AUG representation, and algorithms in Chapter 3.

BigGroum (Mover, Sankaranarayanan, Olsen & Chang, 2018), another extension to GrouMiner, mines for patterns in Android API usages employing a different approach. Their central challenge is to scale groum mining horizontally, mining for patterns in a large corpus of projects, relating to the large API surface that is the Android SDK. They extend the groum representation to include data nodes and data flow definition and use edges. Before mining for subgraphs, they cluster the groums based on the API methods it calls, which reduces the search space for frequent subgraph mining. They further define the relation of ***graph embedding***, denoted $A \preceq B$, a special case of subgraph isomorphism that allows for flexibility with transitive edges. Two graphs A and B are isomorphic if $A \preceq B$ and $B \preceq A$. Using these definitions, they reduce the graph embedding problem to a SAT problem by encoding it into a propositional formula φ, which is satisfiable if and only if $A \preceq B$. This propositional formula can then be fed into a SAT solver.

They then cluster groums into bins of isomorphic groums, which are used to construct a lattice based on the embedding relation. Using this lattice and the frequency of the bins, they categorise each bin as popular, isolated or anomalous. A bin is popular

if its frequency is sufficiently high, and is considered a pattern. A bin is anomalous if its grooms are strictly embedded in a pattern, but the bin is infrequent. Finally, a bin is isolated if it is neither popular nor anomalous, meaning that it is not embedded in another bin and the groum occurs infrequently. Violations may then be found as anomalous bins.

JSMiner (H. V. Nguyen et al., 2014) is a tool that mines inter-procedural patterns in JavaScript code using a graph-based representation called JSModel, based on groums. This tool is similar to ours in the sense that it is able to mine inter-procedural patterns from graph-based representations. Inter-procedural analysis in a JSModel stems from its ability to represent a related function's body in the graph. For example, when a function call is modelled in their graph representation, the corresponding function's body is included in the graph and linked to the call through inclusion edges. This is a solution to one of many challenges in JavaScript pattern mining: Event handlers and callbacks. A lot of JavaScript code bases make extensive use of both anonymous and named functions to pass as data to other functions, which later invoke the function. In order to mine meaningful patterns in the presence of callback-passing style, the model needs to represent the body of the callback function as well. When mining for usage patterns, JSMiner additionally prioritises inclusion edges in an attempt to mine inter-procedural patterns.

The main difference between JSMiner and our approach is that our approach is boundary-agnostic. We do not separate function bodies of callees or use specific edges to represent such information, allowing our tool to mine usage patterns regardless of how its implementation is scattered across method bodies. Although JSMiner may be able to be mine boundary-agnostic patterns, we cannot confirm this. Furthermore, the JSModel representation is closely related to groums, while our representation of method bodies and patterns builds upon AUGs. Even though AUGs are essentially based on groums, they contain more diverse information by using more concrete node and edge representations that can aid in differentiating correct usages from misuses. Finally, JSMiner's evaluation as a misuse detector is very sparse, and leaves a lot to the imagination.

Detecting Misuses Using Probabilistic Models

GraLan (A. T. Nguyen & Nguyen, 2015) is a graph-based statistical language model used for code completion and API recommendation, based on groums. Contrary to DroidAssist (T. T. Nguyen et al., 2015), this model includes all of the information present in a groum, i.e. it also encodes control structures. A trained GraLan model could therefore be used to detect violations, by querying it to retrieve the probability that a certain groum occurs. If this probability is sufficiently low, a violation could be flagged.

PHOG (Bielik et al., 2016) is another probabilistic language model for code. It generalises probabilistic context-free grammars to probabilistic higher order grammars. If a language can be parsed using a context-free grammar, one can generate a probabilistic context-free grammar by counting the frequency with which each production rule is used to parse a training set of programs. However, such probabilistic context-free grammars are shown to be ineffective at making meaningful suggestions. A probabilistic higher order grammar instead takes the context of the expected code suggestion location into account, and makes recommendations based on the frequency of production rules used in a similar context. As with the previously-covered probabilistic models, we may be able to exploit such a model to report violations if the likelihood of a certain

statement given its context is sufficiently low.

Detecting Misuses Using Mutation Analysis

MutApi (Wen et al., 2019) is an API usage violation detector which detects violations using mutation analysis rather than pattern mining. Their motivation is two key insights: A misuse is often a mutant of a correct usage, and deviation from a pattern is too simple of an assumption to be considered a real misuse. Rather than identifying patterns in correct usage of an API, MutApi attempts to identify patterns in misuses of an API. To identify such misuse patterns, it applies mutation analysis by mutating known correct usages with a number of possible mutation operators. Such operators include removing exception-handling code, removing conditions, flipping relational operators, etc. Determining whether a mutation leads to an actual misuse is done by running the considered project's test suite on the mutant and analysing resulting stack traces of failed tests to determine whether the test failure is due to a misuse in the API. If it is, the misuse is modelled in a domain-specific grammar called ***structured call sequences***, which are API call sequences with certain syntactic operators, such as try-catch and if-then-else statements. Finding misuses then corresponds to matching these misuse patterns. Although MutApi achieves high precision and recall, it struggles to find misuses when the test suite is of low quality and cannot kill mutants, and cannot find violations altogether if a project has no test suite. Note also that, contrary to all previously covered tools, MutApi does not rely solely on static analysis, and instead requires dynamic analysis to mine its misuse patterns, in the form of running a test suite and observing the results. However, their detection is fully static still, since it involves matching the misuse patterns to the source code of target projects.

2.3 Evaluating API Misuse Detectors

Evaluating misuse detectors to get an idea of their performance when it comes to detecting actual misuses in real programs relies on such information being readily available. In order to help researchers evaluate their tools, the misuse detection benchmarking suite MUBench (Amann et al., 2016) has been created. Its dataset contains 89 identified misuses across 33 projects and a developer survey, of which over two thirds may lead to crashes in the program. To test tools on this benchmark, it provides a pipeline which automatically fetches the project's source code from version control systems, compiles the project and invokes the detector on this project. The pipeline offers three distinct experiments to measure a detector's precision, its recall, and its recall upper bound, i.e. its recall under perfect training and detection conditions.

The experiments measuring precision and recall run the detectors in full on each of the projects. Therefore, the detector can train its inner mechanism, regardless of actual representation, from the project's source code, and then detect misuses in the same project. Reported violations can then be cross-checked by reviewers against known misuses identified in the dataset, which facilitates the computation of precision and recall. By running different detectors on the same dataset, comparing detectors is made possible, not only in statistics such as running time, precision, and recall, but also regarding which misuses are missed by one detector, but reported by another.

In the experiment measuring recall upper bound, for each misuse identified in the dataset, a corresponding correct usage is crafted. A detector is provided this single

		Recall Upper Bound		
Detector	Precision	Conceptual	Empirical	Recall
DMMC	9.9%	28.1%	23.4%	20.8%
JADET	10.3%	29.7%	23.4%	5.7%
Tikanga	11.4%	29.7%	20.3%	13.2%
GrouMiner	0.0%	75.0%	48.4%	0.0%

Table 2.1 – The results obtained by Amann et al. (2017) of precision, recall, and conceptual and empirical recall upper bound of four existing detectors.

correct usage to train with, and then provided the source code of the misuse. This simulates perfect training conditions, where every misuse can have a correct usage provided for it, and hence the detector can find the misuse. This experiment allows for evaluating the detector's internal representation of patterns and targets, as well as its mining and detection algorithms.

Using the benchmark dataset in MUBench, the authors later classified each of their identified misuses into MuC (Amann et al., 2017), the API-Misuse Classification, a taxonomy of misuses. MuC follows an orthogonal design, classifying misuses both in terms of their involved element (method call, condition, iteration, or exception handling) and the type of misuse (missing or redundant). Conditions are further divided according to the type of condition, such as null checks, or synchronisation.

The classification is then used to provide a conceptual classification of twelve existing detectors, each of which are covered in the previous section. This classification is based on the conceptual capabilities of the detector, rather than real-world performance. For example, they found that none of the detectors are able to handle missing synchronisation conditions, since none of the detectors capture such information.

The authors then select four misuse detectors to run empirically on the dataset: DMMC (Monperrus & Mezini, 2013), JADET (Wasylkowski et al., 2007), Tikanga (Wasylkowski & Zeller, 2011), and GrouMiner (T. T. Nguyen et al., 2009). The eight other detectors are not selected, because they either cannot handle general Java code, or rely on defunct services such as Google Code Search. We replicate their results in Table 2.1. Their empirical evaluation shows that all four of these detectors have extremely low precision (less than 12% in all cases) when applied to real-world projects, achieve fairly low recall, and that their conceptual capabilities exceed their empirical recall upper bound. Although conceptually able to detect misuses of a certain classification, in actuality the detectors often fail to do so. Most surprising is GrouMiner, which has a high conceptual recall upper bound and clearly outranks the other detectors in its empirical recall upper bound, but fails to correctly detect any misuse.

Given the outcome of the various experiments for the detectors, they then investigate the root causes for low precision and recall, and make a number of observations. They identify six root causes for false positives: (*a*) Uncommon usages that are correct, but too infrequent to mine as patterns; (*b*) Imprecise static analysis, including a lack of inter-procedural analysis, leading to imprecise patterns or imprecise detection; (*c*) Alternative usages, causing detectors to report a violation that is an instance of an alternative pattern; (*d*) Method bodies using their own API to implement functionality;

(*e*) Implicit dependencies between object; and (*f*) failure to distinguish call multiplicities.

As an example of the fourth cause, partial usages, a Java `Collection` may use its `add` method to implement `addAll`, and does not have to satisfy the contracts a client needs to oblige. This may lead to a false positive if the detector does not distinguish self-usage from client usage. The fifth cause, implicit semantic dependencies, is prevalent when a client iterates over two collections of the same size. A size check is only necessary on the first collection, but detectors may flag violations due to the missing size check on the second collection, since it does not capture implicit semantic dependencies between the two collections. Finally, as an example of the last cause, consider Java's `StringBuilder`, which has an `append` method that may be called arbitrary many times. Some detectors may mine patterns where this method is called multiple times, and report violations when it is called only once, which is a false positive.

The researches also determine three root causes for false negatives, as identified in the recall upper bound experiment: (*a*) Poor representations failing to capture sufficient information to distinguish misuses from correct usages; (*b*) poor matching, by not considering semantics of the code or the representation; and (*c*) imprecise static analysis leading to missing information that make detectors unable to match a misuse to a correct usage. As an example of the first cause, a representation may fail to distinguish between overloaded methods, or fail to capture concrete constant and literal values in call arguments, leading it to believe its representation of the misuse is identical to the pattern. Not considering semantics of code and representation, as the second cause of false negatives, includes failing to match subtype relations in method call receivers, or failing to match a pattern to a target due to call order violations. A concrete example of the final cause is GrouMiner, which fails to identify the static receiver type of the result of a chained method call, and therefore fails to relate a pattern to a target.

Additionally, the authors observe that detectors lack strong ranking strategies and therefore fail to push true positives to the top of their results. Finally, a lack of correct usage examples often prevents patterns from being mined and therefore violations from being detected. Integrating cross-project mining may help alleviate this problem.

Their findings are vital to improve the state-of-the-art of misuse detection, and many of their insights have been applied in the creation of MUDetect (Amann et al., 2019), their subsequent work. However, we note that a prevalent source of false positives and false negatives relate to intra-procedural analysis. For example, detectors can easily get fooled if a call they believe is missing, is actually present in the body of a first-party callee, or if a call that is missing in the body of a method is in reality always called by the method's callers. Secondly, the issue related to self-usages is essentially an inter-procedural issue as well. Consider again the example of the `addAll` method of a Java collection. Although this method's implementation can use the `add` method and not oblige by its contracts, the callers of `addAll` still need to oblige those rules. Thus, such self-usages would be less problematic if the tool viewed the program under analysis from an inter-procedural angle. This supports our motivation of re-integrating inter-procedural analysis into the state-of-the-art of misuse detection.

To summarise, the previous works on static API misuse detection suffer heavily from a number of shortcomings, leading to low precision and recall. We are primarily interested in the lack of inter-procedural analysis, a prevalent root cause leading to false positives, and which, if added, may also decrease the number of false negatives, thus increasing a detector's precision and recall. Thus, in the next section, we explore existing techniques of inter-procedural analysis in pattern mining and misuse detection, and identify potential strategies to incorporate such analyses into a state-of-the-art misuse detector.

2.4 Inter-Procedural Analyses in Misuse Detection

Since all of the detectors evaluated in the previous section suffer from false positives and false negatives due to the absence of inter-procedural analysis, we need to identify ways to integrate such inter-procedural analyses into the pattern mining and violation detection phases. Since API usage can be spread across multiple methods, mining for patterns inter-procedurally allows us to find more meaningful patterns, by getting a better view of the API usages as a whole. However, patterns mined from such larger views will lead to more false positives, since when detecting violations of the patterns, partial pattern instantiations may reside in transitively called methods. Hence, both pattern mining as well as violation detection need inter-procedural analysis. In this section, we review the inter-procedural capabilities of existing pattern miners and misuse detectors, and highlight a number of key inspirations to our work.

MAPO (Xie & Pei, 2006; Zhong et al., 2009) mines for patterns relating to third-party methods inter-procedurally through the process of inlining call sequences of callees into the sequence of their caller. The first version of MAPO (Xie & Pei, 2006) recursively inlines first-party method calls up to a call depth of three. This may lead to exceptionally large method call sequences, issues with call multiplicities, and may bias the frequent subsequence mining, since the call sequence of a transitively called method can be inlined multiple times if it is called multiple times, either directly or indirectly. The second version of the miner (Zhong et al., 2009) alleviates these issues by employing a more intelligent strategy. A call sequence found in a callee method's body is only inlined when the callee has not been inlined before in the same sequence. This eliminates both direct and indirect recursion, as well as duplication when a method is called multiple times. It also allows MAPO to perform deeper inlinings, unbounded by call depth, until leaf nodes in the call graph are reached.

Empirical evaluation showed that the inlining technique employed in MAPO is effective at finding API usage patterns scattered across multiple implementation methods. However, it is unclear whether pattern mining using inlining is effective at finding API misuses, since MAPO is a pattern miner and recommender, rather than a misuse detector. Our work is heavily inspired by MAPO's call sequence inlining: We use their notion of inlining in a graph-based model, and two of our inlining strategies are based on the end condition of their two inlining algorithms.

Recall that Chronicler (Ramanathan et al., 2007a) mines for inter-procedural patterns by extending their precedence constraints with a function summary of the callee at every call site. In a sense, precede constraint information of the callee is inlined into the precedence constraint chain of the caller. Chronicler's use of function summaries as a memoisation technique forms the inspiration to our two-phase graph construction, where we first create graphs for each individual method body intra-procedurally, and then combine the graphs to form inter-procedural models. This prevents us from having to re-parse and regenerate graphs for every callee, effectively memoising the intra-procedural models.

JSMiner (H. V. Nguyen et al., 2014) embeds the bodies of callees into the graph of their callers, which is very similar to our approach. However, JSMiner explicitly keeps function boundaries in their representation, through an indirection of a function call to a function declaration to the body of the function. We note that this does not allow pattern implementations to be scattered arbitrarily across multiple methods. Thus, we instead decide to erase method boundaries in our inter-procedural graph representation, as if the inlined callee's body was written directly in the caller. Additionally, JSMiner's JSModel representation is heavily based on groums, which have been shown to

perform poorly in the previous section. Thus, we employ an alternative, richer graph representation, which impacts our inlining algorithm and adds additional complexity due to the increase in semantics.

As explained in Section 2.2.1, Colibri/ML (Lindig, 2016) applies inlining on its input relation, allowing it to mine both first-party API and third-party API call patterns. In our approach, when inlining, we specifically decide to remove method boundaries, for reasons described above. This negatively impacts our ability to detect first-party API usage patterns and thus, detect misuses of first-party APIs. Retaining the ability to do this would instead require introducing method boundaries into our representation, and would lead to less general patterns of third-party APIs, as the representation dictates implementation details.

PR-Miner (Li & Zhou, 2005) employs an inter-procedural check to prune false positives from its detection results. Since pattern instances and violations may cross method boundaries, PR-Miner checks a function's callees to determine whether a partial pattern is completed by any of the callees. The tool also checks whether all of the callers of a function complete a partial pattern, in which case the violation in the function is a false positive. This approach partially inspires our inter-procedural violation filtering algorithm. We do not explicitly check if a missing element is present in a callee method, since such information is readily available in the inlined representation. However, our filtering algorithm does check the callers of a method, to prevent false positives if a pattern is extended to completion by all of the callers. Additionally, because of the inter-procedural representation, it may happen that both a caller and a callee contain the same violation, which would lead to duplicate results. We apply heuristics to determine who is to blame for the violation, and use this information to prevent reporting duplicate violations.

2.5 Conclusion

Due to the ever-growing number of popular, ever-changing APIs, employing predetermined rules to detect violations of API contracts has become infeasible. Therefore, recent work in the field of API misuse detection statically infers these rules by mining usage patterns from corpora of projects. They identify misuses of the API by detecting usages that deviate from the mined patterns, under the assumption that a usage must be correct if it occurs frequently enough, and that usages that are similar, but not identical, to the mined usage patterns constitute a bug.

To mine such usage patterns, existing approaches often use either frequent itemset mining, frequent subsequence mining, frequent subgraph mining, or a combination thereof. In general, frequent itemset mining cannot natively capture an order relation, whereas frequent subsequence mining captures total orders, and frequent subgraph mining captures partial orders. These mining techniques are paired with varying representations, depending on the phenomena a tool aims to capture. This combination of representation and mining technique dictates which types of misuses a detector can identify, and thus heavily impact a tool's capabilities. We review a total of 23 existing approaches to misuse detection, some of which are not specifically designed to detect violations of API usages, but whose representation can potentially be used to do so. We summarise these existing tools in Table 2.2.

Evaluation of the four state-of-the-art misuse detectors GrouMiner, DMMC, Tikanga and JADET showed that precision and recall is quite low. Root causes for

this phenomenon include uncommon but correct usages that are misclassified as a misuse due to divergence from the mined patterns, poor representations, and lack of inter-procedural analysis. We determine that a graph-based representation, such as the groums used in GrouMiner, is most promising to detect misuses. Based on the insights gained from the evaluation of these four detectors, and the groum representation, Amann et al. (2019) construct the API usage graph representation, which alleviates the issue of poor representation, but does not tackle the lack of inter-procedural analysis.

We thus analyse previous research to identify potential strategies of incorporating inter-procedural analysis into a graph-based representation. We find that an often used approach is inlining the representation of transitively called functions into the representation of the caller. A tool closely related to our work, JSMiner (H. V. Nguyen et al., 2014), embeds the graph of callee functions into the graph of the caller. However, their approach deliberately retains function boundaries, which limits the ability of the miner to detect API usage patterns whose instances scatter their implementation arbitrarily across different functions. Additionally, JSMiner focuses on pattern mining rather than misuse detection, and thus does not implement inter-procedural violation filtering, which is often found in inter-procedural misuse detectors. Such filters remove false positives that are caused by missing pattern elements which are present inter-procedurally in the function body of any callee, or all callers.

Thus, we conclude that to achieve our research goal, we should extend graph-based misuse detection with inter-procedural analysis. We identify that a viable approach is to inline the graph of a callee into the graph of its callers in a method boundary-agnostic manner, and to design an inter-procedural violation filter. The former allows us to achieve our goal of mining patterns in the presence of helper methods, whereas the latter allows us to remove false positives that appear because of missing program elements when viewing a single method in isolation. In terms of which graph representation to incorporate this approach into, we look in the direction of MUDetect, the current state-of-the-art misuse detector, whose representation is specifically designed to distinguish correct usages from misuses. This tool is the subject of our next chapter, where we continue this literature review.

Tool	Representation	Technique	Can detect misuses relating to... calls?	call order?	conditions?	exceptions?	superfluous calls?	Inter-procedural analysis
PR-Miner	Callee association rules	Association rule mining	✓	✗	✗	✗	✗	False positive filtering
Colibri/ML	Caller-callee relation	Formal concept analysis	✓	✗	✗	✗	✗	Inlining relation
DMMC	Type usages	Majority rule	✓	✗	✗	✗	✗	n/a
CodeWeb*	Reuse association rules	Association rule mining	✓	✗	✗	✗	✗	n/a
PARSEWeb*	Directed acyclic graph of call order	Clustering	✓	✗	✗	✗	✗	n/a
JADET	Temporal order relations	Formal concept analysis	✓	✓	✗	✗	✗	n/a
Tikanga	Computation tree logic formulae	Formal concept analysis	✓	✓	✗	✗	✗	n/a
Chronicler	Call precedence constraints	Subsequence mining	✓	✓	✗	✗	✗	Function summaries
MAPO*	Call sequences	Subsequence mining	✓	✓	✗	✗	✗	Inlining
DroidAssist*	Hidden Markov Model	Probabilistic	✓	✓	✗	✗	✓	n/a
Alattin	Alternative association rules	Association rule mining	✗	✗	✓	✗	✗	n/a
Ramanathan et al. (2007b)	Call precedence and data constraints	Itemset mining, subsequence mining	✗	✗	✓	✗	✗	Function summaries
Chang et al. (2007)	Program dependence graph	Itemset mining, subgraph mining	✗	✗	✓	✗	✗	n/a
Car-Miner	Sequence association rules	Subsequence mining	✗	✗	✗	✓	✗	n/a
Acharya and Xie (2009)	Call sequences	Subsequence mining	✗	✗	✗	✓	✗	Inter-procedural traces
Weimer and Necula (2005)	Finite state automata	Specification mining	✗	✗	✗	✓	✗	n/a
GrouMiner	Groum	Subgraph mining	✓	✓	✓	✓	✗	n/a
MUDetect	AUG	Subgraph mining	✓	✓	✓	✓	✗	n/a
BigGroum	Groum	Subgraph mining	✓	✓	✓	✓	✗	n/a
JSMiner	JSModel	Subgraph mining	✓	✓	✓	✓	✗	Function body embedding
GraLan*	Statistical graph	Probabilistic	✓	✓	✓	✓	✓	n/a
PHOG*	Probabilistic higher-order grammar	Probabilistic	✓	✓	✓	✓	✓	n/a
MutApi	Misuse patterns	Mutation analysis	✓	✓	✓	✓	✓	n/a

Table 2.2 – Overview of existing tools that can be used to detect API misuses. Tools marked by * are not designed to perform violation detection, but its representation and mining technique could potentially be reused to identify API misuses. For space considerations, we refer to the main text for citations.

3 API Usage Graph Mining

The MUBench and MuC studies described in the previous chapter provide important insights into the shortcomings of previous detectors. Using these insights, the state-of-the-art tool MUDetect (Amann et al., 2019) builds upon GrouMiner (T. T. Nguyen et al., 2009), extending its graph-based object usage model (groum) representation, graph mining and detection algorithms, to improve upon its performance.

In this chapter, we take a closer look at MUDetect's graph representation and algorithms. In Section 3.1, we explore the API usage graph representation, its structure, node types, and edge types. Afterwards, we delve into the construction process of AUGs in Section 3.2. Then, in Section 3.3, we describe the tool's mining and detection algorithm. We further review the evaluation of the detector in Section 3.4, which shows that MUDetect outperforms four other state-of-the-art misuse detectors, but also identifies a number of important shortcomings, including a lack of inter-procedural analysis. Thus, we select MUDetect as a base implementation to incorporate graph inlining into. Finally, we conclude this chapter in Section 3.5.

3.1 Structure of an AUG

We start off by introducing the API usage graph (AUG) structure. Figure 3.1 shows an example Java source code snippet, along with the AUG corresponding to the code. We use this example to describe the API usage graph representation. As displayed in the figure, an API usage graph is a labelled, directed, acyclic multi-graph, where both nodes and edges are labelled. Since an AUG is a multi-graph, there may be multiple edges with different labels connecting the same two nodes. We can see this in the top of the graph, where there exist two edges connecting the node labelled `<nullcheck>` to the node labelled `FileInputStream.<init>`.

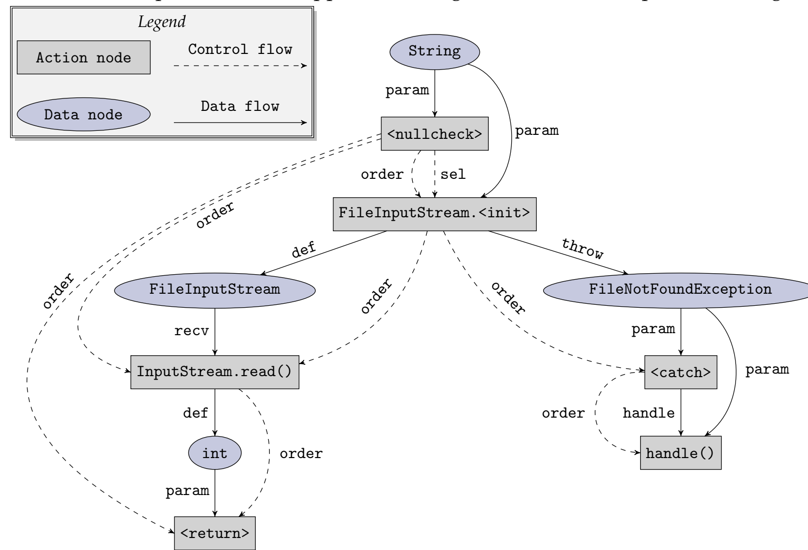
There exist many different node and edge types. Nodes represent program actions, such as method calls, variable assignments, or constructor invocations, and program data, such as variables and constants. Action nodes are represented as gray boxes in the figure, whereas data nodes are depicted as blue ellipses. Action nodes are labelled according to the action they represent, e.g. for a method call, the label is the name of the target method, as is depicted in the `handle()` and `InputStream.read()` nodes in the figure. Data nodes are labelled according to the type of data they represent, such as `String` and `FileInputStream` in the example.

Edges represent different types of control flow or data flow, depicted using dashed and solid edges in the figure, respectively. Edges are labelled according to their concrete type, e.g. “param” for a parameter data flow edge that connects a data node to its use

site, or "order" for a temporal control flow order edge between two action nodes. Such edges can readily be seen in the figure. Note that order edges can be transitive, but these are omitted from the graph for readability purposes.

```java
if (file != null) {
    try {
        FileInputStream fis = new FileInputStream(file);
        return fis.read();
    } catch (FileNotFoundException e) {
        handle(e);
    }
}
```

(a) An example Java code snippet containing branches and exception handling.



(b) Example AUG corresponding to the example code snippet.

Figure 3.1 – An example Java code snippet and its corresponding AUG, adapted from the example given by Amann et al. (2019). Action nodes are shown as gray squares, data nodes as blue ellipses. Data flow edges are shown as solid edges, while control flow edges are dashed edges. Note that transitive order edges are omitted for readability purposes.

There exist eighteen types of action nodes that can be present in an AUG, all relating to possible program statements or expressions and control structures. Because of the sheer number of possible action nodes, we do not go into great detail for all of them. An interesting action node type is the operator node, which represents binary operators present in the method body. The actual operator is abstracted by MUDetect, such that the tool focuses on the presence or absence of such operators, rather than the actual expression used, since conditions may be specified in many different ways. In general, an arithmetic operator is given the label "`<a>`", a relational operator is given the label "`<r>`", and logical operators are removed. An exception to this is a comparison to the `null` literal, which is represented using the "`<nullcheck>`" label, which can be seen in the figure.

Another important feature is that the AUG representation abstracts over the receiver type of a call. As can be seen in the figure, the action node corresponding to the call to the `read` method on the input stream is labelled as `InputStream.read()`, even though we statically know that the receiver object is a `FileInputStream`. However, the receiver type is abstracted to generalise the representation in the presence of method overloading in type hierarchies.

Action nodes only represent actions, the data which is involved in the actions is

represented separately as data nodes. Six types of such data nodes exist, representing for instance variables, literals, or constants. A special case of data nodes is the anonymous instance method node, which is used when a method body constructs an instance of an anonymous class to use as a function object. The methods defined in this instance are represented individually, so that pattern mining can also capture patterns in such function objects. In a way, this is similar to the approach taken by JSMiner (H. V. Nguyen et al., 2014).

To connect all of these nodes, five types of control flow, and six types of data flow edges exist. Note that since an AUG is a multi-graph, multiple edges of different types may exist between the same two nodes. Control flow edges always involve at least one action node, whereas data flow edges always involve at least one data node.

The five type of control flow edges are as follows: (*a*) Order edges, representing temporal call order between nodes, (*b*) selection edges and (*c*) repetition edges, which both connect an action node representing a condition to action nodes in branches controlled by the condition, (*d*) synchronisation edges, which connect a data node to the actions executed under a lock held on the data, and (*e*) finalisation edges, which connect action nodes that may throw an exception in a try-catch-finally construct to the action nodes executed as finalisation code of this construct.

Since the temporal call order relation is transitive, order edges can also be transitive, representing the transitive closure over this relation. Backward edges in call order, which occur in loops, are excluded from the representation to keep AUGs acyclic. Additionally, a data-dependent closure of condition edges is generated: Using data flow information, additional condition edges are instantiated which link action nodes affecting data involved in the condition, to the action nodes of the condition as well as action nodes controlled by the condition. If the condition is involved in an if-then-else construct, the edges are selection edges; if it is involved in a loop condition, the edges are repetition edges.

In terms of data flow edges, there exist definition edges, which link an action node that creates a data value to the data node that is created, as well as two use edges that connect a data node to their use site, namely receiver edges and parameter edges. Throw edges connect an action node that may throw an exception to the exception data node that may be thrown. Exception handling edges connect a catch action node to the action nodes that handle the caught exception. Although exception handling nodes convey control flow information, they are considered data flow edges, since type information of the exception flows into the handling code. A final data flow edge type is the qualifier edge, which connects type annotations such as `@NonNull` to the data node on which the qualifier is applied.

One final edge, which is neither a control flow edge, nor a data flow edge, is the containment edge. Such edges are used to connect an anonymous class instance to the method data nodes defined in the instance, as well as to connect such method data nodes to the action nodes contained within the method body.

3.2 Constructing AUGs

Since AUGs are groums with a richer representation, it comes as no surprise that at a high level, the construction of an AUG is similar to the construction of a groum. AUGs are constructed in three steps. First, source code of the target project is parsed to

```
function CREATE-AUG(astNode)                                                    1
returns "a partial AUG for the given AST node"                                  2
    switch TYPE-OF(astNode) do:                                                 3
        case IfStatement ⇒ CREATE-IF-STATEMENT-AUG(astNode)                     4
        # ...                                                                   5
                                                                                6
function CREATE-IF-STATEMENT-AUG(ifStmt)                                        7
    conditionAUG ← CREATE-AUG(CONDITION-NODE(ifStmt))                           8
    trueAUG ← CREATE-AUG(TRUE-BRANCH(ifStmt))                                   9
    falseAUG ← CREATE-AUG(FALSE-BRANCH(ifStmt))                                 10
                                                                                11
    branchesAUG ← PARALLEL-MERGE(trueAUG, falseAUG)                             12
    return SEQUENTIAL-MERGE(conditionAUG, branchesAUG, SelectionEdge)           13
                                                                                14
function PARALLEL-MERGE(leftAUG, rightAUG)                                      15
    mergedNodes ← NODES(leftAUG) ∪ NODES(rightAUG)                              16
    mergedEdges ← EDGES(leftAUG) ∪ EDGES(rightAUG)                              17
    return new AUG(mergedNodes, mergedEdges)                                    18
                                                                                19
function SEQUENTIAL-MERGE(leftAUG, rightAUG, edgeType)                          20
    mergedNodes ← NODES(leftAUG) ∪ NODES(rightAUG)                              21
    mergedEdges ← EDGES(leftAUG) ∪ EDGES(rightAUG)                              22
    newEdges ← {}                                                               23
    for sinkNode ∈ SINK-NODES(leftAUG) do:                                      24
        for sourceNode ∈ SOURCE-NODES(rightAUG) do:                             25
            newEdges ← newEdges ∪ {new Edge(sinkNode, sourceNode, edgeType)}    26
                                                                                27
    return new AUG(mergedNodes, mergedEdges ∪ newEdges)                         28
```

Listing 3.1 – Pseudo-code describing the AUG construction phase.

an abstract syntax tree (AST) using the Eclipse Java Development Tools. Secondly, a temporary AUG is created from the AST. This temporary AUG captures action and data nodes together with partial usage orders, but is missing the transitive usage order and the data-dependent closure of condition edges. Hence, the final step is to compute these closures and instantiate them in the graph.

The second step, extracting temporary AUGs, is a bottom-up construction process of the graph through post-order depth-first traversal of the AST, following much of the construction process of groums (T. T. Nguyen et al., 2009). We describe this construction process in the pseudo-code shown in Listing 3.1. At each interesting AST node, a partial AUG is created that corresponds to the AST node. For non-leaf AST nodes, such as operators or composite control structures, the AUG is obtained by merging the partial AUGs of its descendants and a partial AUG created for the node itself. For example, the function CREATE-IF-STATEMENT-AUG in the pseudo-code shows this process for if-then-else statements. We create two partial AUGs for the branches and a partial AUG for the condition. We first merge the branches without connecting any of their nodes, since branches are isolated. Then, we merge the result with the partial AUG created for the condition, and connect the condition AUG to the branches via selection edges.

In GROUMINER, two merge operations are defined: Parallel merge ($\vee$) and sequential merge ($\Rightarrow$). Parallel merge of two partial graphs A and B leads to a new graph $A \vee B$ that contains all nodes and edges of A and B, but no edge between any node of A and any node of B. This is also called the disjoint union of graphs, and is shown in the PARALLEL-MERGE function in the pseudo-code. Sequential merge of two partial graphs A and B leads to a new graph $A \Rightarrow B$, also containing all nodes and edges of both A and B, but containing additional edges connecting every sink node of A

to every source node of B. This is shown in the Sequential-Merge function. In general, sequential merge is used when the latter graph has a dependency on the former, such as a temporal order dependency, while parallel merge is used when there are no dependencies.

For example, consider a method call `o.m(e1, e2)`, where e_1 and e_2 are program expressions, o is an instance of class C, and m is a method taking two formal parameters defined in class C. Construction of an AUG will first lead to two independent partial AUGs E_1 and E_2 for the respective expressions used as call arguments. A third AUG M is created for the method call, which contains a single node corresponding to the method call `C.m()`. Note that the node is labelled with receiver's static type instead of the actual instance, as a generalisation. To obtain the composite AUG for the method call, we first merge the partial AUGs for its arguments: $E_1 \vee E_2$. In this case, a parallel merge is used, since arguments are evaluated independently from one another. Since the call itself depends on the arguments, we perform a sequential merge connecting them to the partial AUG representing the method call, leading to the composite AUG $(E_1 \vee E_2) \Rightarrow M$.

Recall that AUGs capture condition edges separately. To do this, the sequential merge operation defined in GrouMiner is extended to instantiate such edges, depending on the actual control flow construct encountered. For an if-then-else statement, the sequential merge operation now adds additional selection edges going from the action nodes representing the condition to action nodes in a branch of the statement which share a data dependency with the condition.

Since AUGs also contain data nodes and data flow edges to represent the data dependencies, such edges need to be instantiated when constructing the graph. Thus, AUG construction adds data nodes for each data entity encountered in the program, initially represented as its own partial AUG. These data AUGs are then merged with other partial AUGs that depend on this data using the sequential-data merge operation. This operation is very similar to the previously-defined sequential merge operation, with the main difference being that it distinguishes between data use (parameter edges, receiver edges) and data definition (definition edges). In essence, when the left-hand side AUG's sink node is a data node, it adds a data use edge, and if the right-hand side AUG's source node is a data node, it adds a data definition edge.

3.3 Pattern Mining and Violation Detection in AUGs

As previously mentioned in Section 2.1.3, mining for patterns in a collection of graphs is performed using a-priori frequent subgraph mining. Although MUDetect's mining algorithm is mostly similar to the previously described a-priori algorithm, the authors tweak this algorithm to produce more meaningful patterns, by exploring the AUGs in a code semantics-aware fashion. Concretely, when extending candidate patterns, it distinguishes between node and edge types to guide the exploration, and only considers nodes as extension candidates if it is connected to the existing pattern by a non-order edge. Further filtering of candidate extensions is performed to only include nodes that affect a usage. For example, pure methods are ignored if their return value is not used, since they cannot otherwise influence the usage.

After candidate extensions are generated, they are clustered into bins of isomorphic extensions. This isomorphism check uses Exas feature vectors, described earlier, and hence this clusters extensions with identical feature vectors into the same bin. Any

inextensible patterns, i.e. candidate patterns that have no extensions but are sufficiently frequent, are added to the set of discovered patterns. For patterns that are extensible, the most frequent candidate extension is selected for further exploration. This leads to a greedy extension strategy that avoids the combinatorial explosion that would take place in an exhaustive search with backtracking.

Similarly to GrouMiner, MUDetect defines a violation as an infrequent strict subgraph of a previously mined pattern. To find such violations, MUDetect performs graph matching to compute overlaps between a target method AUG and a pattern. The graph matching follows the pattern growth approach in frequent subgraph mining, paired with a greedy exploration strategy. However, it does not feature an a-priori pruning phase, as there is no minimum support defined. When an overlap between a pattern and a target is imperfect, i.e. there exists a partial instance of the pattern graph in the target graph, this overlap is reported as a likely violation.

Concretely, the matching algorithm starts by detecting a method call node that is shared among the pattern and the target, which is a common subgraph consisting of one node. Then, it extends this common subgraph with more common nodes. Extending the subgraph is done by selecting the best pattern edge that is also present in the target, and adding this edge and the unexplored common node to the subgraph. Additionally, any other edge that connects this new node to other nodes that are already present in the overlap, are added to the subgraph as well. There may be multiple equivalent edges when extending patterns, which lead to more alternative mappings between pattern and target nodes, and therefore the possibility of selecting a mapping which is non-optimal. To decrease the likelihood of selecting such a non-optimal mapping, the algorithm selects edges with the lowest number of equivalents as the next extension edge.

When a common subgraph cannot be further extended, the subgraph is considered the final overlap between the pattern and target. If this overlap is not isomorphic to the pattern, it is a violation, otherwise, it signifies an instance of the pattern in the target. Afterwards, the algorithm checks if there are more method call nodes that are shared among the pattern and target and not included in any overlap yet, and restarts its exploration from such a node if one is found. This finds all alternative mappings of pattern nodes and edges to the target graph by attempting to maximise the coverage of target nodes.

After violations are detected, they are filtered according to a number of heuristics, as well as filtering out violations of one pattern if the violation is an instance of an alternative pattern. If a violation violates all alternative patterns, it would also be duplicated for each alternative pattern that is violated. In order to decrease the number of duplicates, violations that are subgraphs of other violations are filtered out as well.

Finally, the findings are ranked according to the tool's confidence of the finding being a violation. This confidence can be calculated in many different ways, and the authors find that a ranking function which involves both the pattern support, the number of equivalent violations, and a distance measure between the pattern and the violation, to be the optimal ranking strategy to push true positives to the top of the findings.

Detector	Precision	RUB	Recall
DMMC	7.5%	16.3%	10.7%
JADET	8.8%	16.9%	6.7%
Tikanga	8.2%	8.8%	7.6%
GrouMiner	2.6%	51.2%	3.1%
MUDetect	21.9%	72.5%	20.9%
MUDetectXP	33.0%	-	42.2%

Table 3.1 – The precision, recall, and recall upper bound of GrouMiner, DMMC, Tikanga, JADET, MUDetect, and MUDetectXP on the extended MUBench dataset, as obtained by Amann et al. (2019).

3.4 MUDetect Evaluation

The authors evaluate MUDetect on the MUBench dataset, and further extend this dataset with more misuses identified from a different study. They additionally evaluate the four detectors considered in Amann et al. (2017) on this extended dataset as a comparison. We display their obtained results in Table 3.1. This experimental evaluation shows that MUDetect successfully alleviates many of the problems described by Amann et al. (2017), and achieves higher precision and recall than the four state-of-the-art detectors compared to. However, precision and recall is still fairly low, 21.9% and 20.9% respectively. In a cross-project setting (MUDetectXP), where patterns for a specific API are mined from multiple projects and violations are later detected in a single target project, precision rises to 33%, while recall rises to 42.2%, improving upon other detectors' results more than fourfold. This shows that the AUG representation is successful at capturing more information necessary to detect violations, and that mining patterns from different projects tremendously increases the effectiveness of finding misuses. The full results of this experiment are available on MUDetect's artefact page (Amann, 2019).

Even though MUDetect leads to better results than other detectors, its statistics are still low. The authors investigated the false positives and false negatives, and thereby identified the main reasons for low precision and low recall. They find that many of the false positives (73.7%) are due to uncommon, but correct usages. Such usages occur too infrequently in the project to mine a respective alternative pattern, and the usage is flagged as a misuse as it violates another, more frequent pattern. Mining in a cross-project setting prevents many such situations from happening, but not all of them.

Lack of inter-procedural analysis directly leads to 15.8% of the false positives, where for example missing usage elements are reported, but are present in transitively called methods. Since MUDetect makes no attempt at inter-procedural analysis, it cannot eliminate such false positives, and thus, an inter-procedural violation filter would be beneficial. Other minor reasons for false positives include implicit object dependencies, which are very difficult to detect statically, the detection algorithm choosing a non-optimal mapping in its exploration, and edge cases that were missed by various filters.

The root causes of false negatives, where MUDetect fails to detect actual misuses,

are more divided. Most of the false negatives are due to AUG representation issues: Fifteen of the missed misuses involve illegal literal or constant values given as a parameter to a call. Since AUGs do not capture concrete values, such misuses cannot be detected. When constructing AUGs, self-usages, i.e. method calls on `this` or instance fields, are removed due to the potential of causing false positives because of partial usages. Eight misuses are missed because of this removal, a trade-off of recall for higher precision. The authors note that inter-procedural analysis may make removal of self-usages unnecessary, because it can then capture the full usage.

Another root cause of misuses are superfluous method calls rather than missing method calls, for example, a duplicated call to `next` on an iterator. None of the detectors evaluated can detect such misuses, since they search for missing elements. Matching issues also lead to some false negatives, where the pattern and target contain a single, distinct method call. Such AUGs cannot be matched by design, since they do not share a common method call. Heuristic removal of pure methods in the pattern leads to another six cases of false negatives, where a pattern cannot be matched to a target. Three more infrequent issues cause one false negative each: Heuristically removing subsequent calls to the same method leads to a pattern and target being identical, missed data flow relations because of the assumption of single static assignment causing a null-check to be missing from the pattern, and operator abstraction leading to the tool being unable to detect an inverted condition.

Despite these shortcomings, MUDetect achieves relatively high precision and recall when compared to the other state-of-the-art detectors. Therefore, we consider MUDetect to be a prime candidate for integration of inter-procedural analysis, since the evaluation shows that a common cause of false positives and false negatives can be related to the intra-procedural analysis employed.

3.5 Conclusion

The API usage graph representation used in MUDetect captures a lot of important details in method bodies that help it distinguish correct API usages from API misuses. Its nodes capture many different types of program actions, and additionally capture the data involved in these actions. To link all the nodes, an AUG contains many different types of control flow and data flow edges. In the end, the representation captures nearly all possible parts of a single method body.

Constructing such AUGs involves a depth-first post-order traversal of a method body's abstract syntax tree. For each node of the AST, construction creates a partial AUG, which is later merged in one of two ways with the partial AUGs of the AST node's ancestor and siblings. Two merge operators are defined: Parallel merge and sequential merge. Parallel merge computes the disjoint union of two graphs, i.e. a graph containing all nodes and edges of both partial graphs, but no edges connecting them. This is used to combine e.g. the partial AUGs for two different branches of a conditional statement. Sequential merge, on the other hand, also merges the two graphs into one larger graph, but connects the sink nodes of the first graph to the source nodes of the second. This merge operation is used to connect two partial AUGs representing program statements that are executed sequentially.

To detect misuses, MUDetect first mines its AUGs using a-priori frequent subgraph mining, which has been tweaked to consider code semantics and thereby generate more meaningful patterns. The patterns are then used to detect misuses, by computing

overlaps between a pattern and a target AUG. This detection algorithm follows much of the same structure as pattern-growth frequent subgraph mining, and continually extends a common subgraph to detect maximal overlaps between the two graphs. This leads to instances of the pattern if the overlap spans the full pattern, or violations if the overlap is a strict subgraph of the pattern. Violations are filtered according to heuristics and instances of alternative patterns are eliminated to reduce false positives, and the resulting misuses are ranked according to the tool's confidence in its findings.

The detailed representation of method bodies, the code semantics-aware extension strategy during mining, and the separate phase of overlap detection pays off: MUDetect significantly outperforms four other state-of-the-art detectors. When the tool mines patterns from multiple independent projects, these results are further improved, since it can find more proof of correct usages and alternative usages. However, the detector still regularly suffers from false positives and false negatives, in part caused by a lack of inter-procedural analysis.

Extending the AUG representation to represent multiple method bodies inter-procedurally, rather than a single method body in isolation, as well as introducing an inter-procedural filtering algorithm to MUDetect, could improve upon their already good results. Thus, we select MUDetect as the base implementation to extend with inter-procedural analysis, because of its detailed AUG representation and its relatively high precision and recall.

4 | Inter-Procedural Analysis by Inlining Graphs

As detailed in the previous chapters, we choose to extend MUDetect to incorporate inter-procedural analysis, thereby allowing the tool to eliminate inter-procedural false positives and detect violations in the presence of self-usages. To do this, we integrate graph inlining and inter-procedural filtering into the tool's workflow. For the former, we extend the AUG representation to ***extended API usage graphs*** (eAUG) and ***inlined API usage graphs*** (iAUG). For the latter, we extend MUDetect's filtering phase to additionally perform inter-procedural elimination. A full overview of the phases this extended tool goes through to inter-procedurally mine patterns and detect violations is displayed in Figure 4.1.

At a very high level, our tool performs the same four tasks as MUDetect, but in terms of iAUGs rather than AUGs. First, we construct iAUGs from source code. We then mine the iAUGs for patterns, and afterwards use the patterns to detect potential misuses in the iAUGs. Finally, we filter and rank the results. At a lower level, however, there are some key differences, which we describe below.

The construction phase can be decomposed into three smaller phases. Initially, we parse the source code of one or more Java projects into ASTs. From these ASTs, we

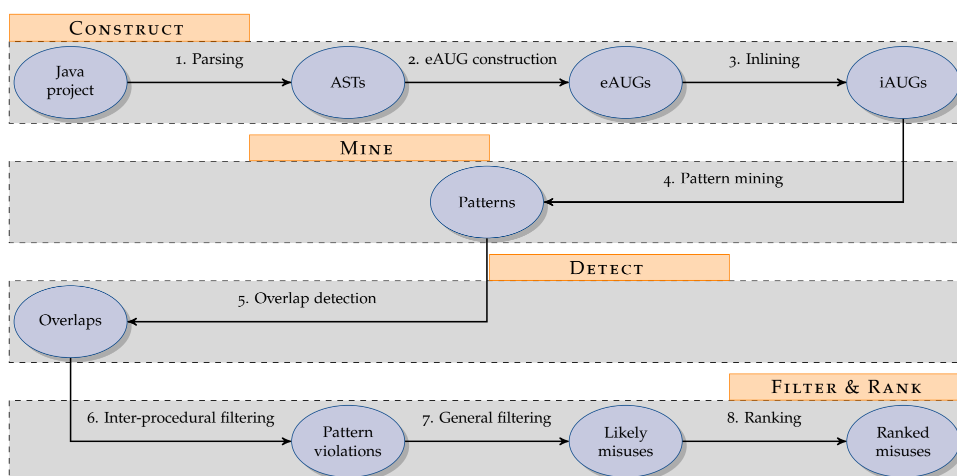


Figure 4.1 – A high-level overview of the phases our tool goes through to inter-procedurally mine and detect API misuses in a Java project. The different phases are shown as numbered edges. Blue ellipses display the data each phase produces. Aggregate phases are displayed in orange, bounded by gray boxes.

generate eAUGs as an intermediate representation for each method body. The structure of such an eAUG is similar to that of the AUG of MUDETECT, but extended with extra information to facilitate the inlining process. We describe these differences in Section 4.1. After eAUGs have been constructed, we apply our inlining algorithm on the eAUGs, the result of which is a collection of iAUGs, one for each eAUG. The inlining process is guided by one of six possible inlining strategies, dictating which calls are to be inlined, and in what order. We go more in depth on the general inlining algorithm, the different strategies, and the workflow of inlining a method call in Section 4.2.

Since iAUGs are extended and larger versions of normal AUGs, the mining and detection phases remain largely identical to MUDETECT's. The main difference in these phases is that we implement a number of performance and memory optimisations which are necessary to make mining and detection feasible. Since these two phases suffer from combinatorial explosion, and graphs are expected to become bigger after inlining, we optimise some of the data structures that are used in these phases to decrease the memory footprint of these algorithms. However, we do not go into detail on these changes, since they mostly consist of changing mutable data structures to immutable ones to promote structural sharing.

Note that in the filtering and ranking phase, we distinguish pattern violations from misuses. Indeed, not every pattern violation is an actual misuse, it may instead be an alternative usage, or a part of a larger, inter-procedural instance of the pattern. Thus, the filtering and ranking phase removes known false positives and ranks the resulting likely misuses, so that violations that are most likely to be a misuse are ranked higher.

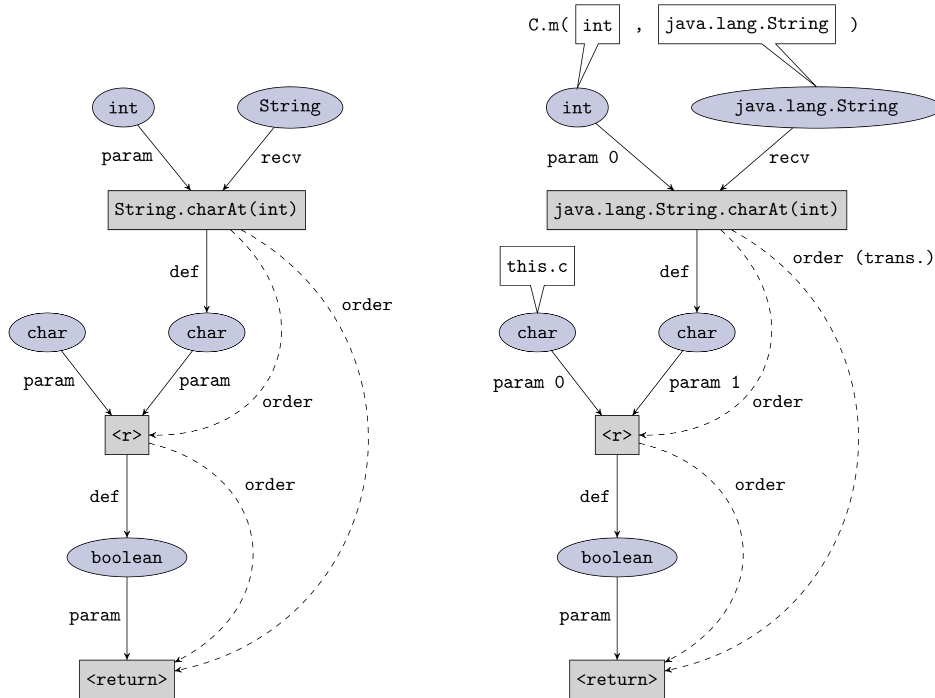
First, we apply heuristic filtering to remove false positives and duplicate results that are the result of the inter-procedural analysis. In a nutshell, we remove false positives that are the result of an iAUG containing a partial pattern which is further instantiated by its parent, and remove duplicate violations that may arise if an inlined callee violates a pattern. We go into more detail on this filtering in Section 4.3.

After inter-procedural filtering, the tool applies general filtering on the results to remove common false positives. For example, if a pattern dictates that the result of a call is to be stored into a variable for later use, whereas the instance does not store the temporary variable and instead chains the call, it would be incorrectly marked as a violation even though the pattern and instance are behaviourally equivalent. Thus, such potential violations are filtered out. This phase additionally filters out pattern violations that are an instance of another pattern, indicating that the violation is instead a valid alternative usage.

Finally, the findings are ranked according to the tool's confidence in it being an actual misuse. This confidence value is obtained from a combination of the pattern support, the number of equivalent violations found, and the distance between the pattern and the violation (the number of missing nodes and edges). Intuitively, if a pattern has a high support, deviations from it are more likely to be a misuse, whereas if a violation has a high number of equivalents, it is more likely to be an alternative usage than an actual misuse. Furthermore, if the violation deviates only slightly from the pattern, i.e. the distance between them is low, the violation is more likely to be an actual misuse than if it deviates significantly, in which case it may be unrelated to the pattern.

```
class C {
    char c = 't';
    boolean m(int i, String s) {
        String result = this.c == s.charAt(i);
        return result;
    }
}
```

(a) Contrived Java code example that highlights some of the key differences between AUGs and eAUGs.



(b) The AUG corresponding to the example method code.

(c) The eAUG corresponding to the example method code.

Figure 4.2 – A graphical overview of the key differences between AUGs and extended AUGs.

4.1 Extended AUG Representation

Inlining method bodies in the AUG representation requires program information which is not readily available in the original representation, such as parameter order, instance field mappings, and exceptions that may be thrown by a called method. To facilitate the inlining, instead of constructing normal AUGs, our tool constructs eAUGs, which are AUGs that contain all of the program information necessary to correctly inline method bodies. The structure of such eAUGs is similar to that of an original AUG, yet contains some key differences, which we illustrate in this section using a contrived Java code example shown in Figure 4.2a. Figure 4.2b shows the original AUG representation for this code, whereas Figure 4.2c shows the corresponding eAUG. We additionally explain the reasoning behind the implemented changes to the representation, and describe how the changes help alleviate some of the challenges in inlining method bodies.

The first difference is readily apparent from the comparison: Type names and method names in eAUGs are fully-qualified, as opposed to the simple names used in the original AUG representation. This is because different Java packages may

contain classes of the same name, and using fully-qualified names eases the distinction between such homonyms. Furthermore, using fully-qualified names allows us to easily determine whether a class or method belongs to a third-party API package.

A second difference can be seen in the parameter edges. In the original AUG representation, when a method call takes multiple parameters, all these parameters are independently connected to the call node via parameter edges. eAUGs instead encode the argument order into the labels of parameter edges to relate data values to their position in the call. In the example eAUG, we can see which data node is used in which position in the operator expression, i.e. the node labelled by "`<r>`", as well as the parameter position of the `int` parameter to the `charAt` call. Encoding the parameter order is necessary to properly link the caller's data nodes that are used as call arguments to the data nodes that represent the arguments in the callee.

However, properly linking these data nodes requires more than just encoding parameter order, since we need to know which data nodes to link to in the callee. Therefore, an eAUG keeps track of a mapping from formal parameters to their data nodes, which constitutes the third difference. This can again be seen in the example, where we annotated this mapping through callout boxes in the method's signature. Note that this information is internal to the eAUG, it is not considered during the mining of patterns to keep the mining method boundary-agnostic. We only display this information in the examples for illustrative purposes.

In a similar fashion, as a fourth difference, an eAUG keeps a mapping of the instance field names used in the method, to the data nodes representing these instance fields. This information is used during inlining to ensure that when a callee is inlined, and the callee is a method called on the same receiver as the caller, the fields used in the caller are shared with the fields used in the callee. This can be seen in the example, where a data node has been annotated with "`this.c`", indicating that this data node represents the `c` field. As with formal parameter mappings, this information is kept internal to an eAUG, and is not considered during mining for generality.

An additional, minor difference that can be spotted in the example is that eAUGs distinguish direct order edges from indirect ones, by explicitly marking transitive order edges as such. This makes it significantly easier to traverse the eAUG according to the execution order of its nodes, which is necessary to obtain a semi-consistent inlining order.[1]

A final difference, which is not shown in the example, is that an eAUG may contain edges which are never present in an original AUG. One such edge is an exception finalisation edge directly connecting an action node representing the throwing of an exception to nodes that represent the finalisation code. Such edges are absent in original AUGs because of the assumption that a method does not catch its own exceptions. However, this may happen, such as to escape control from a highly nested loop.[2] As we will see later, the ability to store such edges is especially necessary in iAUGs, where a callee may throw an exception which is caught by its caller. When the callee is inlined into the caller, the caller's iAUG will contain both the throwing node as well as the catching node.

Our second representation, the iAUG, is an extension of the eAUG. The main difference between the two is that an iAUG can be a lot larger than its corresponding

[1] As we will see later, the inlining order is semi-consistent, rather than fully consistent, since the order in which two different branches are explored individually is undefined. However, our inlining algorithm guarantees that calls are inlined in a way that is consistent with the partial call order.

[2] Although this may be considered a code smell by some, it is valid code nonetheless.

eAUG, since the method's callee eAUGs have been inlined into it. iAUGs additionally keep track of the methods that have been inlined into it (its direct or indirect callees), as well as the methods it has been inlined into itself (its direct or indirect callers). This information is used during inter-procedural filtering, so that we do not have to inspect call graphs to discover caller-callee relations.

4.2 Graph Inlining

Our tool is able to mine for method boundary-agnostic inter-procedural patterns and detect violations of these patterns by creating a composition of a project method and all of its first-party callees. It creates such compositions by embedding the graphs that represent the callees' bodies into the top-level method, a process called inlining. Essentially, our iAUGs provide a broader view of the target code base, as if the API usage that is scattered across multiple methods, was in fact all placed in a single method body.

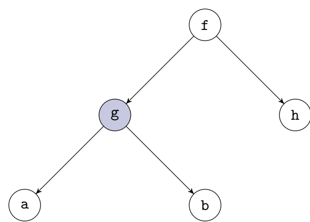


Figure 4.3 – An example call graph.

There are many different ways in which an API usage can be scattered across functions.[3] Consider the example call graph in Figure 4.3, and assume we are focusing on the function `g`, marked in gray shading. The function `g` has two callees (`a` and `b`), one caller (`f`), and one sibling (`h`, a callee of its caller). Assume further that `g` contains a partial API usage that we are interested in, and the remainder of this usage is present in its surrounding call context.

If the rest of the API usage is implemented by one of `g`'s callees, then inlining `a` and `b` into `g` will provide us a full view of the API usage in the iAUG of `g`. Similarly, if the rest of the API usage is implemented in `g`'s caller, then inlining `g` into `f` gives us a full view of the usage in the iAUG of `f`. Finally, if `h` contains the remainder of the implementation of the usage, then the iAUG of `f` will also uncover this inter-procedural usage. This last situation is especially interesting, as `g` and `h` are not directly related. However, inlining allows us to uncover such hidden relations.

Continuing with the same example, assume that both `g`'s caller and callees contain API usages that are part of the larger, inter-procedural usage, i.e. the API usage spans a larger path through the call graph. The simple inlining described in the previous paragraph is insufficient to uncover such larger usages. However, inlining can be done recursively: First, we inline `g` (and `h`) into `f`, then, we inline `g`'s callees into the partially inlined iAUG of `f`. The resulting iAUG then embeds the usages of all of the functions shown in the example. We say that this iAUG is the result of inlining up to a depth of 2, since two levels of the call graph rooted at `f` have been inlined into it.

In theory, we can perform inlining up to arbitrary depths. For example, if we start our inlining process at the main method of a Java program, and choose the depth to be large enough, the resulting iAUG can capture the whole program. In practice however, the feasible inlining depth is limited, since the number of inlined methods increases exponentially as the inlining depth increases. Indeed, if we assume every program method calls at most m other methods, and we inline up to a depth of d, the resulting number of inlinings performed is in $O\left(m^d\right)$. Since each of the inlinings copies all of

[3] Even though we specifically focus on methods and object-oriented programs, a generalisation can be made to functions as well.

the nodes of the callee eAUG into the iAUG, the final iAUG will be enormous for large values of d. Assuming every method eAUG contains at most n nodes, then every inlining may grow the resulting iAUG with n nodes, and the final iAUG will have a node count in $O\left((nm)^d\right)$. Because of transitive order edges, AUGs, and by extension eAUGs and iAUGs, tend to be dense graphs. In essence, for larger inlining depths, iAUGs tend to be very large, densely connected graphs. Since frequent subgraph mining suffers from combinatorial explosion as it is, and is provided with an iAUG for every method in the code base, even moderate inlining depths can be practically infeasible. We go more in depth on this problem in our performance evaluation, in Section 5.4.

Thus, the choice of inlining depth is vital to ensure feasibility of the analysis. However, one can think of many other questions related to inlining. For example, if our inlining algorithm encounters a recursive call, we can either choose to inline it again, or skip this call. Different answers to such questions lead to different iAUGs and may, as a result, lead to different patterns. Additionally, these questions can have a significant impact on the size of the graphs. For instance, every method call we choose not to inline eliminates the presence of this method's nodes in the final iAUG, as well as all of the method's callees. We identify three main questions:

– Should recursive calls be inlined?

– Should duplicate calls be inlined?

– Should methods lacking target API usage be inlined?

Since we do not know which answers lead to the better patterns, we implement six inlining strategies that differ in their handling of the cases described in these questions, which we later experimentally evaluate.

The remainder of this section is dedicated to describing our inlining algorithm in detail. We first introduce our general depth-first inlining algorithm, which takes as input an eAUG of a method, and returns the corresponding iAUG. This is covered in Section 4.2.1. Then, in Section 4.2.2, we describe the six inlining strategies that dictate which method calls should be inlined, as introduced above. Finally, we describe in detail the process of inlining an individual callee into the iAUG in Section 4.2.3.

4.2.1 Main Inlining Algorithm

Our main inlining algorithm can best be described as a pre-order depth-first traversal through an implicitly-generated call tree of a method. Note that we use the term call tree, rather than call graph, so that we can define the inlining depth as the depth of the call tree in the presence of recursion. When a method recursively calls itself, either directly or indirectly, a call tree can be of infinite depth. However, the inlining algorithm is always limited by the user-provided maximum inlining depth, so a potentially infinite call tree is only explored up to this depth.

Listing 4.1 shows a pseudo-code description of our main inlining algorithm. It first creates a copy of the given eAUG, which becomes the iAUG under construction (line 3). It additionally initialises the set of method eAUGs that have been inlined with the top-level method's eAUG (line 4), indicating that the top-level method's eAUG is already contained in the iAUG. Then, it starts the depth-first inlining process by calling the

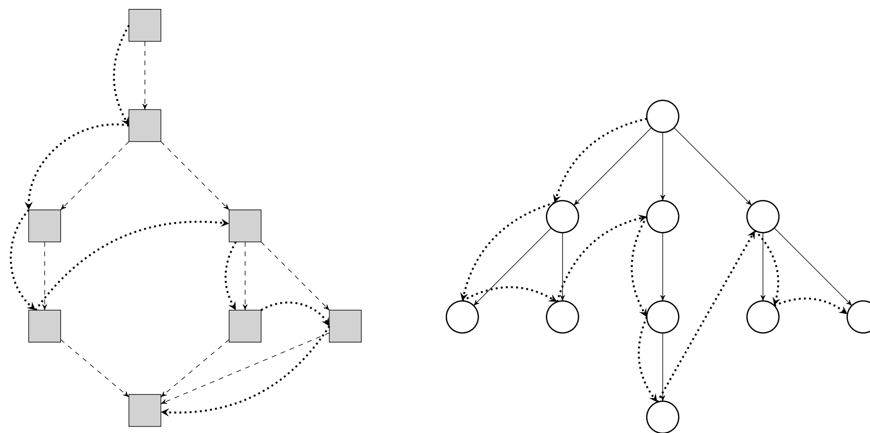
```
function Inline(eAUG, maxDepth)                                              1
returns "an iAUG corresponding to eAUG, inlined up to maxDepth"              2
    global iAUG ← Create−iAUG(eAUG)                                          3
    global inlinedAUGs ← {eAUG}                                              4
    Inline−Depth−First(maxDepth, [eAUG], eAUG)                               5
                                                                             6
function Inline−Depth−First(maxDepth, callStack, targetAUG)                  7
    callNodes ← Discover−Call−Nodes(targetAUG)                               8
                                                                             9
    for callNode ∈ callNodes do:                                             10
        calleeAUG ← Get−eAUG(callNode)                                       11
        if (Is−First−Party(calleeAUG) and                                    12
              Should−Inline−Node(calleeAUG, maxDepth, callStack, inlinedAUGs)): 13
            iAUG ← Inline−Node(iAUG, callNode, calleeAUG)                    14
            inlinedAUGs ← {calleeAUG} ∪ inlinedAUGs                          15
            Inline−Depth−First(maxDepth, [calleeAUG] + callStack, calleeAUG) 16
```

Listing 4.1 – Pseudo-code describing the main inlining algorithm.

function Inline-Depth-First (line 5). This function takes as its input the current call stack, initially containing the top-level method, and a target eAUG which has already been inlined in the iAUG, but whose nodes still need inlining. It first discovers all method call nodes in the target eAUG through the Discover-Call-Nodes function (line 8), which returns a list of nodes, ordered by their partial execution order. This list is ordered in such a way that the algorithm always inlines nodes in order along a sequential path through a method's control flow graph, and in case of branches, inlines nodes sequentially along one branch, halts at the join point of the branches, inlines the other branch sequentially, and then continues after the join point. In other words, one branch is fully inlined before the other branch is inlined, leading to a semi-consistent inlining order since it is undefined which of the two branches is selected first. One such discovery order is displayed in Figure 4.4a.

The algorithm then considers each node in order, and checks whether the call node should be inlined. Calls are only inlined if the target method is first-party (line 12), since API usages consist of interactions with third-party methods and should therefore not be inlined. The algorithm additionally queries the inlining strategy (line 13), which decides whether a call should be inlined based on the target method, the maximum inlining depth, and the previously inlined calls, both globally (`inlinedAUGs`) and in the current path down the implicit call tree (`callStack`). If the node is to be inlined, it performs the inlining process, which substitutes the target method's eAUG for the call node (line 15). The target AUG is added to the set of previously inlined AUGs, so that this information can be used in later calls to the inlining strategy, and finally the algorithm descends into the target eAUG itself to find new callees to be inlined. The traversal is depth-first in the call tree, which is consistent with the execution order: If a call to a function `f` is executed before the call to a function g, then all of `f`'s callees are executed before g too, and thus, all these callees are inlined before g is inlined. This pre-order depth-first traversal of the call tree is shown graphically in Figure 4.4b.

Note that, in many of the cases, the inlining order does not make a difference. However, when a strategy prevents inlining duplicate calls, this order does matter, since only the very first call to a method will be inlined. There are multiple ways one can define the "first" call to a method. We take the intuitive approach by stating that the first call to a method is the earliest possible call executed by the program at runtime, which is why the traversal is pre-order depth-first. Duplicate removal is path-insensitive: We deal with branches in control flow by assuming a total order and analysing one branch before the other, and the resulting set of previously-inlined



(a) Graphical representation of the call node discovery in a simplified eAUG. Data nodes are omitted. Branches in the graph represent branches in control flow. The dotted arrows represent the order in which nodes are selected for inlining in a single graph.

(b) Graphical representation of the inlining order in a call tree. Execution order of children (callees) in the tree is left-to-right. The dotted arrows represent the order in which callees are selected for inlining.

Figure 4.4 – Graphical representations of the inlining order in a single method (left) and a collection of methods as a call tree (right).

eAUGs will contain the eAUGs inlined in the first branch when processing the second branch.

4.2.2 Inlining Strategies

As described in the beginning of this section, there are three main questions one must answer during inlining. We repeat these questions here for convenience:

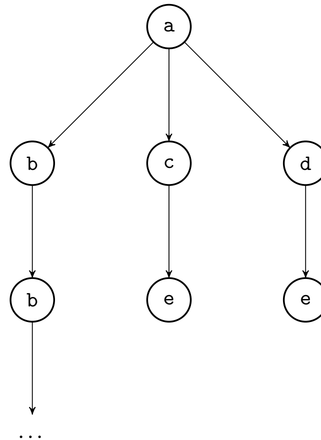
- *Should recursive calls be inlined?*
- *Should duplicate calls be inlined?*
- *Should methods lacking target API usage be inlined?*

Inlining recursive and duplicate calls may significantly increase the size of the resulting iAUG, and may lead to overfitting during the mining phase because of duplicated phenomena. However, not inlining such calls may cause patterns to miss significant features, such as necessary repetitions. Similarly, not inlining methods that contain no direct usage of the targeted API may decrease the size of the resulting iAUG and lead to more concise patterns, since the iAUGs contain less unrelated phenomena. However, it will not explore transitive callees found in the skipped method, which may themselves contain API usages. Such transitive API usages will then be missed.

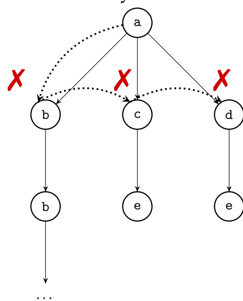
Answering these questions involves a trade-off between the size of the iAUGs and the relevance and correctness of the resulting patterns. Thus, we implement six different inlining strategies, five of which provide different answers to the questions. These strategies implement the Should-Inline-Node function in the pseudo-code shown in Listing 4.1. Figure 4.5 shows an overview of the behaviour of the inlining strategies.

```
class C {
    void a() {
        b(); c(); d();
    }
    void b() {
        api.m(); b();
    }
    void c() {
        api.m(); e();
    }
    void d() {
        e();
    }
    void e() {
        api.m();
    }
}
```

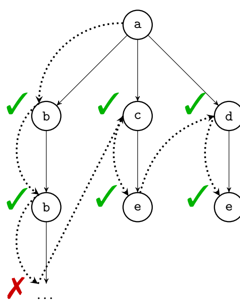
(a) Contrived example code used to illustrate the inlining strategies. Method `a` is the top-level method used in the examples, which is to be inlined. Method `b` contains direct recursion, method `c` and `d` contain a duplicate call to `e`. Additionally, method `d` contains no direct API usage, but transitively uses the API through `e`.



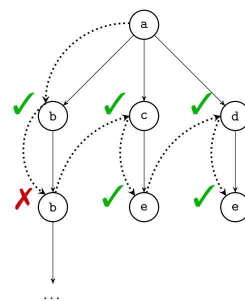
(b) The call tree representing the example code. API method calls are omitted for brevity, as they are never inlined.



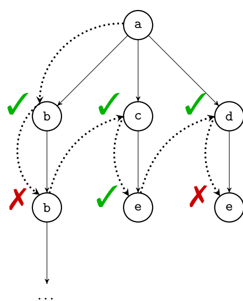
(c) Inlining traversal as guided by the no inlining strategy.



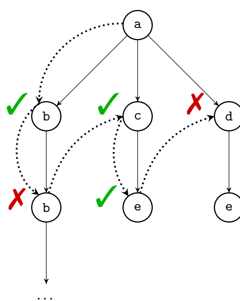
(d) Inlining traversal as guided by the inlining cutoff strategy, with a maximum inlining depth of 2.



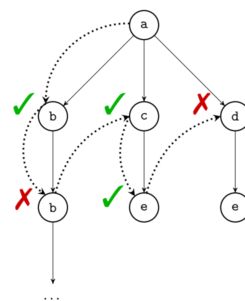
(e) Inlining traversal as guided by the no recursion strategy.



(f) Inlining traversal as guided by the no duplicates strategy.



(g) Inlining traversal as guided by the no recursion strategy with pre-emptive cutoff.



(h) Inlining traversal as guided by the no duplicates strategy with pre-emptive cutoff.

Figure 4.5 – Examples illustrating the inlining strategies. Figure a shows the example code, Figure b the corresponding (simplified) call tree. Figures c–h show the inlining traversal under different strategies. Calls that are decided to be inlined by the strategy are labelled with a green tick, whereas calls that are ignored are labelled with a red cross. Maximum inlining depth for all strategies is set to 2.

The No Inlining Strategy This first strategy is trivial: It always returns `false` when asked if a node should be inlined. This allows our tool to be customised to only mine and detect intra-procedurally. Using this strategy, no inlining ever takes place. We include this strategy mainly to compare our approach to MUDetect in the evaluations, and is not intended to actually be used in practice. Figure 4.5c shows a typical application of this strategy. Note that the depth-first traversal of the call tree is still applied, but the strategy simply prevents any inlining or further exploration of the tree.

The Inlining Cutoff Strategy The second strategy is similar to the one implemented in the first version of MAPO (Xie & Pei, 2006). This strategy inlines every method call up to a given maximum inlining depth `d`, and thus answers the main questions as follows:

– *Should recursive calls be inlined?* Yes.

– *Should duplicate calls be inlined?* Yes.

– *Should methods lacking API usage be inlined?* Yes.

The inlining cutoff strategy decides on whether a node should be inlined by calculating the depth of the current path down the call tree, using the given call stack, and comparing it to the maximum depth. It allows methods to be inlined as long as it would not exceed the maximum inlining depth. Figure 4.5d shows this strategy in action with a maximum call depth of 2: It allows all of the nodes on levels 1 and 2 of the call tree to be inlined, but cuts off the exploration at level 3.

The No Recursion Strategy Our third strategy eliminates recursive calls during inlining. Whenever a call would lead to recursion, either directly or indirectly (such as in mutual recursion), the call is prevented from being inlined to avoid duplicated phenomena in the resulting iAUG. This strategy is also parametrised with a maximum inlining depth, as in the inlining cutoff strategy, to prevent iAUGs from getting too large. As such, it provides the following answers to our three main questions:

– *Should recursive calls be inlined?* No.

– *Should duplicate calls be inlined?* Yes.

– *Should methods lacking API usage be inlined?* Yes.

We keep track of potential recursion by maintaining a stack of ancestor eAUGs throughout the descent of the call tree, i.e. the path of methods entered to reach the method call to be inlined. This stack resembles the call stack of the program at runtime if it started at the top-level method and reached the method call on which the strategy has to decide. To determine if the call would lead to recursion, we then simply have to check whether the callee eAUG is present in this stack. If it is, the strategy prevents the call from being inlined. Additionally, it cuts off the exploration if it exceeds the maximum inlining depth, in a manner identical to the cutoff strategy. An exploration using this strategy is shown graphically in Figure 4.5e, where the inlining process cuts off at the recursive call to `b` and continues with `c`.

The No Duplicates Strategy The fourth strategy implemented in our tool eliminates both duplicate calls and recursive calls, and is inspired by the strategy employed in the second version of MAPO (Zhong et al., 2009). Note that recursion is a special case of duplication, and thus it does not make sense to have a strategy that removes duplicate calls, but not recursive calls. This strategy prevents a call from being inlined it is has ever been inlined before, not necessarily in the same path down the call tree. This way, we only retains the call which is executed first at runtime. Retaining duplicates in the iAUG leads to potential overfitting during the mining, since the duplicated evidence may be considered a pattern if it occurs too frequently. Thus, preventing duplicates may help prevent overfitting. In short, this strategy's answers to the main questions are:

- *Should recursive calls be inlined?* No.
- *Should duplicate calls be inlined?* No.
- *Should methods lacking API usage be inlined?* Yes.

During the traversal of the call tree to inline calls, the algorithm maintains a set of all eAUGs that are already inlined into the iAUG. When the eAUG of the callee to be inlined is already present in this set, the call is a duplicate and is skipped by the exploration guided by this strategy. The no duplicates strategy also extends the cutoff strategy, to limit the exponential growth of the iAUG, and thus may decide to not inline a call even if it is not a duplicate, but exceeds the depth threshold. Figure 4.5f displays this strategy applied on the example code. As with the previous strategy, the recursive call to `b` is not inlined, since recursion is removed. Additionally, we prevent inlining of the second call to `e`, which is called by `d`, since `e` has already been inlined through `c`, and is therefore a duplicate.

The No Recursion Strategy With Pre-Emptive Cutoff To further keep the iAUG from growing to an excessive size due to the exponential growth of inlining calls, our fifth strategy prevents inlining any method in which it cannot find direct proof of API usage. This removes many unrelated phenomena in the final inlined AUG, but may also incorrectly remove API usage from the method's transitive callees. This is because we only check for API usage intra-procedurally in the eAUG to be inlined, and stop inlining pre-emptively if there is no usage. Our reasoning is that API usage in transitively called methods is likely unrelated to the usage found in the caller if the intermediate method contains no usage whatsoever. Our three main inlining questions are answered below.

- *Should recursive calls be inlined?* No.
- *Should duplicate calls be inlined?* Yes.
- *Should methods lacking API usage be inlined?* No.

This strategy builds further upon the no recursion strategy, so it first checks for the inlining depth cutoff and the recursion elimination. Only when both of these checks return a positive result, allowing the call to be inlined, will this strategy check for API usage in the callee's eAUG. If this eAUG contains no usage of the target API, it is not selected for inlining. This is shown in Figure 4.5g, where the call to method `d` is not inlined since it does not contain any direct API usage. This example also shows that

the second call to `e` is never considered for inlining, even though it contains an API usage, since we check for API usage intra-procedurally.

Two more points are noteworthy. First, this strategy only makes sense when we are targeting one or more specific APIs. When no API is targeted specifically, our goal is to mine patterns involving any API, and we can therefore not prevent inlining of any method. Second, this strategy does not prevent inlining if the top-level method contains no direct API usage. This can also be seen in the example: The method `a` contains no direct API usage, but its callees are still inlined. This approach is intentional, so that our tool can discover patterns scattered across multiple sibling callees although the top-level method contains no direct usage. For example, recall the call tree given in Figure 4.3, and assume that function `f` contains no direct usage, but functions `g` and `h` contain a usage which, if composed, make up a meaningful pattern. Even though `f` contains no usage, inlining its callees will uncover a usage which is otherwise difficult to detect, since `g` and `h` are not directly related in a caller-callee relation.

The No Duplicates Strategy With Pre-Emptive Cutoff Our tool's final inlining strategy combines the previous two strategies to provide a negative answer to each of the main inlining questions:

- *Should recursive calls be inlined?* No.
- *Should duplicate calls be inlined?* No.
- *Should methods lacking API usage be inlined?* No.

As such, this strategy is able to remove more irrelevant or duplicated phenomena than any of the other strategies. In essence, it is the logical conjunct of the result of the no duplicates strategy, and a pre-emptive cutoff as in the previous strategy. Figure 4.5h shows this graphically, and leads to the same exploration as the previous strategy. This is because the call to method `d` is ignored by the pre-emptive cutoff. However, if `d` were instead to contain an API usage of its own, the resulting exploration would be equivalent to the one in Figure 4.5f.

Summary of Inlining Strategies Five of the six strategies are designed to provide differing answers to our three main questions, which leads to different iAUGs and thus, different patterns. We empirically evaluate each of these strategies in our evaluation, to determine which strategies lead to better recall and precision. The no inlining strategy is designed to perform no inlining, as is evident from its name, and is only present as a comparison to MUDetect. We summarise the six strategies in Table 4.1.

4.2.3 Inlining Individual Callees

We have yet to describe how method bodies are inlined into the iAUG of their caller. Although this may appear as a trivial task of copying over nodes and edges, one must carefully take the semantics of the methods into consideration, ensuring that control and data flow between the caller and callee is consistent. Since method boundaries are erased, we must appropriately adjust the composite graph's edges. For instance, the callee's control flow should not influence the control flow of the caller, but the callee should inherit the control flow of the caller. For example, when the callee is called in a

Strategy	Inline... recursive calls?	duplicate calls?	calls w/o API usage?	Inlining depth
No inlining	✗	✗	✗	0
Depth cutoff	✓	✓	✓	$\geqslant 1$
No recursion	✗	✓	✓	$\geqslant 1$
No duplicates	✗	✗	✓	$\geqslant 1$
No recursion with pre-emptive cutoff	✗	✓	✗	$\geqslant 1$
No duplicates with pre-emptive cutoff	✗	✗	✗	$\geqslant 1$

Table 4.1 – Overview of the behaviour of the inlining strategies

branch, the inlined graph should be extended so that the callee's nodes are also called in the branch. Additionally, the caller and callee may share the same data, for example through parameters or instance fields, and thus, when the callee is inlined, its data nodes must be adjusted and shared with the caller.

A condensed version of the Inline-Node function, which inlines the method body of a method call target into an iAUG, is shown in Listing 4.2. It consists of two main steps. First, we inject the nodes and edges of the callee's eAUG into the iAUG of the caller, leading to two disconnected subgraphs in the iAUG. Then, we connect the two disconnected subgraphs based on the incoming and outgoing edges of the original call node. Many of the different edge types need to be handled uniquely, to account for semantics. Additionally, these functions may add new edges, for example to instantiate a transitive relation.

As mentioned above, we must ensure that inlining the callee does not lead to undesired changes to the control flow of the caller. The callee's graph may contain nodes that influence its control flow, and when connecting it to the inlined AUG, these nodes may lead to such undesired changes. There exist four action node types that denote changes in control flow: Return nodes, throw nodes, continue nodes and break nodes, which correspond to their respective statements. The latter two do not form a problem, as they are local to a loop in the callee, and therefore cannot influence the existing control flow of the caller. The former two cause an escape from the control flow of the callee and, when inlined into the caller, may mistakenly cause an escape from the caller's control flow instead. This is illustrated in Figure 4.6. The first example shows the presence of a `break` statement, which solely influences the control flow in a loop. Naively inlining this code into the caller does not lead to changes in the caller's control flow outside of this loop. However, the second example shows a naive inlining of a `return` statement, which does lead to changes in the caller's control flow. Due to naively inlining this statement, the call to `doB` is never executed, whereas in the original code, it would be executed. Note also that this breaks the typing of the method: `caller` is said to not return anything, but naively inlining would lead to the method returning an integer.

Thus, when injecting nodes into the caller's iAUG, we do not copy over return nodes to prevent such cases from happening. Return nodes are used in one of two

```
function Inline-Node(iAUG, callNode, calleeAUG)                                1
returns "the iAUG with callNode replaced by calleeAUG"                         2
    for calleeNode ∈ Nodes(calleeAUG) do:                                      3
        if Type-Of(calleeNode) ≠ ReturnNode:                                   4
            Add-Node(iAUG, calleeNode)                                         5
                                                                               6
    for calleeEdge ∈ Edges(calleeAUG) do:                                      7
        if Type-Of(Target(calleeEdge)) ≠ ReturnNode:                           8
            Add-Edge(iAUG, calleeEdge)                                         9
                                                                               10
    for inEdge ∈ Incoming-Edges-Of(callNode, iAUG) do:                         11
        switch Type-Of(inEdge) do:                                             12
            case OrderEdge ⇒ Relink-Incoming-Order-Edge(inEdge)                13
            # ...                                                              14
                                                                               15
    for outEdge ∈ Outgoing-Edges-Of(callNode, iAUG) do:                        16
        switch Type-Of(outEdge) do:                                            17
            case OrderEdge ⇒ Relink-Outgoing-Order-Edge(outEdge)               18
            # ...                                                              19
                                                                               20
    Remove-Node(iAUG, callNode)                                                21
    return iAUG                                                                22
```

Listing 4.2 – Pseudo-code describing the Inline-Node function of the inlining algorithm.

cases: To signify an early change in control flow of the callee, or to propagate data from the callee to the caller. When inlining, the former case is handled by execution order edges, whereas the latter case is handled by sharing the callee's returned data node with the caller. Therefore, the callee's return nodes are unnecessary and undesired in the caller's inlined AUG. Note that throw nodes are still injected into the caller, even though they may cause control flow changes as well. However, a throw statement in a callee method's body may influence the control flow of the caller if the caller fails to catch the exception. Thus, we copy over throw nodes during the inlining, and link it to exception-handling nodes of the caller if such nodes are present. This ensures that the caller's control flow is equivalent to a non-inlined version, regardless of whether it catches the exception.

In short, our first step of the inlining injects the callee's nodes and edges into the caller, except for return nodes and edges connecting to such nodes, to ensure no changes to control flow are mistakenly introduced. The second step then relinks the edges of the call node to connect the inlined graph, potentially adding edges to keep control and data flow consistent. We describe this process for each of the potential edge types graphically in the remainder of this subsection.

Relinking Data Flow Edges

Data flow edge adjustments often involve merging a data node from the caller with a data node of the callee, to indicate that both share the same data. Thus, we distinguish five cases of adjustments that need to be made, based on the data edge that is being used. For outgoing definition edges, we merge the data returned from the callee with the data node of the caller that is defined by the call. The callee's parameter data nodes are merged with the caller's argument data nodes, as linked to the call with incoming parameter edges. An incoming receiver edge to the call in the caller iAUG is handled by both merging the callee's `this` instance with the receiver, and merging any shared instance field usage across the caller and callee. Outgoing throw edges that link the original call to the exceptions it may throw are relinked to the nodes in the callee that

```
class C {
    void caller() {
        doA();
        callee();
        doB();
    }

    void callee() {
        for (int i = 0; i <= 5; i++) {
            if (i % 2 == 0) {
                break;
            }
        }
    }
}
```

(a) Example of a callee with a node that influences control flow local to a loop.

```
class C {
    void caller() {
        doA();
        for (int i = 0; i <= 5; i++) {
            if (i % 2 == 0) {
                break;
            }
        }
        doB();
    }
}
```

(b) Naively inlining the callee of the example in Figure a into the caller. This does not lead to undesired changes in control flow of the caller.

```
class C {
    void caller() {
        doA();
        callee();
        doB();
    }

    int callee() {
        return 0;
    }
}
```

(c) Example of a callee with a node that influences control flow of the whole method.

```
class C {
    void caller() {
        doA();
        return 0;
        doB();
    }
}
```

(d) Naively inlining the callee of the example in Figure c into the caller. This leads to undesired changes in the control flow of the caller.

Figure 4.6 – Examples illustrating undesired control flow changes through naive inlining. On the left are original examples, on the right are the examples where the `callee` method has been inlined into the `caller` method in a naive manner.

may throw this exception. Finally, incoming containment edges tell us that a method call is contained within a method body of an anonymous class instance, or a lambda, and hence, all inlined nodes of the callee are also contained within this method body. A graphical representation of the relinking process for each of these cases is shown in Figure 4.7, and is further explained below.

The observant reader may notice that certain connection types are not handled. This is because such connections cannot be present in the AUG. For example, qualifier edges can only connect two data nodes, and thus never involve a method call node. As another example, a parameter edge can never have a method call node as a source, since it links a data node used as a parameter to the use site of the parameter, and a call can never be a parameter.[4] In general, data flow edges always involve at least one data node as either its source or its target, and thus call nodes can only be present at one side of the edge.

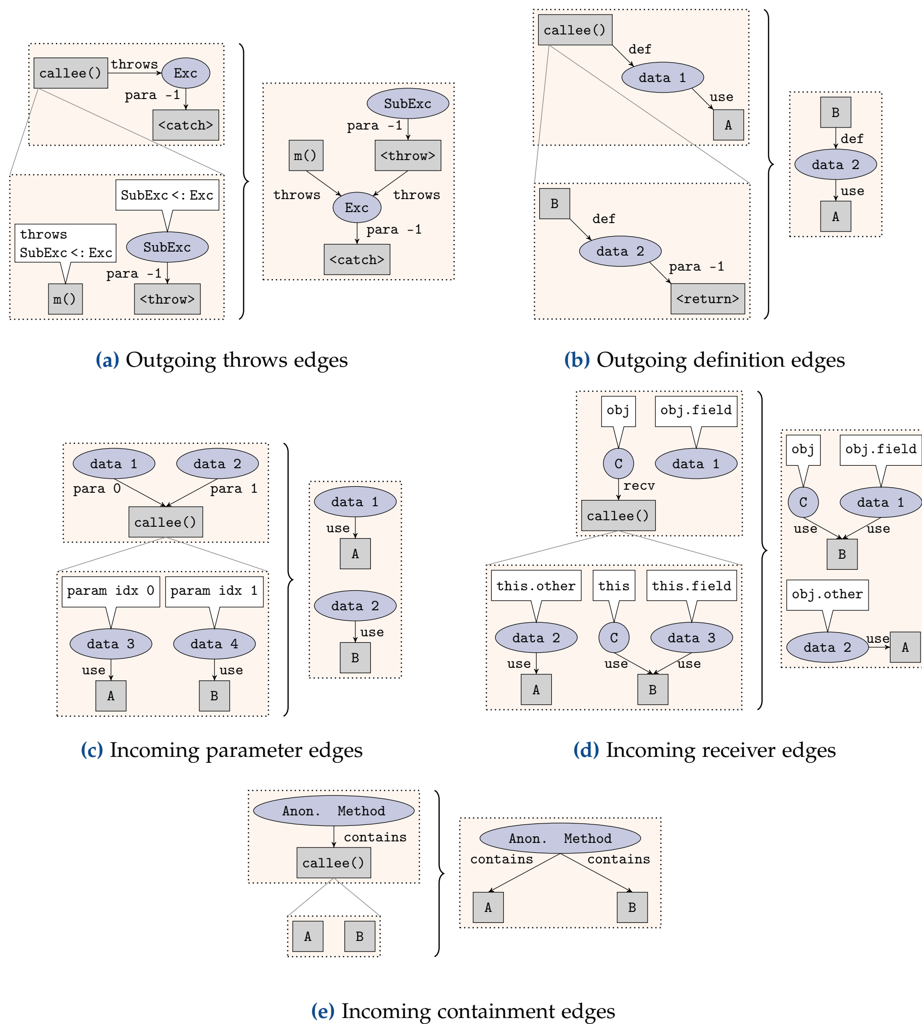
Outgoing Throws Edges A "throws" edge starting at a call indicates that this call may lead to an exception being thrown, which is then caught by the caller. This happens when a call is placed inside of the try-block of a try-catch construct. When we inline the callee, we then must also ensure that its actions are considered part of this try-clause. However, not all actions in the callee throw an exception that is caught, and naively relinking the throws edge would lead to superfluous edges. Thus, we identify the callee's action nodes that may throw an exception that is caught by the caller, and add a throws edge going from these action nodes to the exception that is thrown.

Figure 4.7a shows an example of this. The callee can contain two types of nodes that may throw an exception: A direct throw node (on the right in the callee) and a transitive method call that may throw the exception (on the left). Both of these are then connected to the thrown exception. Note that we account for subtype relations using additional type hierarchy information, and use a method's `throws` information to determine whether it may throw the exception. Additionally, we ignore any potentially thrown exceptions if they are already caught by the callee itself.

Outgoing Definition Edges When a method call returns data to the caller, and the caller uses this data by e.g. storing it in a variable or chaining it into a new method call, the caller will contain a data node which is defined by the call, as indicated by an outgoing definition edge. Such an example is shown in Figure 4.7b, where the call to `callee` defines the data node `data 1`, which is later used in action `A`. The callee then contains a return node, and the returned data node is connected to this node by a parameter edge. In the example, this data node is `data 2`, produced by action `B`.

Essentially, `data 1` and `data 2` would be the same data node in the inlined AUG, and since return nodes are removed for reasons described earlier, we need to relink such edges correctly. Thus, we remove `data 1`, the data node of the caller, and retain `data 2`, the data node of the callee, with all of the edges present in the callee. All of the outgoing use edges of `data 1`, such as the use edge to action node `A`, are relinked to have the callee's node as the source. Therefore, we make the data sharing relation between caller and callee concrete. Note that if a callee may return one of multiple data nodes because of branching, then all of the callee's returned data nodes are retained, and they receive the same relinking.

[4] In source code, this is not necessarily true: There may be cases where the return value of a call is immediately passed as an argument to another call. However, in AUGs, such chaining is represented using additional temporary data nodes.



(a) Outgoing throws edges **(b)** Outgoing definition edges

(c) Incoming parameter edges **(d)** Incoming receiver edges

(e) Incoming containment edges

Figure 4.7 – Condensed overview of data flow edge relinking during inlining. The partially inlined AUG of the top-level caller is displayed in the upper left portion of the figure, the lower left portion shows the callee's eAUG. The right-hand side of the brace shows the iAUG of the caller with the callee's eAUG inlined and the edge relinked. Irrelevant nodes and edges are omitted from the representation. Action nodes whose type is irrelevant carry generic labels such as `A` and `B`.

Incoming Parameter Edges A similar data sharing relation takes place when a caller provides the callee with data via call arguments. Such a case is illustrated in Figure 4.7c, where the call to `callee` receives two arguments: `data 1` and `data 2`. The order in which these parameters are given can easily be identified by inspecting the parameter edges' labels. The callee then has two different data nodes for the arguments, here `data 3` and `data 4`. Using our eAUG representation, which maps parameter indices to the data nodes representing the parameter values, we can also easily identify which data node corresponds to which parameter, and we can therefore relate `data 1` in the caller to `data 3` in the callee, and `data 2` to `data 4`.

When inlining the callee into the caller, we then remove all data nodes corresponding to parameter values of the callee, and instead use the data nodes used as arguments in the caller. To this extent, our tool relinks the outgoing use edges of the data values to have the argument data as the source node instead. We can see this in the example,

where the inlined AUG contains the two action nodes of the callee, but uses the data nodes of the caller instead.

Incoming Receiver Edges Callers not only share data with callees via parameters, they may also share data via the callee's receiver instance, i.e. the object on which the callee method is called. In this case, we need to ensure both that the `this` object in the callee is consistent with the receiver of the method call, and that the fields of this object which are used in the callee, are shared with the caller.

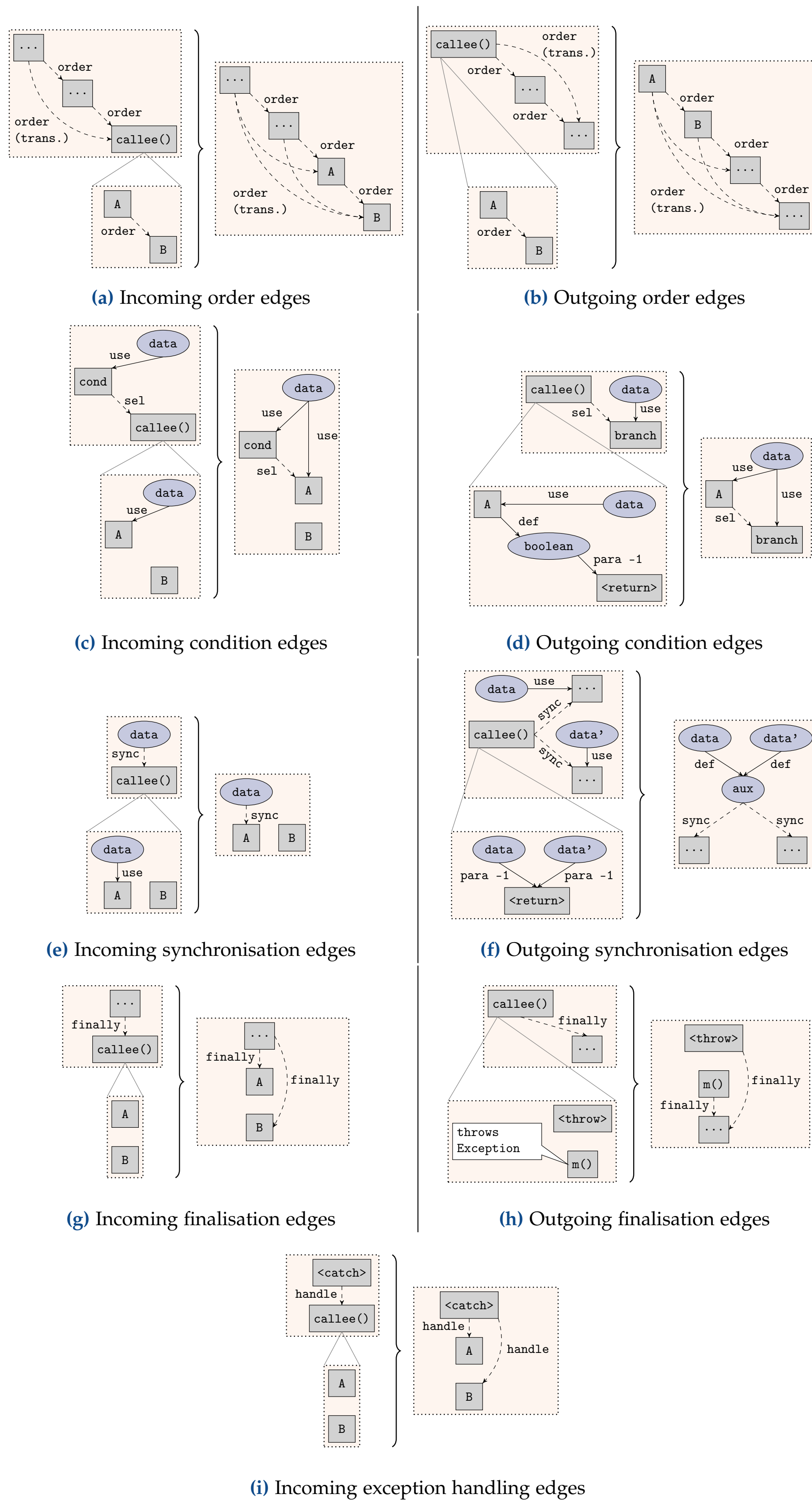
In the illustration in Figure 4.7d, the caller calls the method `callee` on the object `obj`, and also uses this object's field `field` elsewhere. The callee uses its own instance (`this`) and this field in an action `B`, as well as a new field `other`, unknown to the caller, in an action `A`. When we inline this callee into the caller, we need to make sure that the action `B` uses `obj` rather than `this`, as to not confuse it with a call on the caller's instance. We additionally need to ensure that the data node for `obj.field` is shared among the caller and callee, as well as the data node for `obj.other`, so that any future inlinings can share it. This can be seen in the inlined AUG on the right-hand side of the brace, where the callee's action `B` now uses the data nodes of the caller, and the callee's data node for the `other` field is now annotated to be a field of `obj`.

Incoming Containment Edges When a method constructs an instance of an anonymous class, such as a lambda, this instance is modelled as an anonymous class instance node. Such anonymous classes contain methods themselves, which are also modelled as anonymous method nodes. To model the code of these methods, a sub-AUG is created for the anonymous method, and linked to the data node representing the anonymous method via containment edges. Since such anonymous methods may themselves call other methods, these other methods are inlined as well, and hence a call can have an incoming containment edge. These are handled straightforwardly by instantiating containment edges for all inlined nodes, as shown in Figure 4.7e.

Relinking Control Flow Edges

Recall from Section 3.1 that an AUG can contain five types of control flow edges. We additionally consider the exception-handling edge, classified as a data flow edge, to be a control flow edge, as its relinking involves control flow information rather than data flow information. From these six edge types, we distinguish nine different cases to consider when relinking control flow edges: For order edges, finalisation edges, condition edges (both selection and repetition), and synchronisation edges, we consider both incoming and outgoing edges involving the call node. For exception-handling edges, we only consider incoming edges, since such edges always start at a catch node. We handle each of these cases uniquely, as illustrated in Figure 4.8.

Order Edges When a method performs a call to a callee, all of the callee's actions are executed after the caller's actions leading up to the call, and executed before the caller's actions following the call. Thus, when inlining a callee, to retain this ordering, we link up the call's direct and indirect incoming order edges to the source action nodes of the callee's eAUG. To instantiate the transitive execution order relation, we duplicate these edges to all of the action nodes that are reachable from the eAUG's source node, and mark them as transitive. This is shown in Figure 4.8a. An analogous



(a) Incoming order edges
(b) Outgoing order edges
(c) Incoming condition edges
(d) Outgoing condition edges
(e) Incoming synchronisation edges
(f) Outgoing synchronisation edges
(g) Incoming finalisation edges
(h) Outgoing finalisation edges
(i) Incoming exception handling edges

Figure 4.8 – Condensed overview of control flow edge relinking during inlining, following the same structure of Figure 4.7. Irrelevant nodes and edges are omitted from the representation.

case is made for outgoing order edges, as shown in Figure 4.8b, but here we instantiate transitive order edges from all nodes that may reach a sink node.

Condition Edges A call may be involved in a condition edge in one of two ways: It may be executed in a certain branch, or it may form a part of the condition that controls the branching. Recall that condition edges are only present between a condition and a node executed in a branch if the nodes share a data dependency. Thus, in both cases, we need to identify data dependencies between the action nodes in the call and the action node at the other side of the edge, which requires knowledge of data sharing relations between caller and callee. Therefore, this relinking is deferred until all data flow edges have been relinked.

When the call is a part of a branch, i.e. the condition edge has the call as the target, we identify all action nodes in the callee that are data-dependent on the condition, and instantiate the appropriate condition edges between the branching condition and the inlined callee action nodes. This is illustrated in Figure 4.8c, where node `A` shares data with the condition, whereas `B` does not.

When a call is instead a part of the condition, where the condition edge has the call node as source, we need to link all of the callee's action nodes that may influence the outcome of the condition to the caller's action node executed in the branch. To identify such action nodes in the callee, we start at its data sinks and traverse upwards in the callee's eAUG, following data flow edges to discover action nodes on which the condition is data-dependent. We then instantiate the appropriate condition edge for each of these action nodes of the callee. This is illustrated in Figure 4.8d, where the callee's action node `A` is identified as an influential node.

Synchronisation Edges Relinking synchronisation edges is fairly similar to relinking condition edges. When the call is the target of a synchronisation edge, i.e. the call is executed while the program holds a lock on some data, we relink this synchronisation edge to all of the callee's action node that are data-dependent on this lock. Figure 4.8e shows this for a caller that takes a lock on a data node.

When a call is instead the source of a synchronisation edge, this indicates that the program takes a lock on the return value of the call. Thus, we identify the data node that may be returned by the call by inspecting the callee's eAUG, and relink the synchronisation edge from this returned data node. If the call may return multiple distinct data nodes, we aggregate these data nodes into a new auxiliary node and relink synchronisation edges from this aggregate node. Alternatively, if we were to relink synchronisation edges from each of these distinct nodes individually, this may be confused with taking multiple locks, whereas in reality only a single lock is held.

Exception Handling and Finalisation Edges A call node may be the target of an exception handling or finalisation edge if the call is made in a catch or finally block of a try-catch-finally construct. Intuitively, when a method is called as part of exception handling code, all of the callee's action are also part of exception handling code. Thus, the exception handling edge is relinked to all of the callee's inlined action nodes, as shown in Figure 4.8i. Figure 4.8g shows the analogous case for incoming finalisation edges.

A call node may also be the source of a finalisation edge, when the call is made in the try block of a try-catch-finally construct, and there is a finally block defined in the

construct. Whenever an exception is thrown in the call, the finalisation code is executed, and therefore we relink the finalisation edges to start from all nodes that may throw an exception, which are direct throw nodes and transitive calls that may lead to an exception. This is similar to the relinking of throws edges, explained previously, and is illustrated in Figure 4.8h. Note that an analogous case could be made for exception handling edges, yet such edges do not appear in an AUG, since exception handling edges always have a catch node as their source, whereas finalisation edges do not, since finalisation code may be present even when the exception is not caught.

4.3 Inter-Procedural Filtering

Since patterns are mined from iAUGs rather than intra-procedural AUGs, the resulting patterns are inter-procedural as well. During violation detection, we continue to use the iAUGs to compute overlaps between a pattern and a target, so that we can detect pattern instances and violations in transitively called methods. However, due to this inter-procedural analysis, we may run into an increase in false positives caused by partial pattern overlaps in callees which are "completed" by the caller. This is alluded to by Li and Zhou (2005), who employ two different inter-procedural techniques to remove false positives. A violation in PR-Miner is one or more missing API calls in a function. If a function is missing an API call, they first check whether any of the function's callees contains the call, and then whether all of the callers of the function contain this call. If any of these two conditions is met, the violation is a false positive, since the API call is always present in the context of the function.

The former condition is unnecessary to check in our tool, since we detect overlaps inter-procedurally through iAUGs and hence, can detect continuations of pattern instances in a method's callees. However, the latter condition still needs to be checked, and in actuality, because of our usage of inter-procedural patterns, may cause many more false positives. We may additionally obtain duplicate violations, which cannot be considered false positives because there is an actual violation of the pattern, but only one of the violations is relevant to be considered a real misuse. We describe both of these cases by example in Section 4.3.1, and introduce our general inter-procedural filtering algorithm in Section 4.3.2.

4.3.1 False Positives and Duplicate Violations

To illustrate the problem of inter-procedural false positives and duplicate violations, assume we have a pattern of the form `DB.connect; DB.query; DB.close`, stating that these three methods must be called on a database object in order. We simplify our pattern for generality and brevity, but this can easily be extended to the AUG representation. As we will see later in this section, our filtering algorithm does not rely on the AUG representation, and can equally be used in other inter-procedural detection settings. The code example in Figure 4.9a shows an inter-procedural instance of this pattern rooted at the `executeQuery` method: All three of the methods are called in order, one (`DB.connect`) is called transitively through the `connectToDB` method. When our tool detects violations of the pattern in the `executeQuery` method, it receives the iAUG for this method, which has the `DB.connect` call inlined into it, so it can find the instance as expected. However, when our tool starts its violation detection at the `connectToDB` method, it finds a partial overlap between the pattern and this method: The call to `DB.connect` is present, yet neither `DB.query` nor `DB.close` are found in the

```java
class DBQueryEngine {
    Result executeQuery(String query) {
        DB db = new DB("localhost", 5000);
        connectToDB(db);
        Result result = db.query(query);
        logger.debug("Executed query");
        db.close();
        return result;
    }

    void connectToDB(DB db) {
        logger.debug("Connecting to DB");
        db.connect("admin", "*****");
        logger.debug("Connected to DB");
        // more logging, telemetry, ...
    }
}
```

(a) Code example showing an instance of the pattern.

```java
class DBQueryEngine {
    Result executeQuery(String query) {
        DB db = new DB("localhost", 5000);
        connectToDB(db);
        db.close();
        return result;
    }

    void connectToDB(DB db) {
        logger.debug("Connecting to DB");
        db.connect("admin", "*****");
        logger.debug("Connected to DB");
        // more logging, telemetry, ...
    }
}
```

(b) Code example showing a violation of the pattern.

Figure 4.9 – Contrived code examples showing an inter-procedural instance and an inter-procedural violation of the pattern `DB.connect; DB.query; DB.close`.

method itself, or any of its callees. Such a partial overlap would mistakenly be flagged as a violation, although all of the method's callers (here, only `executeQuery`) ensure the pattern is instantiated completely.

The second code example, shown in Figure 4.9b, contains an inter-procedural violation of the pattern: Both `DB.connect` and `DB.close` are called, but the call `DB.query` is missing in both caller and callee. Our tool would report a violation present in both `executeQuery` as well as `connectToDB`, since the former is missing the query, and the latter is missing both the query and the closing of the connection. In both cases, the query is missing, so there is one real violation, the other violation is a duplicate of the first one. Let us assume the call to `DB.query` needs to be placed in the caller, and thus, the caller is to blame for the violation: The violation in the callee would be a false positive, falling into the same category as the one above, but cannot be detected as such, since not all of `connectToDB`'s callers correctly instantiate the pattern. On the other hand, assume that the call needs to be present in the callee instead. Then, the violation in the caller is actually present transitively, but should not be reported, since the caller is actually not to blame, and the callee should be fixed instead.

It is generally difficult, if not impossible, to determine whether an absent call should be present in the caller or the callee. In this example, the violation is on a method boundary, making this task even more difficult: One part of the pattern is instantiated in the caller, another part in the callee, and the missing part is in the middle of the two instantiated parts. We cannot reliably determine whether it is the caller or the callee which should instantiate this missing portion. Although our representation erases method boundaries, opting to keep these does not alleviate this problem completely. It would allow us to state, for example, that the majority of instances of the pattern have the caller instantiate the portion of the pattern. However, only the developer who wrote the code really knows what the intent is. Yet, we still need to filter out such duplicate violations, and thus, we need to resort to heuristics.

To heuristically determine which method is to be blamed for a given violation, we investigate the callers of a method. Given a pair of caller and callee methods containing the same violation, our tool inspects the all of the callee's other callers in search of the same violation. If all of these other callers contain the same violation, we heuristically determine that the callee is to be blamed for the violation, based on the insight that it is less likely to repeat the same mistake many times. If, however, at least one other caller instantiates the pattern correctly, we shift the blame to the caller that does contain the

violation. Indeed, if it were instead the callee that contained the violation, any other caller would suffer the same violation as well.

However, we must still take care when blaming a certain method for a violation, since this method's other callees may be causing the anomalous behaviour instead. For instance, if we blame the caller for a violation, we determine the partial overlap in the callee to be a false positive due to a strict subpattern which cannot be detected as such because of a true violation in the caller. However, this true violation in the caller may be caused by one of the caller's other callees, unrelated to the callee that was considered before. Similarly, if we shift the blame to the callee, the violation may actually be caused by the callee's own transitively called methods. Thus, the process of blaming the method that is most likely to be causing the issue is very much an iterative process of refinement of our beliefs.

4.3.2 Filtering Algorithm

Using the insights gained from the previous subsection, we design five general rules to decide on whether a violation of a pattern in a method m is a false positive, duplicate, or likely misuse:

1. If none of the callers of m contain a violation of the same pattern, the pattern violation in m is considered a false positive. The violation is a partial instance of the pattern, and this instance is completed inter-procedurally by all of the method's callers;

2. If the callee has exactly one caller c which contains a violation of the same pattern, we heuristically decide on which of the two to blame for the violation based on the number of missing elements in their respective violations:

   (a) If both violations are missing the same number of graph elements, we decide the violation is caused by the callee m, and consider the violation in c to be a duplicate. In other words, the caller does not add pattern elements to the callee, and thus the partial pattern instance is completely present in the callee. Therefore, we heuristically determine the callee to be the cause of the violation as well;

   (b) If the caller's violation is missing less graph elements than the callee's violation, the caller is blamed for the violation, since it further completes the partial pattern instance in the callee. We assume here that the callee has a contract which the caller fails to satisfy. Additionally, from a usability point-of-view, if we report the caller as containing a misuse, and in reality the misuse resides in the callee, it is easier for a user to spot that the callee may be causing the misuse instead. However, if we report the callee as containing the misuse, and the misuse is actually caused by the caller, the user may instead classify the reported violation as a false positive, without inspecting the caller;

3. If m has multiple callers, and all of these callers contain a violation of the same pattern, we blame the callee, and consider all callers to be duplicate violations. Our reasoning is that it is less likely to repeat the same mistake in all of the callers, and it is therefore more likely that the callee is causing the violation. Note that this is a heuristic, since it may be possible that all of the callers actually do contain the same mistake;

4. If m has multiple callers, and at least one of the callers *does not* contain a violation of the same pattern, we blame all of the callers that do contain the violation, and consider m to be a false positive that cannot be discovered using Rule 1 due to this violation in the callers. If the violation was instead present in the callee m, all of its callers would contain the same violation.

Note that when we blame a certain method of a violation, we are not specifying that this method itself is the cause of the violation. The violation could instead be caused by the method's other callees, which needs to be investigated afterwards. Our rules merely state that a certain method is likely not causing the violation. If we apply these rules iteratively to eliminate candidate violations, in the end, we obtain the methods that cannot be eliminated using these rules, and thus, are actual violations.

Listing 4.3 describes our inter-procedural filtering algorithm using these five rules in pseudo-code. It takes as input the set of potential violations, which are initially all classified as undecided. It then categorises the violations as confirmed, false positive, or duplicate, in two phases: First categorising false positives according to Rule 1, then categorising duplicates according to the remaining rules. Both of these phases are implemented as fixpoint iterations, which continually refine the classification using the insights gained from previous iterations. Finally, once both phases have finished, all violations that are marked as false positives or duplicates are removed, and the filter outputs the violations that are either confirmed, or still unknown, in which case the rules are not powerful enough to reliably determine duplicates.

Removing false positives due to strict subpatterns that are extended to complete instances by its callers is implemented by the function Remove-Subpattern-False-Positives (lines 7–16). If a violating method has callers, and none of its callers contain an actual violation of the pattern, the violation is marked as a false positive (Rule 1). The function No-Callers-Violate returns true if all of the method's callers either do not contain a violation, or contain a violation that has been previously marked as a false positive. Thus, we employ fixpoint iteration to continually refine the classification if any new violations have been marked as false positives. To illustrate, consider we have three methods, a, b, and c, where a calls b and b calls c. Thus, b is inlined into a, c is inlined into both a and b. In the first iteration of the algorithm, b may get marked as a false positive, since a extends its partial pattern instance to completion. However, c may not get marked as such since, depending on the iteration order over the violations, b is still considered an actual violation, and thus, not all of c's callers are free of violations. In a second iteration, b is now marked as a false positive, and c can be determined as a false positive as well. In the third iteration, no changes to the classification are made, and the fixpoint algorithm terminates.

The second function, Remove-Duplicates, implements Rules 2–4 to remove duplicate violations. First, if we already know that at least one of the violating method's callers is confirmed to cause the violation, we can mark the violation, and any potential violations in the method's callees, as a duplicate (lines 20–21). Otherwise, we check the callers of the violating method to determine who to blame. If the violating method has exactly one caller, which necessarily contains the same violation,[5] we check for the number of missing elements in both violations, as dictated by Rule 2. If the caller extends the pattern, i.e. has less missing pattern elements than the callee, we mark the violations of the same pattern in the callee, and any of its transitive callees, as duplicates, and blame the caller (Rule 2(a), lines 23–26). Otherwise, the caller does not further extend the pattern, and the blame is fully put on the callee. Thus, the caller's

[5]If the caller did not contain the same violation, the violating method would already be filtered out previously.

```
function VIOLATION−FILTER(violations)                                          1
returns "the set of violations, with duplicates and false positives removed"   2
    violations ← REMOVE−SUBPATTERN−FALSE−POSITIVES(violations)                 3
    violations ← REMOVE−DUPLICATES(violations)                                 4
    violations ← Retain−Only−Confirmed−And−Unknown(violations)                 5
    return violations                                                          6
                                                                               7
function REMOVE−SUBPATTERN−FALSE−POSITIVES(violations)                         8
    for violation ∈ violations do:                                             9
        if (HAS−CALLERS(violation.method)                                      10
              and NO−CALLERS−VIOLATE(violation.pattern)):                      11
            MARK violation as False−Positive                                   12
                                                                               13
    if violations have changed:                                                14
        return REMOVE−SUBPATTERN−FALSE−POSITIVES(violations)                   15
    else:                                                                      16
        return violations                                                      17
                                                                               18
function REMOVE−DUPLICATES(violations)                                         19
    for violation in violations do:                                            20
        if any of SAME−VIOLATION−IN−CALLERS(violation.method) is Confirmed:    21
            MARK violation as Duplicate                                        22
            MARK calleeViolations as Duplicate                                 23
        else if HAS−ONE−CALLER(violation.method):                              24
            if CALLER−EXTENDS(violation.pattern):                              25
                MARK violation as Duplicate                                    26
                MARK calleeViolations as Duplicate                             27
                MARK callerViolation as Confirmed                              28
            else:                                                              29
                MARK callerViolation as Duplicate                              30
        else if HAS−MULTIPLE−CALLERS(violation.method):                        31
            if ALL−CALLERS−VIOLATE(violation.pattern):                         32
                MARK callerViolations as Duplicate                             33
            else if SOME−CALLERS−VIOLATE(violation.pattern):                   34
                MARK violation as Duplicate                                    35
                MARK calleeViolations as Duplicate                             36
                if all of OTHER−CALLEE−VIOLATIONS(caller) are Duplicate:       37
                    MARK callerViolation as Confirmed                          38
                else if any of OTHER−CALLEE−VIOLATIONS(caller) is Confirmed:   39
                    MARK callerViolation as Duplicate                          40
                                                                               41
        if (violation is Unknown                                               42
              and all of SAME−VIOLATION−IN−CALLEES(violation) are Duplicate):  43
            MARK violation as Confirmed                                        44
                                                                               45
    if violations have changed:                                                46
        return REMOVE−DUPLICATES(violations)                                   47
    else:                                                                      48
        return violations                                                      49
```

Listing 4.3 – Pseudo-code describing the inter-procedural filtering algorithm.

violation is marked as a duplicate (Rule 2(b), line 28). Note that we do not mark the callee as a confirmed violation, since the violation may instead by caused by one of its own transitive callees. Whether this is the case, is determined in a future iteration of the fixpoint algorithm, by having the transitive callees guide the decision.

If the violating method instead has multiple callers, and all of the callers violate the same pattern, we mark all these violations in the callers as duplicates (Rule 3, lines 30–31). As with the application of Rule 2(b), we do not immediately mark the callee's violation as confirmed, since it may still be a duplicate if it is actually caused by the callee's own transitive callees. On the other hand, if the violating method has multiple callers, but only some contain the same violation, the callee and all of the violations in

its own callees are marked as duplicates (Rule 4, line 32–34). For each of the violating callers, we check their other callees: If all of the other callees have already been marked as a duplicate, the caller's violation becomes confirmed (lines 35–36), if any of the other callees is confirmed to be the cause of the violation, the caller's violation is marked as a duplicate (lines 37–38).

Finally, if the violating method has no callers, we defer the decision to the method's own callees. Thus, the decision process is mostly guided by the callees of a method. A final check if performed that marks a violating method as a confirmed violation if none of the previous checks determine it to be otherwise, and all of the method's callees are determined to be duplicates. Since this algorithm implements a fixpoint iteration, we perform a new iteration if the classification has been refined. This way, new classifications of callers and callees can be used to classify a violation in a method.

Eventually, after the algorithm reaches a fixpoint in its classification, we end up with a set of confirmed violations, and a set of duplicate violations. We lastly remove all duplicate violations, and continue further, intra-procedural filtering on the set of confirmed violations, as well as any violation we cannot reliably classify (i.e., violations that have been classified as unknown).

4.4 Conclusion

To implement inlining and inter-procedural filtering into MUDetect, we extend its AUG representation to obtain two new representations. The extended API usage graph (eAUG) is similar to an AUG, but contains additional information to facilitate the inlining process. Such information includes, among others, the order of parameters to a call, a mapping of a method's formal parameters to the data nodes in the eAUG, a mapping of the instance fields of objects used in the method to the data nodes representing these fields, and an explicit distinction between transitive and direct order edges. The second representation is the inlined API usage graph, which is the main representation used in the tool and is used during mining and detection. An iAUG is similar to an eAUG, but embeds the eAUGs of the method's callees as well. Additionally, in an iAUG, we keep track of the method's direct and indirect callers, as well as its direct and transitive callees, for use in the inter-procedural violation filter.

Constructing inlined API usage graphs consists of first constructing extended API usage graphs according to the normal graph construction phase. Then, we perform inlining on these eAUGs, which provides us the iAUGs. This inlining involves first discovering first-party method calls in the eAUG, and substituting the call nodes for the eAUG of the callee. In this substitution, we copy over all nodes and edges of the eAUG to embed them into the iAUG under construction, except for nodes representing return statements, since inlining them may cause undesired changes in control flow represented in the iAUG. The partial iAUG obtained after injecting the nodes and edges of the callee is a disconnected graph containing the original iAUG under construction, as well as the eAUG of the callee.

We thus need to instantiate edges that connect these two disconnected subgraphs. The way these edges are instantiated is dictated by the edges connecting to and from the original method call node in the iAUG. In essence, we remove these original edges and instantiate them to connect to the eAUG's subgraph, or, in other words, we relink the edges. Each edge type that needs to be relinked needs to be handled uniquely, to account for the semantics of the edges. For example, when an order edge is relinked

from a method call node to the subgraph, this edge needs to be relinked both to the source nodes of the subgraph, as well as all nodes that are reachable in the subgraph via other order edges. This is necessary because the execution order relation is transitive. As another example, when relinking outgoing finalisation edges from the call node, we need to identify all nodes in the callee that may throw an exception, and relink the edges to have these nodes as their source.

We develop six inlining strategies, which dictate which calls should and should not be inlined. The first of these strategies performs no inlining, and is provided with the intention of comparison to MUDetect. The other five strategies differ in how they handle recursion, duplicated calls, and calls to methods that do not contain any direct API usage. Our first strategy, the inlining depth cutoff strategy, simply inlines every call it finds, but limits its exploration of the call tree to a user-supplied depth. The second implemented strategy eliminates recursion, and is aptly named the no recursion strategy. It thus prevents inlining a call node if the call is recursive, either directly or indirectly. This prevents phenomena from being duplicated in the iAUG, which may lead to overfitting during mining. Similarly, the no duplicates strategy prevents inlining a call if the eAUG corresponding to the target method has been inlined in the iAUG before. The order in which calls are inlined guarantees that the retained call is the one which is reached first by the program at runtime. We additionally implement a variation for both the no recursion and no duplicates strategy, which prevents inlining a call if the target method contains no direct usage of a targeted API. This extension is called the pre-emptive cutoff, since it stops inlining even though transitively called methods may still contain API usage. However, we assume that it is unlikely for an API usage in a caller and an API usage in an indirect callee to be related if the intermediate callee contains no API usage. Thus, this strategy eliminates phenomena that are likely unrelated to the API usage.

Using these inlined API usage graphs in the mining phase provides us inter-procedural patterns. These patterns are used in the detection phase, which directly eliminates the false positives caused by missing pattern elements that are actually present in the callee, since the callee is embedded into the iAUG of its callers. However, the detector may still report false positives due to partial pattern overlaps in a method where its callers further extend this overlap to be a complete instance of the pattern. Similarly, we run the risk of duplicate violations, as a violation present in a callee method would be reported in its callers as well, since the callee is inlined into the callers. To alleviate these issues, we implement an inter-procedural false positive and duplicate violation filter, according to heuristics.

To eliminate false positives due to a callee method missing pattern elements that are instantiated in the caller, we check all callers of the violating method. If we find that none of the callers contain the same violation, we decide that the callee is instead part of a larger, inter-procedural pattern instance, and remove it on the grounds of it being a false positive. To eliminate duplicate violations, we rely on heuristics to determine which method to blame as the cause of the violation, and prevent reporting a violation in the other methods. We design a number of rules to determine this, based on the insight that if a callee contains a violation, then all of its callers necessarily contain the same violation. Thus, in essence, when any of the callee's callers do not contain the same violation, we assume that the other callers are to be blamed for the violation. If all of the callee's callers contain the same violation, we assume the callee is causing the actual misuse, since it is unlikely to make the same mistake multiple times.

Thus, our implementation of inlining in MUDetect makes the tool able to both mine inter-procedural patterns, as well as detect inter-procedural instances and

violations of these patterns. To prevent reporting additional false positives due to these inter-procedural patterns, we filter the reported violations. Because of this, our approach should be able to effectively remove inter-procedural false positives from the detector's findings. To verify this, we perform empirical evaluation, which is the subject of the next chapter.

5 Experimental Evaluation

To evaluate the efficiency of our inlining approach and the different inlining strategies, we run the experiments present in MUBench (Amann et al., 2017) on our tool. These experiments measure our tool's effectiveness at finding real API misuses in Java projects. We additionally collect metrics on the size of the generated iAUGs, and measure the time taken to construct iAUGs, mine patterns, and detect overlaps, in order to gain insights on the effect that larger iAUGs have on these phases.

We perform four different experiments:

- Experiment RUB, to measure the recall upper bound, i.e. the recall our tool achieves under perfect training and detection conditions;
- Experiment P, to measure the precision of our tool and the different inlining strategies. Here, we take nine different Java projects, review the top-20 findings of our approach for every inlining strategy, and count the number of true positives found;
- Experiment R, to measure the recall of our approach and the six strategies, on the same nine projects as experiment P. Using previously known misuses in the MUBench dataset and any new misuses found in experiment P, we count the number of false negatives of each strategy as misuses that have not been detected by the tool;
- Experiment Perf, to measure the changes in AUG sizes and running time for the different inlining strategies and different inlining depths, and to compare the cost of inter-procedural and intra-procedural analysis in misuse detection.

We use two subsets of the dataset provided by MUBench for the first three of our experiments. Table 5.1 summarises these datasets. The full dataset of MUBench consists of 280 known misuses present in 67 Java projects, spanning 100 unique versions (Table 5.1, row "MUBench"). We discuss the chosen subsets further when describing the experimental setup of each experiment.

Table 5.2 shows the specifications of the system on which our experiments are performed. All of our experiments are run on a MacBook Pro with an Intel Core i7 CPU, clocked at 2.6GHz, and 16GB of main memory. The experiments are run through Docker, whose engine is assigned 4 CPU cores and 4GB of main memory. Note that our tool is fully sequential and thus only uses on such core. We provide each run for a single project with 2GB of Java heap space, which we consider a reasonable amount. Although modern consumer-grade computers often feature upwards of 16GB of main memory, our tool would not be the only process running on this computer and thus needs to share this memory. We allot 2 hours of running time to each, after which we

Dataset	#P (#V)	#M (#1st)	#CUs	#APIs
MUBench	67 (100)	280 (n/a)	218	n/a
Experiment RUB	31 (52)	163 (26)	162	53
Experiments P, R	9 (9)	91 (16)	n/a	23

Table 5.1 – Overview of the datasets used in the experiments. The first column (Dataset) shows the name of the dataset. The second column (#P) shows the total number of unique projects present in the dataset, along with the number of project versions in parentheses. The third column (#M) shows the number of previously identified misuses in the dataset. The parenthesised numbers in this column show how many of these misuses exercise an internal, first-party API. The fourth column (#CUs) displays the number of crafted correct usage examples corresponding to the misuses. The last column contains the total number of different APIs that are exercised in at least one of the misuses. We use "n/a" if the any metric is irrelevant for our purposes.

consider the run to have timed out. All benchmarks are run in isolation: Only one project is targeted in any single run, and only one run is performed at any one time. Between runs, the JVM is fully restarted.

In the remainder of this chapter, we dedicate a section to each different experiment: experiment RUB is considered in Section 5.1, experiment P in Section 5.2, experiment R in Section 5.3, and experiment Perf in Section 5.4. In each of these sections, we first describe the corresponding experiment's individual setup in detail, then present the results of the experiment and compare it to our baseline, and finally provide an extensive discussion. Afterwards, we list the threats to validity of our experimentation in Section 5.5, and we conclude in Section 5.6.

5.1 Recall Under Perfect Conditions

The purpose of our first experiment is to confirm that our representation of iAUGs captures sufficient details of API usages to distinguish correct usages from misuses. To do this, we simulate perfect training and detection conditions, which allows us to obtain an upper bound on the recall our tool may obtain when run unsupervised under less than perfect conditions. Thus, we can also identify cases where our tool fails to distinguish correct usages from misuses, and derive root causes for these cases.

5.1.1 Experimental Setup

This experiment is equivalent to experiment RUB performed by Amann et al. (2019), and we use a nearly identical dataset. However, due to relocation of the source code repositories of three projects,[1] the automated benchmarking pipeline cannot retrieve these projects, and their misuses have therefore been excluded from the test. Thus, we end up with a dataset containing 31 unique projects and 163 misuses, all but one of which have a corresponding correct usage (Table 5.1, row "Experiment RUB").

This experiment is run as follows: For each of the 162 correct usages in the dataset,

[1] The projects in question are iText, ArgoUML, and BattleForge

Hardware Specifications	
Model	MacBookPro13,3
CPU Model	Intel Core i7-6700HQ
Physical CPU cores	4
Base clock speed	2.6GHz
Max. TurboBoost	3.5GHz
RAM	16GB 2133MHz LPDDR3
Storage	PCIe 3.0 SSD
Software Specifications	
Docker version	18.09.2
Docker virtual cores	4
Docker engine memory	4GB
JRE version	1.8.0_171
Scala version	2.13.0

Table 5.2 – Overview of the hardware and software specifications of the system on which the evaluations are run.

we run the detector and provide it the correct usage example to train on. We set the minimum support to 1, resulting in the single correct usage to be fully picked up as a pattern. This simulates perfect training conditions, where the mined patterns are correct and exhibit the right amount of generality to distinguish misuses from correct usages. Then, we provide the detector the source code file of the Java class of which a method contains a misuse corresponding to the correct usage. We ask the detector to provide us its findings, i.e. the violations of the mined patterns that it finds in the source code, along with the context it used to detect the misuse, i.e. the missing AUG elements. We can then obtain the recall upper bound as the percentage of misuses in the dataset that are actually found by the detector.

The correct usages provided in the MUBench dataset, and by extension the dataset used in this experiment, only contain a single method and thus are intra-procedural. Therefore, running this experiment with inlining has no effect on the mined patterns. In essence, performing inter-procedural analysis in this experiment would only affect the detection phase, where its only purpose is to decrease the number of false positives due to pattern instances spanning multiple methods. Since none of the misuses in the dataset contain such usages, we decide to only consider the no inlining strategy in this experiment. Thus, we run this evaluation only on our tool with the no inlining strategy enabled, and on MUDetect as a baseline. The main difference between the two is that our tool does not remove self-usages, i.e. method calls on the same instance, and we therefore expect the recall upper bound of our approach to be slightly higher than that of MUDetect.

To summarise, for each of the 162 misuses that have a corresponding correct usage in the dataset, we train our tool without inlining using the single correct usage with a minimum support of 1. We then detect misuses in the target method containing the

Detector	#Hits	RUB
MUDETECT	112	69.1%
No inlining	120	74.0%

Table 5.3 – Results of experiment RUB for MUDETECT and our tool with the no inlining strategy. The second column shows the number of correctly reported misuses, the third column shows the corresponding upper bound of recall.

Root cause	#Misses (%)
Representation	21 (50%)
Matching	20 (48%)
Analysis	1 (2%)

Table 5.4 – Root causes of missed API misuses of our tool. The first column shows a high-level categorisation of the cause, the second column shows the corresponding number of misuses missed due to this cause, as well as its percentage in the total number of misses.

misuse, and count the number of hits that our tool finds overall. Recall upper bound is thus the percentage of these 162 misuses that our tool correctly identifies using the perfect patterns obtained from the correct usage.

5.1.2 Results

Table 5.3 shows the results of experiment RUB on MUDETECT and our approach without inlining for the previously described dataset. We can see that our approach successfully identifies eight of the misuses MUDETECT misses, increasing the recall upper bound by nearly 5%. This increase is mainly because we lift the limitation of self-usage in AUGs, which allows us to encode method calls on `this` and instance fields. Additionally, our tool finds all misuses found by MUDETECT. This confirms that our extended representation of AUGs captures sufficient details to distinguish correct usages from misuses, and captures more such details than the original AUG representation. However, MUDETECT deliberately removes such self-usages as a trade-off between precision and recall, since self-usages may lead to more false positives. We investigate the capability of our inter-procedural analysis at removing such false positives further in experiment P.

For each of the missed misuses, we additionally inspect the misuse and corresponding correct usage to identify the main reason the misuse cannot be detected. The results of this inspection are summarised in Table 5.4. Although our representation correctly distinguishes correct usages from misuses, there are still 21 cases where this representation is too abstract. Furthermore, 20 misuses are left unidentified by our tool because of improper matching. We go into more detail on these root causes in our discussion below.

5.1.3 Discussion

As can be seen in the results presented previously, the representation in our approach is in some cases still too abstract to detect certain misuses. In 19 of these 21 cases, the misuse uses an illegal literal or constants as a parameter value to a call. For example, many such misuses use DES ciphers in a method using Java's cryptography API, which is considered an unsafe practice. The correct usage instead uses AES encryption. The difference between the pattern and the misuse lies in the value given to a factory

method, which is a string literal. Since the AUG representation captures abstract types rather than concrete values for data, it considers both usages to be equivalent, and thus cannot report a violation.

Capturing such values may lead to a loss of generality in our patterns. If many legal parameter values exist, each of which is used infrequently, the resulting patterns mined from a project may fail to capture a pattern in the method call sequences because the parameter values are different for each pattern extension. This in turn may lead to a lower recall value overall, where it cannot detect certain missing method calls. One possible solution to this is to capture concrete parameter values as multisets in pattern extensions, and later perform other types of mining, such as frequent itemset mining, to determine which parameter values are legal. This could be extended further to integrate association rule mining, which could identify relations between parameter values for different calls in the same pattern. For example, if a message is encrypted with an AES cipher, the corresponding decryption should use the same cipher. Future work should investigate if such an approach pays off.

The remaining two cases where our representation is too abstract involve flipped operators in conditions. Since all relational operators are abstracted to the same label, comparisons of the form `a != null` and `a == null` are considered equivalent in our tool. The two misuses involve such mistakes in conditions, which our tool also cannot detect. This is a deliberate design choice, which allows us to focus on the presence or absence of a condition, rather than its concrete implementation, since conditions can be written in many different ways. A possible solution and an item of future work is to incorporate more intelligent analysis in these conditions, and encoding a form of path conditions into an AUG. These could for instance specify that a value is certainly not null, or certainly larger than some other value, for a specific branch in the AUG, and would result in patterns containing more semantic information that can be used in violation detection.

Another frequently occurring root cause for unidentified misuses is insufficient matching, leading to 20 false negatives. In eight of these cases, the misuse involves superfluous elements, such as an extra call to a method which should only be called once, redundant exception handling, etc. Our detection algorithm cannot detect such misuses as it specifically searches for missing elements, rather than superfluous elements. In five cases, the pattern involves only getters, which are heuristically ignored by the mining algorithm if they are determined to be semantically irrelevant. Thus, these correct usages do not lead to a pattern and can therefore not be used to detect misuses. Similarly, four cases involve a pattern of a single call, where the target misuse uses a different method. Our tool cannot detect such misuses, since it requires that the pattern and target share at least one method call. Two more cases solely involve method calls on strings, which we heuristically filter out from pattern mining as strings are determined to be too unspecific of a receiver to make up meaningful patterns, mostly since they are used extensively in different ways in each project. A final missed case is due to patterns abstracting over the concrete condition edge type, and the matching considers selection and repetition edges to be equivalent. This again allows us to focus on the presence or absence of conditions, rather than how many times a branch is executed. The misuse in question is expected to use a loop, but uses an if-then-else statement instead.

One final cause for an unidentified misuse is a limitation in our static analysis, where we assume single static assignment. When this is not the case, we may fail to detect a data relation between multiple data values that are assigned to the same variable in different branches, leading to a false negative.

Interestingly, although the dataset contains 26 misuses relating to first-party APIs, our tool still identifies misuses relating to these APIs, even though it is conceptually only able to report misuses in third-party APIs. This is because we remove all calls to first-party methods that have not been inlined, to not confuse the miner when a call is not inlined in one method, but is inlined elsewhere. However, we only remove calls of which we have an AUG corresponding to the callee's body. Since our detector is only given one method body to train with, it cannot detect any first-party calls as first-party, and thus does not remove such calls from the pattern. When detecting, such missing calls can then be reported. Performing inlining would not affect this: If the first-party call was instead inlined during pattern mining, this method's body would be completely present in the pattern. Thus, any missing call to this method can be detected since its body is not be present in the target method's iAUG.

5.2 Precision of Top-20 Findings

Our second experiment aims to assess the precision of our detector configured with the different inlining strategies, i.e. the percentage of true positives among its findings. Since the detectors may output a large number of findings which need to be inspected manually, we limit ourselves to the top-20 findings of each. For this test, we select nine projects present in MUBench as our dataset, based on previous evaluations of MUDetect. We describe this selection in the next subsection.

5.2.1 Experimental Setup

This experiment is based on experiment P performed in the MUDetect evaluation. The main difference is that we use a different subset of projects, which have been selected based on previous experimentation of MUDetect. We choose a subset of projects to decrease the number of findings we need to manually review. For this experiment, we run seven different detectors, for which we need to review at most 20 findings for each project. A quick calculation shows that this requires us to manually review at most 1.260 findings, whereas reviewing all top-20 findings for all 100 project versions in the MUBench dataset would require 14.000 manual reviews. Additionally, each run may take up to two hours for each project, thus running this experiment fully on each project version and each detector could require nearly two months of continuous runtime.

In essence, to obtain our set of target projects, we investigate the results of previous evaluations presented by Amann et al. (2019) and select projects of which we expect to see a change in behaviour when incorporating inter-procedural analysis. Thus, we select five projects based on previously-observed inter-procedural shortcomings that are present in the project, as evident in MUDetect's experiments P and RUB. The resulting projects are displayed in the first five rows of Table 5.5: We select three projects where MUDetect incorrectly reports at least one misuse which is due to the lack of inter-procedural analysis, and a further two projects where MUDetect fails to detect a misuse since it involves instance field usages. We further extend our dataset with four projects selected based on MUDetect's assessed precision and recall, to generalise our dataset and to compare our results in situations where inter-procedural issues are less prevalent. The four resulting projects are displayed in the last four rows of Table 5.5.

Project	Version	Reason	Description
Apache Lucene	1918	Self-usage FN	Search and indexing library
Closure	319	Inter-procedural FP	JavaScript compiler
ChenSun	cf23b99	Inter-procedural FP	Social network server
Jigsaw	205	Inter-procedural FP	HTTP server
TestNG	677302c	Self-usage FN	Testing framework
Asterisk-Java	304421c	Low number of findings	Telecommunications library
Commons BCEL	24014e5	Low precision	Java bytecode library
Joda-Time	cc35fb2	Low recall	Date and time library
VisualEE	410a80f	High recall	Maven plugin

Table 5.5 – Overview of the Java projects selected for experiments P and R. The project and corresponding project version (or commit identifier) as displayed in the first two columns. The third column describes the reason for the selection. The last column provides a brief description of the project.

We perform this experiment as follows: For a given detector and a given project, we provide the detector the full source code of the project as training data. Thus, the detector mines its usage patterns unsupervised: We do not validate the correctness of the acquired patterns. Then, we ask the detector to provide us its findings of potential misuses, and we keep only the first 20 findings, ranked by the detector's confidence in its findings. For each such finding, we validate the correctness by inspecting both the pattern, the detected violation, the source code, and the surrounding call context. Finally, we calculate a detector's precision as the percentage of its findings that are actual misuses, i.e. true positives.

We perform a quite strict validation of detector findings. A finding is only counted as a hit if it correctly points us to at least one program element that is involved in a likely bug in the program, and the violation correctly indicates which program elements are missing. For example, if a finding points us towards a method call which is missing proper exception handling, but the violation tells us instead that it is missing a part of a completely different pattern, we consider this a miss, since the detector does not inform us of the missing exception handling. Additionally, by inspecting the call context, we manually review that a reported misuse is not in actuality an instance of a larger, inter-procedural pattern. For example, if a detector reports that a method accesses a collection without checking its size, yet all of the callers of the method ensure that the collection is large enough, we do not count such a report as a true positive.

If a detector reports a false positive, we also inspect this finding more closely to identify the root cause of the false positive. This allows us to determine the main shortcomings of our tool, as well as certain improvements over MUDetect. We are mainly interested in false positives due to (a lack of) inter-procedural analysis, but nonetheless inspect all causes.

Preliminary testing has shown that inlining can be quite expensive, and on large projects our detector often requires more than two hours to produce results. To prevent too many timeouts from occurring, we limit the maximum inlining depth to one level. In other words, we only inline callee methods once, and do not perform any transitive

inlining. Intuitively, API usages in two different methods are more likely to be related to a larger usage if one method calls the other directly, rather than transitively through an intermediate method. To additionally decrease the number of iAUGs that need to be mined, we run our detectors in targeted API mode. Here, we pre-process all known misuses of a project to determine the misused APIs, and filter out the iAUGs that do not contain any usage of this API before the mining phase.[2] Although this may significantly decrease the number of mined patterns and, as a result, the number of findings, we observe that MUDetect's top-20 findings generally involve these same APIs, and thus, both approaches can find the same misuses.

Previous experimentation also showed that the default minimum support of 10 leads to excessive overfitting of inlined callee methods. To prevent this, we set our tool's minimum support to 15. Although this may again decrease the number of found patterns, inlining also adds more evidence for the miner to find, which may lead to patterns that capture either too little detail to distinguish correct usages from misuses, or too much detail to identify alternative but less frequent usages as correct. We return to this issue in our discussion of the results.

As a baseline for comparisons, we run the same experiment on MUDetect. We keep its default minimum pattern support value of 10, and do not run it in targeted API mode. We keep this default configuration to prevent hampering the tool's ability to detect previously-unknown misuses.

To summarise, for each of the nine projects shown in Table 5.5, we run MUDetect (the baseline) and six instances of our tool, one for each inlining strategy. We allow the tools to mine patterns and detect violations unsupervised, and review each of the top-20 findings for every combination of project and tool. We count the number of true positives by inspecting the violation and its context in the project's source code, and obtain a detector's precision as the total percentage of true positives among its total number of reviewed findings. MUDetect is configured using its default minimum support value, our tool is configured with a support value of 15 to prevent overfitting to callees. Additionally, we allow our tool to filter out iAUGs that are unrelated to the misused APIs, and perform inlining up to a maximum depth of 1, to prevent timeouts and memory issues. We present the results of this experiment in the next subsection.

5.2.2 Results

Table 5.6 shows the results of experiment P. These results are shown graphically in Figure 5.1. At first glance, our inter-procedural analysis does not improve precision at all. However, note that our main objective is to remove inter-procedural false positives from the results. Since we only consider the top-20 findings and the detectors generally provide more than 20 results, for every inter-procedural false positive we remove, a false positive caused by another reason takes its place. Note also that MUDetect is able to detect misuses in first-party APIs, while our approach removes such usages. This allows MUDetect to identify six more misuses, and ignoring these misuses, as displayed in the second row of the table, moves its precision closer to some of the inlining strategies present in our tool.

We can make some important observations in these results. We can see that our strategies produce a near-identical number of true positive findings, except for the strategy that performs no inlining, as well as the strategy that employs recursion

[2] Note that this is different from the pre-emptive cutoff, which prevents inlining eAUGs that do not contain API usage. Targeted API mode performs filtering after iAUG construction, rather than during inlining.

Detector	#Findings	#Hits	Precision
MUDetect	127	23	18.1%
MUDetect (Third-party only)	121	17	14.0%
No inlining	104	6	5.8%
Depth Cutoff	112	15	13.4%
Recursion Elimination	120	15	12.5%
Duplicate Elimination	140	14	10.0%
Recursion Elimination w/ Pre-Emptive Cutoff	136	13	9.6%
Duplicate Elimination w/ Pre-Emptive Cutoff	136	9	6.6%

Table 5.6 – Results of experiment P for MUDetect (counted with and without detected misuses of first-party APIs), and our detector with the six inlining strategies. The total number of findings for each detector is listed in the second column. The third column shows the number of true positives among these findings. The final column shows the calculated precision.

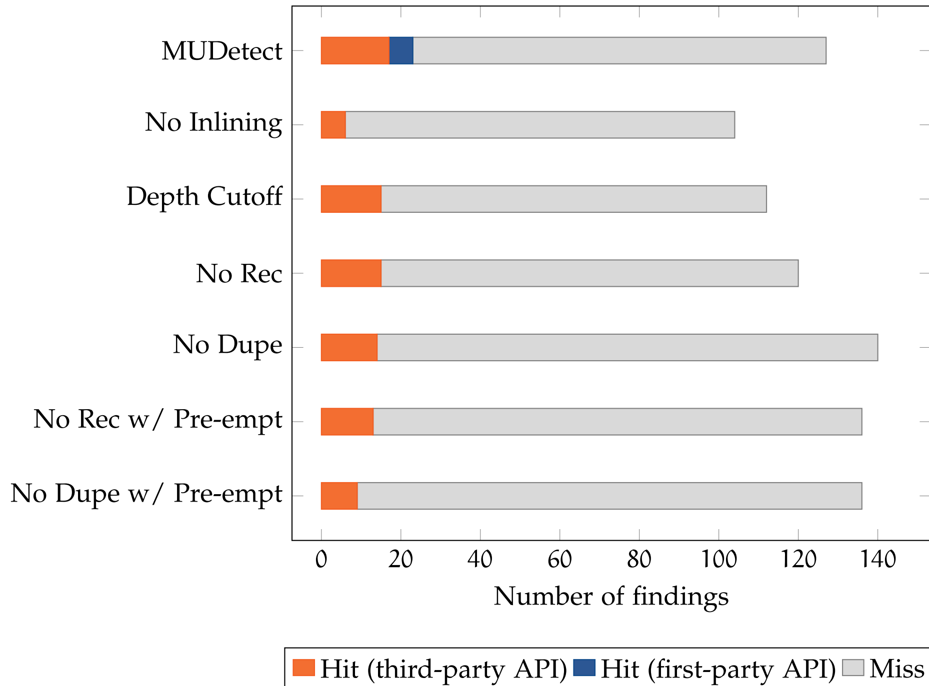


Figure 5.1 – Graphical representation of the results presented in Table 5.6, as stacked bar charts showing the number of true misuses exercising third-party and first-party APIs, and the number of false positives.

Strategy	#Hits	
	Closure	ChenSun
Duplicate Elimination	3	6
Recursion Elimination w/ Pre-Emptive Cutoff	6	2
Duplicate Elimination w/ Pre-Emptive Cutoff	3	2

Table 5.7 – Detailed view of three strategies' number of true positives on Closure and ChenSun. All detectors report 20 findings in total for both projects.

elimination, duplicate elimination, and a pre-emptive cutoff. Out of all strategies, the simple depth cutoff strategy appears to achieve the highest precision. However, it reports the same number of true positives as the recursion elimination strategy, but the latter reports more findings and thus, more false positives. We also observe that these two strategies report the exact same misuses, leading us to believe that they are largely equivalent.

Since the no inlining strategy does not inline any calls, but removes all first-party calls, it is only able to capture the simplest of intra-procedural third-party API usage patterns. Because of this, it cannot detect actual misuses, since it cannot distinguish correct usages from misuses. This observation is further supported by its low number of findings, suggesting that removing too much information from the AUG representation leads to too unspecific patterns. However, this gains us a very important insight: When inlining is enabled, our approach effectively mines inter-procedural third-party API usage patterns that allow it to detect significantly more misuses than with intra-procedural patterns.

The remaining three strategies lead to more divided situations, especially on two projects: Closure and ChenSun. We provide a more detailed view on this situation in Table 5.7. These results are interesting. For ChenSun, we can see that duplicate elimination leads to a relatively high precision, whereas a pre-emptive cutoff on the recursion elimination strategy leads to a low precision. The true positives often relate to resources that are not closed along certain execution paths. This project uses a helper method to close its resources, and thus the pattern is actually inter-procedural. However, a pre-emptive cutoff fails to inline this helper method, since it does not detect it as using the targeted API. This is because the `close` method, implemented by the resources, is actually inherited from a superclass that is declared in a different API, and AUGs resolve all calls to have the superclass as receiver to prevent issues with polymorphism. Thus, the two strategies employing a pre-emptive cutoff fail to detect the API usage in this helper method, and thus cannot construct a pattern which can identify the misuses. For Closure, duplicate elimination removes too many evidence for the miner to identify useful patterns. Thus, detection leads to many more uncommon usages that are identified as misuses, thus leading to more false positives. When we combine the two strategies, we remove too much relevant evidence in both cases, which results in more false positives.

Since one of our main goals is evaluating how inter-procedural analysis prevents false positives, we gather a list of root causes for false positives. We distinguish three root causes for false positives relating to inter-procedural analysis:

– Usages spanning multiple methods which are detected as false positives due to

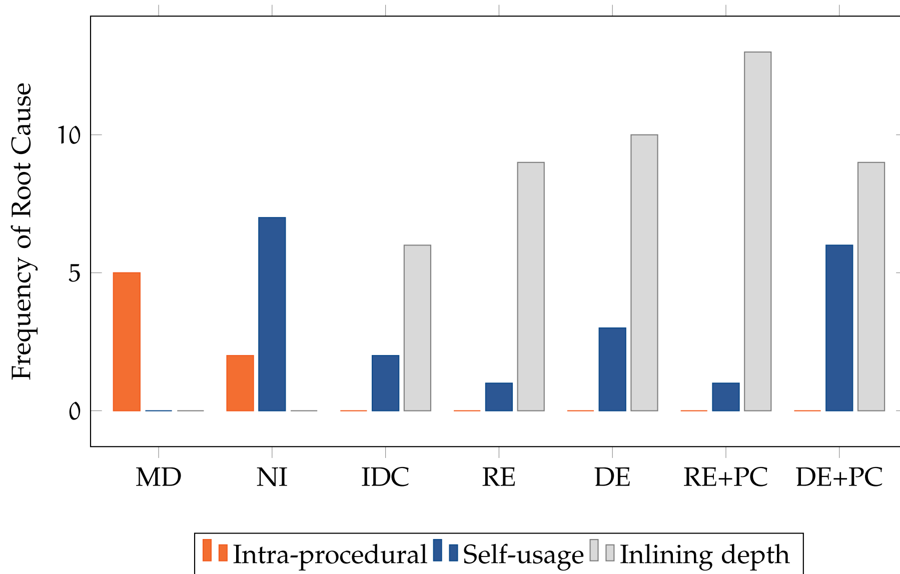


Figure 5.2 – Frequency of inter-procedural root causes of false positives, for each detector. MD: MUDetect; NI: No inlining strategy; IDC: Inlining depth cutoff; RE: Recursion elimination; DE: Duplicate elimination; RE+PC: Recursion elimination with pre-emptive cutoff; DE+PC: Duplicate elimination with pre-emptive cutoff.

intra-procedural analysis;

- Self-usages that should be removed using inter-procedural analysis by investigating the method's callers;
- Usages spanning multiple methods that cannot be detected due to the inlining depth being too shallow.

We represent the frequency of these root causes for each of the detectors graphically in Figure 5.2. The two detectors that do not use any inter-procedural analysis, i.e. the original MUDetect and our tool without inlining, both suffer from false positives caused by pattern elements being present in either transitively called methods, or callers of a method. Interestingly, our tool reports less such false positives, which we attribute to the less specific patterns that it finds. However, it finds more false positives caused by self-usages, which are not present in MUDetect's results, since it removes such usages.

We can see that our tool reports no false positives caused by a lack of inter-procedural analysis if run using inlining. However, it reports a lot more false positives due to pattern elements that are present in non-inlined methods. For example, the duplicate elimination strategies both report two misuses of a missing program element which is actually present in a duplicate call. This, together with insights gained from the general precision of the tool, leads us to believe that duplicate elimination has a negative effect on our tool's precision.

Furthermore, all of the strategies report misuses of missing program elements which are present in methods that have not been inlined. All such mistaken reports are present in the ChenSun project, where the helper method described earlier is not inlined. Although a larger inlining depth will eliminate these specific false positives, they will reappear for methods further up the call tree. Thus, we conclude that we need to extend our inter-procedural violation filtering mechanisms, where we naively

assumed inlining would be sufficient to detect inter-procedural instances of patterns. Although inlining *does* detect such instances, they are not taken into account when filtering false positives for callers.

Finally, we also notice that our inlining strategies still struggle with self-usages. In all of these cases, our detector reports a misuse in the implementation of an iterator. For example, the implementation of `hasNext` of a wrapped iterator may use the `hasNext` method on one of its fields, without calling `next`. Although our tool should be able to detect usages of this first-party `hasNext` method in other methods, and therefore inline it, in practice it fails to do so, since method calls are resolved to the superclass of the implementation. Thus, the implementation methods of a custom iterator are not inlined into any method, and we cannot use our inter-procedural analysis of callers to identify such false positives. However, we can see that false positives caused by such self-usages are a lot more prevalent when we do not perform inlining, meaning that we can in fact eliminate some of these false positives.

5.2.3 Discussion

Our experimental results identify a number of important shortcomings of our tool and its strategies. However, we still consider this experiment to be successful, as it shows that our approach correctly removes inter-procedural false positives and is able to mine method boundary-agnostic inter-procedural patterns. The experiment additionally provides us valuable insights into the root causes of the shortcomings, which allows us to devise strategies to overcome these issues in the future. We go into more detail on potential strategies later in this subsection, however we will first take a more detailed look at some of our tool's patterns and findings.

A first interesting project we take a closer look at is Joda-Time. Only four of our evaluation subjects actually produce a result on this project, the other three fail due to timeouts or memory issues. MUDetect produces no output, since it crashes during its mining phase due to garbage collection overhead, caused by inefficient memory usage. The inlining depth cutoff and recursion elimination strategies both fail to produce results within two hours, and are therefore cancelled before their run is complete. These are the only runs that failed throughout all of our experimentation. We go into more detail on this subject in our performance evaluation in Section 5.4.

However, this is not the only interesting behaviour of our detector on this project. The four other detectors that complete their run output only false positives, even though there exist over 35 misuses in this project. However, these misuses all violate the same pattern in the same way. This leads to the miner mistakenly picking up the misuse as a pattern, and the detectors actually report the single correct usage of the API as a misuse. In other words, the sheer number of misuses in this project lead to the tools overfitting towards the misuse. A potential solution to this problem of overfitting is to mine for usage patterns from multiple, unrelated projects. This could additionally solve our problem of overfitting towards inlined callee methods. Although MUDetect can be run in cross-project mode, where it mines patterns from multiple projects, we did not evaluate our tool in such a setting, since preliminary testing showed that, due to inlining, running the cross-project miner takes over 12 hours for a single API.

During the selection of projects for our dataset, we chose Asterisk-Java since it has a low number of findings. In our experimentation, MUDetect reports merely four findings, all of which are false positives. Our detector reports no findings for this project, regardless of the inlining strategy employed. It turns out our detector only

mines ten patterns, all of which are small, containing less than five nodes each. Thus, the patterns are too unspecific to detect any misuses. For VisualEE, our detectors also report zero findings, since they are only able to mine two patterns, one containing one node, and one containing two nodes. MUDetect, on the other hand, reports three findings, all of which are true positives. This difference can be attributed to our larger minimum support value, but is also caused by too strict filtering in targeted API mode. The three misuses exercise Java's `Scanner` class, and thus, we only mine for patterns in iAUGs containing usages of this class. However, `Scanner` implements Java's `Iterator` interface, and thus MUDetect, which does not perform target API filtering, can additionally use AUGs with iterator usage to mine for patterns, and can thus detect missing `hasNext` calls in the project.

Our inlining aids in detecting boundary-agnostic inter-procedural patterns. To show this, consider again the ChenSun project. ChenSun has two layers: A server package, and a database package. Many of the misuses are present in the database package, which uses Java's SQL API. Resources present in this API should always be closed by calling the `close` method. ChenSun defines a helper method to take care of this, which, in addition to calling this `close` method, performs additional exception-handling and null-checks. Through inlining, our detectors (except for the pre-emptive cutoff strategies, for reasons described earlier) manage to mine inter-procedural patterns, involving creating SQL statements, executing a CRUD operation on a database, and then closing the statement and any results. Although closing these resources is done by the helper method, the resulting patterns make no mention of a helper method, which allows the pattern to be used in different contexts.

Considering the Closure project again, we find three instances inter-procedural false positives reported by MUDetect involving an iterator. Two of these examples are illustrated in Figure 5.3. Let us first consider the example given in Figure 5.3a, which shows an inter-procedural instance of a pattern involving retrieving an iterator from a collection, checking if the iterator has a next element, and accessing this next element. The method `updateFunctionNode` retrieves an iterator, checks for the existence of an element in this iterator, and then passes the iterator to `constructAddOrStringNode` if this is the case. This latter method then accesses the iterator without checking whether an element exists, since its caller ensures that this is the case. MUDetect reports two misuses for these two methods: One for `updateFunctionNode`, since it uses an iterator but is missing a call to `next`, and one for `constructAddOrStringNode`, since it calls `next` without checking if this is safe to do. Both of these misuses are actually false positives. Furthermore, MUBench regards the latter of the two reported misuses as a true positive, even though it can never be the case that the iterator is accessed while it is empty. Our tool, with inlining enabled, reports no misuses for either method: `updateFunctionNode` is never detected as a misuse, since the transitive call to `next` is inlined into it, whereas `constructAddOrStringNode` is filtered out during inter-procedural violation filtering, since all of its callers further extend the expected pattern.

Figure 5.3b, on the other hand, shows an actual misuse. Here, the `replaceCallNode` method retrieves an iterator and immediately passes it to `constructStringExprNode`, which in turn immediately accesses the first element in this iterator. None of the functions check to see if the iterator actually has elements. This is a true inter-procedural misuse. Intuitively, we can determine that `replaceCallNode` is causing the misuse, since the other caller ensures that the iterator has an extra element before calling `constructStringExprNode`.[3] MUDetect again reports both methods as a misuse, for the same reasons as the previous example. During our experimentation, we only

[3]This other caller is actually `constructStringExprNode` itself, since it calls itself recursively.

```
class ReplaceMessages {
    private void updateFunctionNode() {
        Iterator<CharSequence> iterator = message.parts().iterator();
        Node valueNode = iterator.hasNext()
            ? constructAddOrStringNode(iterator, argListNode)
            : Node.newString("");     }

    private Node constructAddOrStringNode(Iterator<CharSequence> partsIterator,
                                          Node argListNode) {
        CharSequence part = partsIterator.next();

        if (partsIterator.hasNext()) {
            constructAddOrStringNode(partsIterator, argListNode);    } } }
```

(a) Example of an inter-procedural instance of a pattern involving an iterator, simplified from an actual usage in the Closure project. The method `updateFunctionNode` retrieves an iterator and ensures it has at least one element, after which it is passed to `constructAddOrStringNode`. The latter rightfully assumes this iterator has a next element, and retrieves it without performing a check. It later calls itself recursively with the advanced iterator if there are more elements. Intra-procedurally, both methods may be regarded as a misuse, since the former uses an iterator but appears to forget to access it, whereas the latter accesses an iterator without first checking if there is an element. However, inter-procedurally, the pattern is fully instantiated, and there is no misuse.

```
class ReplaceMessages {
    private void replaceCallNode() {
        return constructStringExprNode(message.parts().iterator(), objLitNode);    }

    private Node constructStringExprNode(Iterator<CharSequence> parts,
                                         Node argListNode) {
        CharSequence part = parts.next();

        if (parts.hasNext()) {
            constructStringExprNode(parts, objLitNode);    } } }
```

(b) Example of an inter-procedural misuse of a pattern involving an iterator, simplified from an actual misuse the Closure project. The method `replaceCallNode` retrieves an iterator and passes it to `constructStringExprNode`. The latter assumes this iterator has a next element, and retrieves it without performing a check. It later calls itself recursively with the advanced iterator if this iterator has additional elements to process. Intra-procedurally, both methods may be regarded as a misuse, since the former retrieves an iterator, but appears to make no further use of it, whereas the latter accesses an iterator without first checking if there is an element. Inter-procedurally, there exists only one misuse, since both usages are related to each other. The misuse is likely caused by `replaceCallNode`, since it does not check whether the iterator has an element before calling `constructStringExprNode`, which assumes that the iterator does. This theory is further supported by `constructStringExprNode`'s other callee (itself), since it does perform the check, which leads to no misuse.

Figure 5.3 – Two examples of an inter-procedural pattern instance and pattern violation, simplified from actual source code found in Closure's `ReplaceMessages` class.

count one of these two reports as a hit, since the other is an inter-procedural false positive. Our tool reports only one misuse, present in `replaceCallNode`, made possible by inter-procedural filtering and inlining. Thus, we conclude our tool can indeed filter out inter-procedural false positives.

However, as shown in the previous subsection, our current approach leads to extra false positives, both due to self-usages and too shallow inlining. In case of self-usages, our tool should be able to eliminate such false positives through inter-procedural filtering, but in practice often fails to do so because of limitations in our static type analysis. Consider the example shown in Figure 5.4, which is slightly adapted and heavily simplified from actual code found in the Closure project. This example shows a class, `NeighborIterator`, which exhibits two cases of self-usage in the `next` and `hasNext` methods, where it calls these same methods on an internal iterator as part of its implementation. Although our tool should be able to eliminate false positives in this usage, for example by inlining the custom iterator's `hasNext` and `next` methods in the `getNeighborNodes` method, it fails to do so. This is because it maps these two calls to the superclass `Iterator`, and thus assumes they are part of a third-party API.

```
class LinkedDirectedGraph {
    public List getNeighborNodes(Node node) {
        // ...
        for (Iterator i = node.neighborIterator(); i.hasNext(); ) {
            Node n = i.next();    } }

    static class Node {
        private Iterator neighborIterator() {
            return new NeighborIterator();    }

        private class NeighborIterator implements Iterator {
            List<Edge> edgeList;

            @Override
            public boolean hasNext() {
                return edgeList.hasNext();     }

            @Override
            public Node next() {
                return edgeList.next().source;   } } } }
```

Figure 5.4 – Example of a self-usage in a custom iterator, simplified from an actual example in Closure's `LinkedDirectedGraph` class. The `NeighborIterator` uses an internal iterator to implement its own API, and thus does not follow the same rules as users of this iterator should follow. Without proper inter-procedural checks, the usages in the iterator's methods can be reported as a misuse, since `next` calls another `next` without calling `hasNext`. Inter-procedural checks can however identify that when this internal iterator's `next` method is called, the corresponding `hasNext` method is also called through the other implemented method. This can for example be seen in the `getNeighborNodes` method.

```
class SocialNetworkBoards {
    public static String viewBoards(String username) {
        String msg = SocialNetworkDatabaseBoards.getBoardList(connection, username); }}

class SocialNetworkDatabaseBoards {
    public static String getBoardList(Connection connection, String username) {
        Statement getBoardsStmt = null;
        try { getBoardsStmt = connection.createStatement(); }
        catch (SQLException e) { /* ... */ }
        finally { DBManager.closeStatement(getBoardsStmt);   } } }

class DBManager {
    public static void closeStatement(Statement stmt) {
        if (stmt != null) {
            try { stmt.close() }
            catch (SQLException e) { /* ... */ } } } }
```

Figure 5.5 – Example of a highly nested pattern instance, where a SQL statement is created and closed through a helper method. This example is based on actual code found in the ChenSun project. The `SocialNetworkBoards` class is part of the server package, and contains a method which is used to view a list of bulletin boards. The board list is retrieved from the database through the `getBoardList` method of the `SocialNetworkDatabaseBoards` class, which resides in the database package. This method opens a SQL statement, performs a query with it (not shown), and ensures the statement is closed through a helper method `closeStatement`, present in the `DBManager` class. All of the declared methods are first-party.

A potential solution would be to adopt more advanced static analysis that can map the usages to `NeighborIterator` instead, by observing data flow relations between `getNeighborNodes` and `neighborIterator`, even though the statically declared types are `Iterator`.

The other frequent cause of inter-procedural false positives in our tool is caused by too shallow inlining depths. To illustrate, consider the example shown in Figure 5.5, which is derived from actual source code found in the ChenSun project. This example

contains three methods: `SocialNetworkBoards.viewBoards` is part of the server-side component of this project, whereas `SocialNetworkDatabaseBoards.getBoardList` and `DBManager.closeStatement` are part of the database layer. The `closeStatement` method is a helper method that performs various exception handling and condition checks to ensure that an SQL statement is always closed after a function returns. Consider we have a pattern that involves first creating a statement through `Connection.createStatement`, and finally closing this statement via `Statement.close` in a finally-block of a try-catch-finally construct. When constructing an iAUG for `getBoardList`, our detector inlines the `closeStatement` helper method and thus the `close` call is present in the iAUG. However, when constructing an iAUG for `viewBoards`, only `getBoardList` is inlined, whereas the helper method is skipped and the call is removed from the AUG since the inlining would exceed the maximum depth of 1. Thus, although our detector reports no misuse in `getBoardList`, it does report a misuse in `viewBoards`, since its iAUG indicates that it creates a statement, but never closes it.

Although increasing the maximum inlining depth would prevent this false positive from occurring, since the call to `close` would then be inlined, this is not a reliable solution. If there were instead another method, e.g. a HTTP request handler that delegates requests to specific methods, and this method calls `viewBoards`, then at an inlining depth of 2, the helper method would still not be inlined and would again lead to a false positive. Such a naive approach would only work if we allow inlining up to infinite depths, which is infeasible. A potential solution would be to instead perform inlining dynamically while detecting violations. This way, when we detect a missing graph element, we could attempt to further inline a method call and check if this callee contains the missing element. Another similar option would be to perform such checks during violation filtering. Our current filter only removes false positives of a callee if all of its callers extend the violated pattern to completion. This could be adjusted so that this check works both ways, allowing it to eliminate false positives in callers if one of its callees further extend the violated pattern.

Although these root causes for false positives provide us key insights into the shortcomings of our approach, there is one more root cause which is responsible for over 65% of the false positives in each and every one of the detectors: Uncommon usages. If there are too few correct usage examples for the miner to pick up on, it cannot create patterns that correctly distinguish misuses from correct usages. Additionally, if a pattern is overrepresented in a project, alternative but correct usages involving the same API elements may be flagged as violations of the pattern. Although cross-project mining aids in alleviating these issues, we strongly believe this is insufficient. To provide an example, consider the retrieval of an element from a Java collection. Intuitively, we know that we should always check that an element is present in a collection before attempting to retrieve it, and failing to do so may lead to uncaught exceptions and crashes. This is a source of misuses which we often observe in the dataset. However, there are many ways one can perform such checks, for instance by comparing the result of a call to the `size` method to an integer, calling `isEmpty` on the collection, or calling `hasNext` on an iterator. We observe from our assessment of false positives that all of the evaluated detectors struggle with these distinct alternatives. Often, instead of calling `hasNext` on an iterator as expected by the pattern, the programmer instead manually calls the `size` method. These alternative usages are too infrequent to be picked up as a pattern, and thus the detector mistakenly reports it as an API misuse.

However, semantically, all three of these usages may be equivalent. Rather than mining from more and more sources to find new and alternative ways to use an API correctly, we instead believe that it is more important to abstract away from how such

preconditions and usages are actually implemented. This would allow the usage model to focus on the semantics of program code, rather than the actual implementation. For example, it may represent that a certain call on a collection should only take place when the collection is not empty, regardless of how such a condition is actually implemented. However, capturing such semantic information is very much a non-trivial task.

On a final note, during this experiment, we identified eight previously unknown misuses. Two of these misuses are identified by MUDetect: In Jigsaw, it identifies a situation where a method may be called on a variable that can potentially be null, and in VisualEE, it identifies a case where a Java `Scanner` is queried for a next token without checking if there is such a token. The other six findings are identified by our tool, using one of the inlining strategies. In ChenSun, we identify two additional cases where a database resource may not be closed along exceptional paths. In Closure, one finding reports a missing emptiness check before calling `peek` and `pop` on a collection. Another finding in this same project involves a pattern where a call to `size` on a collection controls a branch that contains a call to `iterator` on this same collection. Our tool reported that two such calls are present, that the former indeed controls a branch containing the latter, but that the calls have different collections as receivers. Inspecting this further, we discovered that this is an actual bug in the program, where two nearly identical pieces of code were copied to iterate over two different collections in the same method, but the variable name for the iteration had not been changed correctly. The fifth new misuse our tool detects is a case where a `File` instance may be initialised with `null` in the Jigsaw project, which would lead to a null pointer exception. The final new misuse detected by our tool is present in the Lucene project, where the next element of an iterator is accessed without first checking if there is such an element. We use these eight newly identified misuses in our next experiment.

To summarise this discussion, we identify that our tool correctly removes inter-procedural false positives, but this leads to new false positives due to self-usages in methods and too shallow inlining depths. We theorise that more advanced static analysis could partially help in eliminating the former, whereas the latter requires more advanced filtering or detection mechanisms. However, the most prevalent cause of false positives still is uncommon usages. This could potentially be fixed by mining for usage patterns across multiple independent projects, but usage models should consider more the semantics of the program, rather than the actual implementation. Nonetheless, our detector manages to find six new misuses present in four projects thanks to inter-procedural mining and removal of inter-procedural false positives.

5.3 Recall Under Realistic Conditions

Our previous experiment provides us insights into the number of false positives our tool reports in real Java projects. Another important metric is the number of false negatives, i.e. the real misuses that our tool fails to find. To measure this, we let our evaluation subjects run unsupervised on the same nine projects of the previous experiment. We then search its results for specific misuses that are known to be present in the project, based on the MUBench dataset and the new misuses found in the previous experiment. We count how many of these misuses each test subject finds, and determine a detector's recall as the percentage of expected misuses that are correctly reported.

5.3.1 Experimental Setup

In this experiment, we reuse the dataset presented in the previous section. We additionally review all of the 91 known misuses in MUBench for these projects, and find that five of these are actually false positives, four of which are due to a lack of inter-procedural analysis when creating the dataset. All four of the inter-procedural misuses involve a missing item existence check when using an iterator, but looking at the call context of the misuse, we see that the collections in question are always guaranteed to be non-empty, either by the caller of the method or a getter used in the method's body. The last false positive suggests that a stack may be popped without an element ever being pushed, however, iteration order shows that an element always exists. Thus, we do not count these false positives towards the overall recall of a detector. However, we add the eight newly-identified misuses from the previous section to our set of expected misuses. In the end, this provides us 94 expected misuses for our evaluation subjects to find, 76 of which exercise a third-party API.

We run our evaluation subjects identically to the previous experiment, and keep the parameters. To recapitulate, we use MUDetect as our baseline, which is allowed to mine with a minimum support of 10. We run our detector with each of the six inlining strategies individually in targeted API mode, and set the minimum support to 15. We then scan each detector's results in search of the expected true misuses. In order not to manually review potentially thousands of findings, we perform a targeted search based on the name of the method in which the misuse takes place, as well as the line number of the program element involving the misuse. We review the matching findings to ensure that it points us to the correct misuse, in a similar manner as the previous experiment, and if it does, we count it as a hit. We then get the recall of a detector as the percentage of expected misuses that it actually reports. We present our results in the next subsection.

5.3.2 Results

We present the results of this experiment in Table 5.8, and graphically in Figure 5.6. We can immediately see that our approach leads to a lower recall value than the baseline. This is to be expected, since a significant portion of the expected misuses involve a first-party API, which our tool cannot detect by design. For fairness, we re-evaluate this experiment and only count the expected misuses that exercise third-party APIs. The results of this is given between parentheses in the table, and in blue in the figure. We can see here that all of our inlining strategies that actually perform inlining achieve similar recall values, which are also comparable to the baseline's. The no inlining strategy significantly falls short, since it removes too much evidence while not performing inlining, and can thus not mine many meaningful patterns. The recursion elimination strategy, both with and without a pre-emptive cutoff, achieves the highest recall out of all our strategies, which is actually equal to the recall of MUDetect on this experiment.

We can also see that adding a pre-emptive cutoff leads to no changes to the recall of a detector: Both the recursion elimination strategy as well as the duplicate elimination strategy retain the same recall value if a pre-emptive cutoff is added. However, this does not mean the two versions mine equivalent patterns, as we have seen in the results of our previous experiment that a pre-emptive cutoff leads to a decrease in precision. We notice that a pre-emptive cutoff leads to more specific patterns because of the removal of some unrelated elements that would otherwise be inlined. Although

Detector	#Hits (#Third-party)	Recall (Third-party)
MUDetect	26 (18)	27.7% (23.7%)
No inlining	10 (10)	10.6% (13.2%)
Depth Cutoff	17 (17)	18.1% (22.4%)
Recursion Elim.	18 (18)	19.1% (23.7%)
Duplicate Elim.	16 (16)	17.0% (21.0%)
Recursion Elim. w/ Pre-Emptive Cutoff	18 (18)	19.1% (23.7%)
Duplicate Elim. w/ Pre-Emptive Cutoff	16 (16)	17.0% (21.0%)

Table 5.8 – Results of experiment R for MUDetect and our detector with the six inlining strategies. The second column displays the total number of hits found by the detector, whereas the third column gives the corresponding recall. The parenthesised numbers provide these metrics where misuses of first-party APIs are not counted.

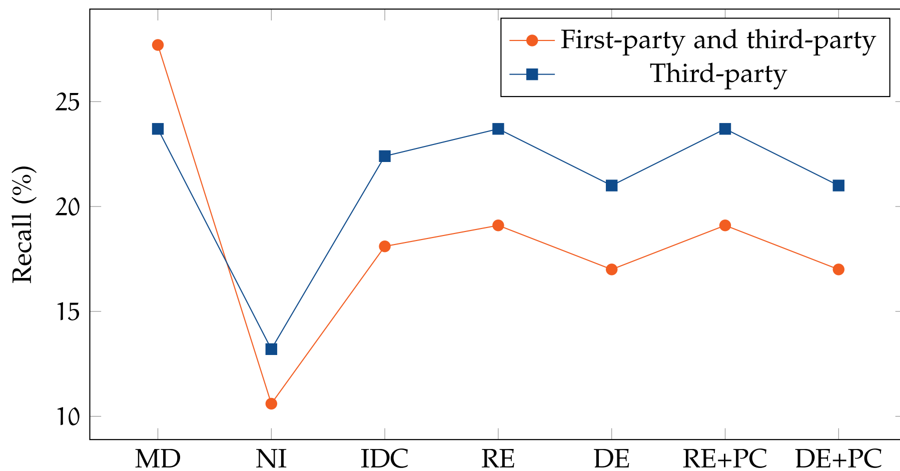


Figure 5.6 – Recall for each detector. MD: MUDetect; NI: No inlining strategy; IDC: Inlining depth cutoff; RE: Recursion elimination; DE: Duplicate elimination; RE+PC: Recursion elimination with pre-emptive cutoff; DE+PC: Duplicate elimination with pre-emptive cutoff.

such patterns focus a lot more on the API usage itself, they lead to more high-ranked false positives due to uncommon but correct usages.

MUDetect in total identifies eleven misuses which our tool fails to find. Eight of these misuses exercise a first-party API, the remaining three are all present in one project, where our tool fails to mine any meaningful pattern. However, since we achieve similar recall, this means that our tool finds other misuses which MUDetect fails to find. This varies for the different strategies, but across all strategies, our tool detects 10 true misuses that are missed by MUDetect. We attribute this to our inter-procedural boundary-agnostic mining, which can find patterns even when such patterns are not readily apparent from a single method.

5.3.3 Discussion

There are two main observations that draw our attention in this experiment. First, it is obvious that our approach struggles heavily with usages of first-party APIs, and second, we miss a significant number of misuses when we cannot mine a meaningful usage pattern for the correct usage of the API. We explore both of these shortcomings in this discussion.

The inability of our detector to find patterns in first-party API usages can be partially blamed on our targeted API filter, which filters out any iAUGs that have no usage of a targeted API before the graphs are fed to the miner. Since iAUGs contain no leftover first-party calls, the filter eliminates all iAUGs that solely make use of the targeted first-party API, and the miner can therefore not find any patterns in their usage. However, changing this filter will introduce other problems. The main shortcoming is that iAUGs capture no usage of first-party APIs, due to the inlining and method boundary erasure, which instead captures implementation details of the API. This is a deliberate design decision. Capturing the implementation of the first-party API allows us to detect potential misuses of underlying third-party APIs that cross the boundary between the first-party client code and the first-party API. For example, a project may define an API which it uses itself, and communicates with the API through Java collections. Erasing this first-party API boundary allows us to detect misuses of the underlying collections between the client and the API. Essentially, there are two sides to consider in terms of first-party API usage. On the one hand, we want to detect misuses of other, underlying APIs, as illustrated above. On the other hand, we want to detect misuses of the first-party API itself. Our approach focuses solely on the former, whereas MUDetect focuses mostly on the latter.

Combining these two approaches of handling first-party APIs is non-trivial. One potential solution is to introduce a dual representation, one which captures both usages of underlying APIs that cross boundaries between client code and first-party APIs, as well as usages of the first-party API itself. This could be done by re-introducing method boundaries into our representation. However, this should only be done in a controlled fashion: Introducing method boundaries to usages of normal helper methods may significantly decrease the generality of the patterns. Thus, a potential alternative approach is to inline all first-party calls, and erase the method boundaries for all such calls except when the call forms part of a first-party API. Unfortunately, it is unclear how such first-party APIs can be reliably identified, other than asking the user of our tool to specify the project's APIs manually. Such an approach would significantly hinder the applicability of our approach in a fully automated context.

The second shortcoming that leads to many missed misuses is the inability of our miner to infer patterns from API usage if the usages are too infrequent. This leads us to two conclusions. First, specifying a hard, absolute minimum support threshold is insufficient to capture rare usages. For example, in the VisualEE project, our miner is only given 17 iAUGs to detect patterns from, while the minimum support value is 15. In the end, it only mines two patterns from these iAUGs, one of which contains only a single node, while the other contains two nodes and one edge between them. Using a relative minimum support threshold may alleviate this problem and allow us to mine less frequent patterns. There are many ways we could specify such relative thresholds, and three options come to mind. First, we could state that a usage can be considered a pattern if it is present in more than a certain percentage of the usages that exercise a certain set of API methods. Second, a relative threshold could state instead that a usage must be present in at least a certain fraction of the total number of methods. Third, we could use a relative threshold when extending patterns, and only extend a candidate

pattern if at least a certain percentage of its supporting fragments can be extended in an identical way. Future work should investigate the effect of such relative thresholds.

We further notice that MUDetect can mine reasonable patterns in the VisualEE project, and actually detects the expected misuses too. Therefore, our second conclusion is that our minimum support value is too high to mine some important patterns. We increased the minimum support value of our tool to prevent it from overfitting towards callee methods that are inlined into callers. To illustrate, assume a project contains a method that is called independently 10 times from other methods. Thus, the method will be inlined independently 10 times as well, and for a minimum support of 10, the miner would pick up the whole callee as a pattern, leading to overfitting. Such overfitting in turn leads to lower precision, and preliminary testing showed that with a minimum support of 10, our detector rarely reported a true positive in its top-20 findings. We thus increased our minimum support to 15, which alleviates much of this overfitting and increased precision. However, as we see in this experiment, it also means that some patterns cannot be detected at all. It is unclear how we can prevent such overfitting in an intra-project setting, since we still want to capture the relevant API usage in the callee if it is inlined multiple times independently. One option would be to slice the iAUG and removing irrelevant nodes and edges, but we run the risk of removing elements that are actually part of a meaningful pattern. A better option would be to perform pattern mining in a cross-project setting, where it is less likely that similar usages in multiple different projects contain the same irrelevant usage. However, as previously discussed, inlining leads to larger graphs, and in a cross-project setting, we would need to inline many more graphs. This is expensive, and makes cross-project mining currently infeasible.

Thus, to recapitulate, our main observations from this experiment are that our approach achieves a recall similar to our baseline when only considering misuses of third-party APIs, except when no inlining is performed. If we include misuses of first-party APIs, our approach performs poorly, since it cannot capture first-party API usages at all. Extending our iAUG representation to capture such usages would require re-introducing method boundaries to first-party API calls, which requires us to reliably distinguish first-party APIs from helper methods, which is difficult to do automatically. We further observe that our miner sporadically fails to infer meaningful patterns, which is due to our increased minimum pattern support. However, decreasing this support would lead to lower precision because of patterns overfitting towards specific usages, especially in inlined methods. We theorise that such overfitting could be decreased by either slicing irrelevant usages from iAUGs, which can mistakenly remove relevant usages in the pattern, or mining patterns from multiple independent projects, which is currently too expensive due to the increased size of iAUGs after inlining.

5.4 Performance Evaluation

Our final experiment aims to determine the cost of performing inter-procedural analysis. We investigate both the ability of our inlining strategies to decrease the size of iAUGs, as well as the increase in cost when inlining at deeper levels. In essence, we measure this by running our detector at varying levels of inlining for different strategies, and measure the size of the constructed iAUGs, as well as the time spent mining the graphs and detecting violations. We then compare the obtained measurements to our strategy that does not perform inlining, and investigate the results. We describe this experimental setup in the next subsection, present the results in Section 5.4.2, and discuss the implications of our results in Section 5.4.3.

5.4.1 Experimental Setup

In this experiment, we run three of our inlining strategies at maximum inlining depths 1 and 2, and record various metrics during runtime: We measure the size of the constructed iAUGs, both in number of nodes and number of edges, as well as the time spent constructing iAUGs, mining them, and detecting violations. We choose the inlining depth cutoff strategy, the recursion elimination strategy and the duplicate elimination strategy as our main evaluation subjects. We omit the strategies with a pre-emptive cutoff from this experiment, since these can only be run in targeted API mode, and targeting a specific API leads to many iAUGs being removed before mining, which makes it difficult to compare the obtained runtime measurements. We additionally perform this experiment on our detector without inlining, and the original MUDetect tool. We do not measure graph sizes for MUDetect, since it does not record such information, and we choose not to adapt the tool to include it to avoid hampering its capabilities.

We use our detector without inlining as the baseline for comparisons with the remainder of our selected strategies. The reason is that it does record graph size information, and we can thus compare inlined graph sizes to non-inlined graph sizes. Another reason is that in order to make iAUG mining feasible in terms of memory consumption, we performed some optimisations and adjustments that are not present in MUDetect, which were briefly described in Chapter 4. Because of these optimisations, comparing our inlining to MUDetect may produce biased results. However, we do use the runtime measurements obtained from MUDetect to compare to our detector without inlining enabled.

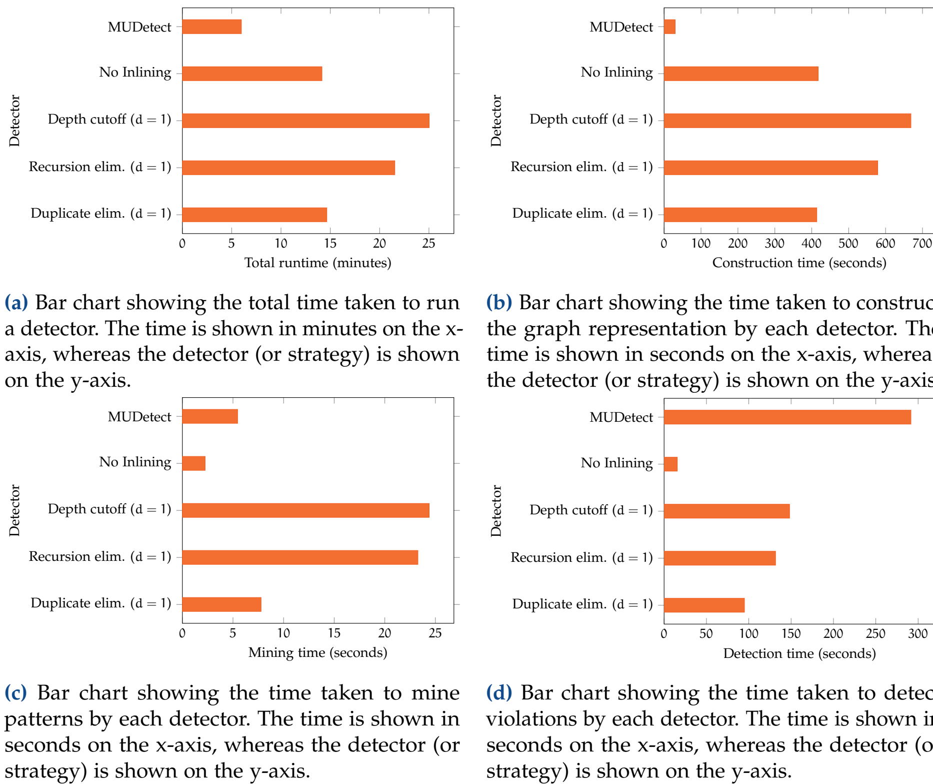
Contrary to the previous experiments, we use a larger timeout value before we cancel a detector's run. In the previous experiments, we use a timeout of two hours, whereas we use an eight-hour timeout here. The reason is that we want to gain better insights into the running time at larger inlining depths, which often take more than two hours to run. Setting too low of a timeout value would lead to too many runs being cancelled before we can get interesting measurements.

Additionally, we only run these experiments on one project, Closure version 319. This is a fairly large project, consisting of over 6000 defined methods and thus, over 6000 constructed iAUGs. We choose to limit ourselves to one project since the time spent mining and detecting depends on many different variables, including the number of mined patterns and their support. Since these are variables we cannot control, we cannot make fair comparisons between different projects.

5.4.2 Results

Let us first look at the time measurements for inlining up to a depth of one level. Figure 5.7 shows an overview of our results as bar charts. In Figure 5.7a, we display the total running time of each detector. We can clearly see from this figure that MUDetect performs its complete run quite a lot faster than our approach. We can also see already that the simple inlining depth cutoff is the slowest approach out of all strategies, whereas the duplicate elimination strategy manages to achieve a running time similar to the strategy that performs no inlining at all.

To further investigate these differences in runtime, we can inspect the running times of each individual phase. The first phase our tool performs is constructing the eAUGs and inlining them to iAUGs. The time taken to perform these steps by each detector



(a) Bar chart showing the total time taken to run a detector. The time is shown in minutes on the x-axis, whereas the detector (or strategy) is shown on the y-axis.

(b) Bar chart showing the time taken to construct the graph representation by each detector. The time is shown in seconds on the x-axis, whereas the detector (or strategy) is shown on the y-axis.

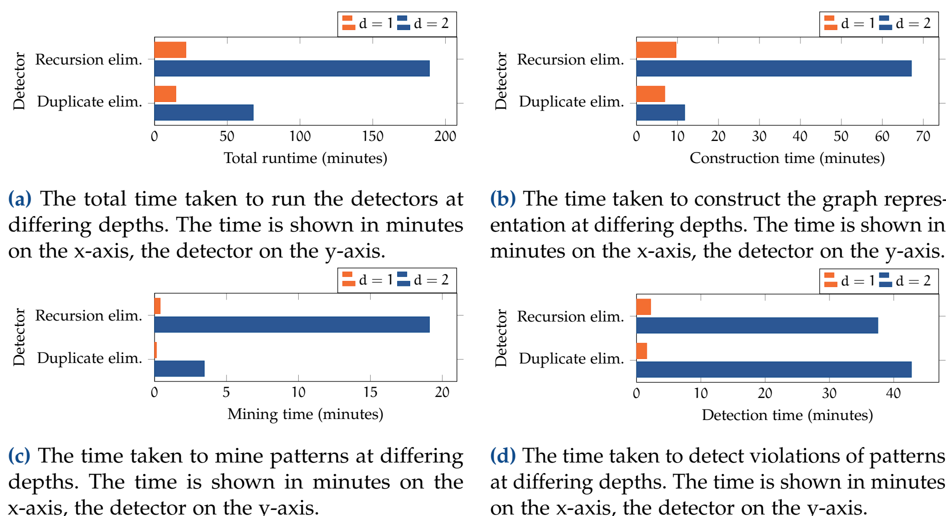
(c) Bar chart showing the time taken to mine patterns by each detector. The time is shown in seconds on the x-axis, whereas the detector (or strategy) is shown on the y-axis.

(d) Bar chart showing the time taken to detect violations by each detector. The time is shown in seconds on the x-axis, whereas the detector (or strategy) is shown on the y-axis.

Figure 5.7 – Results of runtime measurements of MUDetect and our tool with the no inlining strategy, the inlining depth cutoff strategy, the recursion elimination strategy and the duplicate elimination strategy. The latter three are configured to inline up to a maximum depth of one level, as annotated by $d = 1$.

is shown in Figure 5.7b. We can see here that the inlining process adds significant overhead to the construction of the graph representation. This is to be expected, since inlining needs to traverse entire graphs and copy over potentially many nodes and edges. Additionally, it performs analyses to determine data dependencies between callers and callees to properly relink some edge types. Interestingly, when our approach does not perform inlining, it still spends a lot of time constructing graphs. Although we expect it to instead achieve runtimes similar to MUDetect, because of the design of our inlining algorithm, it still traverses entire graphs while not inlining any method call. We theorise that this redundant traversal is causing the increase in construction time.

Although graph construction is significantly outperformed by MUDetect, the difference in time taken to mine patterns is smaller, although still significant. As we can see in Figure 5.7c, because of our data structure optimisations, our approach outperforms MUDetect in the mining phase when the no inlining strategy is selected. These memory optimisations are vital for the feasibility of the mining, since mining generates a lot of pattern extensions, which may quickly exhaust memory availability, especially when inlining is applied. Although we focused on memory optimisations, these translate to runtime performance improvements since memory allocations are expensive and immutable data structures often have beneficial performance characteristics. When we do perform inlining on the input graphs, mining becomes slower, which is to be expected since the graphs may become a lot bigger. However, from a higher perspective, we notice that these differences are not as significant as we expected, at least on an inlining depth of one. Mining takes less than half a minute in all cases,



(a) The total time taken to run the detectors at differing depths. The time is shown in minutes on the x-axis, the detector on the y-axis.

(b) The time taken to construct the graph representation at differing depths. The time is shown in minutes on the x-axis, the detector on the y-axis.

(c) The time taken to mine patterns at differing depths. The time is shown in minutes on the x-axis, the detector on the y-axis.

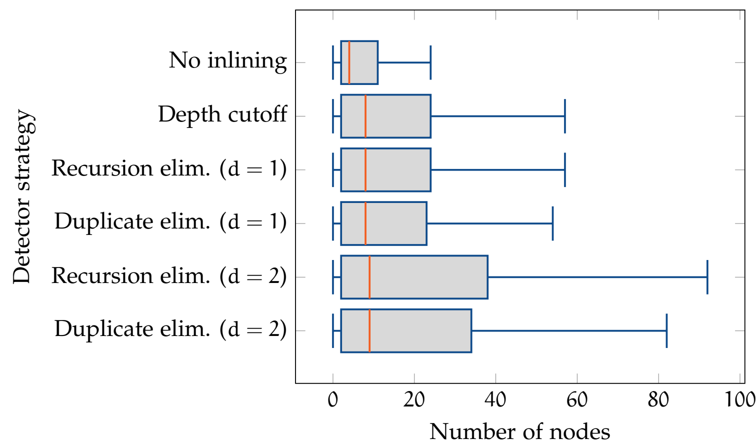
(d) The time taken to detect violations of patterns at differing depths. The time is shown in minutes on the x-axis, the detector on the y-axis.

Figure 5.8 – Bar charts showing the running time measurements for the recursion elimination strategy and duplicate elimination strategy at an inlining depth of 2 ($d = 2$), compared against the same strategy at an inlining depth of 1 ($d = 1$). Results for the inlining depth cutoff strategy are omitted, since it did not finish within 8 hours.

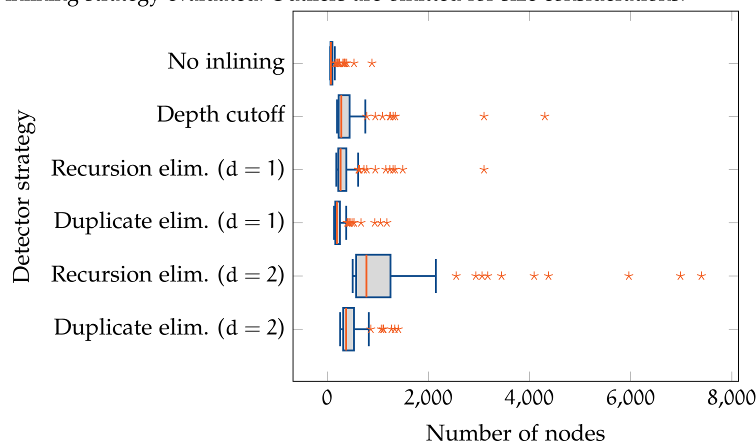
which is still quite fast, relative to the time spent constructing and detecting violations. This is because the mining algorithm uses a-priori pruning techniques, thus it can effectively eliminate many useless candidate extensions. One final interesting observation we can make from these mining times is that the duplicate elimination strategy spends significantly less time mining than the other two strategies that perform inlining. This can be attributed to its ability to eliminate a lot of calls that would otherwise be inlined, leading to less duplicate phenomena in the graphs that would be picked up by the miner. This also means that certain candidate extensions become less frequent, thus more pruning can be performed by the a-priori mining algorithm, and less candidate extensions need to be checked for isomorphism.

Although mining is still reasonably fast, a lot more time is spent on detecting misuses, as shown in Figure 5.7d. We can immediately see that some of our optimisations carry over to this phase, as the no inlining strategy significantly outperforms MUDetect in detection time. However, since this strategy also picks up on less patterns due to the removal of first-party calls, there are naturally less potential misuses to detect, and thus less work to perform during detection. Even when inlining is applied, our approach still takes less time to detect violations than MUDetect. However, it takes significantly longer than when no inlining is performed, because the graphs and patterns are larger. Again, the duplicate elimination strategy is the faster of the three strategies, while the simple inlining depth cutoff strategy is slowest, which is due to the varying degrees of elimination they perform.

Detecting misuses inter-procedurally through inlining at a depth of one level in a call tree is feasible, although significantly slower than performing detection intra-procedurally. The situation changes drastically when we increase our inlining depth only slightly. This is shown in the bar charts in Figure 5.8, which displays the runtime measurements obtained for the recursion elimination and duplicate elimination strategies if graphs are inlined up to a depth of two. We additionally display the measurements for a depth of one again, as a means of comparison. The figure does not include measurements for the inlining depth cutoff strategy, since we do not have any: This strategy did not produce any results after 8 hours.



(a) Box plots depicting the number of nodes in the constructed iAUGs for each inlining strategy evaluated. Outliers are omitted for size considerations.



(b) Box plots depicting node sizes the 50 largest graphs for each inlining strategy evaluated.

Figure 5.9 – Box plots showing summaries of the graph vertex set size measurements for each inlining strategy in our detector.

We can immediately see that all phases take significantly longer for a larger depth. In the case of constructing graphs, the situation is less severe for the duplicate elimination strategy, since it can prevent even more duplicated inlining at the second level, and thus construction does not take a lot longer, whereas the recursion elimination strategy struggles heavily because it cannot eliminate as much calls. While mining, this extra elimination pays off for the duplicate elimination strategy, which is a lot faster than the recursion elimination strategy. However, both strategies are significantly slower when graphs are inlined up to two depths, since graphs can become a lot bigger and thus, there are many more paths along which a candidate pattern can be extended. This leads to search space explosion, and more graph isomorphisms to check.

When it comes to detection of violated patterns, both strategies suffer heavily from the increase in graph sizes. Since violation detection also continuously extends candidate graph overlaps, it deals with the same search space explosion as mining does. However, mining can prune candidates that will not lead to a pattern at every step of its exploration, while detection cannot. This leads to the detection algorithm having to explore a large portion of this search space, which explains its enormous increase in running time for both strategies.

Thus, from the results provided so far, we can see that graph construction, pattern mining, and violation detection tend to take increasingly longer time as graphs grow in size. To confirm these findings, we additionally record the number of nodes present in the constructed iAUGs. The results are displayed in Figure 5.9. Figure 5.9a depicts box plots that describe the node set sizes of all graphs for a detector. We omit outliers from these graphic because of size considerations. We note that graphs clearly grow for different inlining depths, as can be seen by the wider box plots for each inlining depth. However, the median is low, suggesting that half of the graphs are in reality fairly small. Indeed, there are a lot of methods which see little growth, or even no growth at all, when inlining. This happens when methods call very little other first-party methods, such as getters and other simple methods. We also observe that the median increases only slightly when the inlining depth increases from 1 to 2. This can again be attributed getters and other simple methods: If we inline a getter method that calls no other methods at depth 1, this cannot lead to any further inlining at depth 2, since there are no transitive callees.

Thus, we can see that most graphs do grow as the inlining deepens, but the growth shown in Figure 5.9a does not fully explain the immense increase in running time we see in Figure 5.8. Thus, we turn to look at the outliers, which we previously omitted from our graphical representation. Figure 5.9b shows box plots for the 50 largest graphs for each detector strategy. We can see here that a larger inlining depth leads to more and larger outliers. For example, the recursion elimination strategy at depth 2 constructs multiple iAUGs that contain over 5000 nodes. Since the outliers are already quite large graphs at lower inlining, growing them even further leads to graphs of relatively enormous size which take a lot longer to process. Note that the outliers in the duplicate elimination strategy all contain less than 2000 nodes, for both inlining depths, because this strategy can eliminate a lot of inlining.

To summarise, our experimental results show us that, as expected, graph construction, pattern mining, and violation detection take longer to run as graphs get larger. We also find that the inlining strategies which we designed to reduce the number of repeated inlinings, help decrease the size of the graphs and thus attain more favourable running times. We show that running our tool at larger inlining depths quickly becomes infeasible, requiring over three hours to run without duplicate elimination at a two-level depth, which is due to large growths in the size of graphs and the combinatorial explosion encountered during mining and violation detection.

5.4.3 Discussion

The results of our experiment show that performing inlining to greater depths leads to a large increase in running time of our tool. Thus, it would be beneficial to devise ways to speed up the different phases. Improving the running time of graph construction could allow us to mine patterns in a cross-project setting, which is currently too expensive. Indeed, currently inlining up to a depth of one takes over five minutes for a single project, and thus, if we are to mine patterns from 20 different projects, graph construction alone could take nearly two hours. Although at an inlining depth of one level, the mining is still relatively efficient, at larger inlining depths it gets a lot slower, so further performance improvements are necessary. Similarly, and more importantly, the detection phase should be subject to optimisations as well, since it equally suffers from search space explosion, yet cannot prune as efficiently as the mining can.

We list a number of potential strategies at tackling these issues below. However, future work should first focus on determining whether inlining at larger depths pays

off in the quality of the mined patterns. Intuitively, it is more likely to have API usages continue in a callee than in a callee of a callee, i.e. a transitive callee. Thus, we are unsure whether our approach would benefit from inlining up to larger depths. Nonetheless, even at a lower depth, some phases can still benefit from optimisations and improvements.

In terms of constructing and inlining AUGs, it is worth noting that our current implementation is highly unoptimised and there are likely multiple tweaks that could be made to improve its performance. A perhaps invasive improvement would be to incorporate the inlining into the eAUG construction phase. Currently, these are split up into two phases, in part because this facilitates the reuse of eAUGs if a method is inlined into different iAUGs, without having to regenerate an eAUG every time. However, proper edge relinking could be made a lot more efficient if it was performed immediately when constructing the graph, rather than as an ad-hoc solution. The reasoning here is that to relink edges, we often need to identify data dependencies, sub-type relations, and other information that is already computed during construction, but discarded before inlining takes place. Some of this information is fairly expensive to compute, such as the data dependencies, so preventing such duplicate computations may pay off. One strategy where a lot of redundant work takes place is the no inlining strategy, which needlessly traverses graphs while it does not inline anything. This is trivial to optimise, but is not implemented since our tool was never intended to not perform inlining, and the strategy solely exists for comparison to MUDetect.

Mining is still reasonably efficient, mostly thanks to the effectiveness of a-priori subgraph mining algorithms at pruning the search space, the use of Exas feature vectors to check for graph isomorphism, and our existing optimisations to the latter. If mining were to become a problem, such as on large inlining depths or in a cross-project setting, an option could be to first cluster graphs according to the API methods used, and perform mining on each of the clusters rather than the whole set of AUGs. This technique was applied in BigGroum (Mover et al., 2018) to effectively mine patterns in 500 Android applications simultaneously. They further use program slicing to reduce the size of the graphs that are fed to the miner, which, as we see from our experiment, has an effect on the mining time. Thus, such techniques may be beneficial to reduce the time spent mining for patterns.

However, the time spent detecting violations of patterns much outweighs the time spent mining for them. This is unfortunate, since it may be interesting to apply these detections in a "mine once, detect many" situation. For example, mining could be applied once for a large number of projects, producing a number of general, high-quality patterns. These patterns could then be saved in a database, and later used to detect violations many times in either one or multiple projects, through e.g. continuous integration services. However, if the time spent detecting is too long, such automation approaches would be unattractive. One strategy to decrease the running time of detection would be to perform detection in intra-procedural graphs, which is fast, and either perform inlining dynamically, or "puzzle" the different intra-procedural overlaps back into a larger, inter-procedural overlap. The former could search for overlaps in an eAUG and if it expects a certain graph element, but finds a first-party call instead, perform a dynamic inlining to check if the callee contains the missing element. The latter would simply mark this missing element as missing, and after all overlap detection has been performed, a process similar to inlining could be performed to check that the partial overlap of a callee with a pattern completes the partial overlap of the caller with this pattern, after which the two overlaps could be merged together. As another option, note that GrouMiner does not perform a separate violation detection step, and instead considers all inextensible candidate patterns as violations if there exists a

larger pattern of which the violation is a strict subpattern. Thus, they perform violation detection directly during mining, and we could use similar techniques to bypass the slowness of detection completely. However, this would make it impossible to apply our tool in a "mine once, detect many" scenario, since detection would necessitate mining at all times. Additionally, this would introduce extra challenges to cross-project mining, and recall that GrouMiner performed quite poorly on the MUBench evaluation, which may in part be due to lack of a separate detection phase.

5.5 Threats To Validity

In this section, we list the threats to internal and external validity of our experimentation.

Internal Validity We did not fine tune the parameters of MUDetect in our baseline, such as its minimum pattern support and its ranking strategy, and instead used the configuration that is presented as optimal by Amann et al. (2019). Similarly, we did not fine tune the minimum pattern support value in our detector for each project, and instead used a hard threshold obtained from preliminary testing on different projects. It may be that a different minimum support value leads to better or worse results for a specific evaluation subject or a specific project. Future work should investigate the effects of a relative minimum support value on the precision and recall of our detector.

All detector findings were reviewed manually by the same researcher. Thus, this may introduce researcher bias into the classification of findings into hits or misses. To prevent this, we focus only on findings that constitute an actual bug in the program, and neglect code smells and anti-patterns, which are highly subjective. We decide whether an API usage contains a bug based on the MUBench dataset of confirmed misuses, the official documentation of the APIs involved, and the source code and call context surrounding the misuse. However, the MUBench dataset may in itself contain false classifications. To prevent such false classifications from impacting our results, we first reviewed all known misuses and their surrounding source code, and removed any misuse we found to be a false positive.

We did not repeat the runs in our performance experiment, and thus the reported differences in running times may have been influenced by external factors. Two such factors are JVM warmup times and overhead added by Just-In-Time (JIT) compilation. Since all of the runs take a long time to complete, we consider these factors to be negligible. Since the experiment is run in a Docker container, the resulting measurements may be influenced by overhead of its virtualisation. However, all of the runs would suffer these same overheads, making the results comparable. There may be other, unknown factors influencing the running times of the detectors, which we neglected. Nevertheless, the purpose of this experiment is to obtain an general idea of the performance differences between different strategies and inlining depths, rather than specific running time ratios.

External Validity In our experiments P and R, we specifically select five projects in which we previously observed false positives and false negatives due to a lack of inter-procedural analysis. This may not generalise to other projects. To alleviate this bias, we additionally select four projects based on MUDetect's observed precision and recall, in an attempt to obtain a more varied dataset. However, this still does not

guarantee generalisation to other projects. It is our immediate future work to extend these experiments to a larger dataset.

Similarly, the dataset of misuses employed in experiment RUB might not be representative. To alleviate this, we used the full MUBench dataset, which contains a large number of misuses found in mature Java projects. This state-of-the-art benchmark contains misuses which cover a large range of general misuse classifications, which allows us to assess our tool's capabilities in great detail.

Finally, we only perform experiment Perf for one project. Obviously, this runs the risk of not being representative. However, as mentioned previously, the purpose of this experiment is to obtain a high-level view on the potential differences in running time as a guideline, rather than statistically sound results. Additionally, since this experiment takes a long time to run, we chose to run it at a higher inlining depth for one project, rather than a lower inlining depth for multiple projects, so we can assess the increased cost of inlining depths.

5.6 Conclusion

The experiments described in this chapter provide us important insights into the capabilities of our approach. We find that our tool has a higher recall upper bound than the baseline, which is because we do not remove self-usages. However, our tool falls short on precision and recall, which is mainly due to its inability at detecting misuses in usages of first-party APIs. We additionally find that inlining strategies that eliminate less calls to inline lead to better precision and recall, since strategies that perform more elimination may incorrectly remove important phenomena from the mined patterns. Mining patterns intra-procedurally but also removing first-party method calls from AUGs leads to very poor precision and recall, as evidenced by our no inlining strategy. The other strategies, which do perform inlining, are successful at capturing inter-procedural patterns, and can therefore detect more misuses, leading to better precision and recall. We also observe that these strategies do not report violations of missing pattern elements that are present in the surrounding call context, indicating that our inter-procedural filter can effectively eliminate false positives.

However, inlining comes at a cost. Our detector takes a lot longer to run, due to the great increase in graph sizes, especially at larger inlining depths. Additionally, we suffer from a new type of false positive that arises when the inlining depth is too shallow, where the detector incorrectly reports a missing pattern element which is present in a transitively called method, but this method is not inlined since it exceeds the depth cutoff. Furthermore, we suffer from many of the same limitations as MUDetect. A majority of false positives are due to uncommon but correct API usages, which our detector flags as a violation because they deviate from a mined pattern. We also fail to identify misuses if the misuse is rooted in the concrete data value used as a parameter to a call, since the AUG representation captures abstract types rather than concrete values.

Our experimental evaluation identifies a number of important shortcomings, and we thus devise ways to alleviate these. For instance, our inter-procedural violation filtering algorithm does not consider the case where a callee contains an extension of a pattern that is absent in the caller, on the assumption that such extensions are inlined into the caller. This assumption breaks if the inlining depth is too shallow, and thus, this filter should be adapted. Alternatively, we could opt to perform inter-procedural

violation detection by dynamically inlining the graphs of the callee if a program element is missing in the caller. Our approach additionally struggles in removing some false positives related to self-usage, where an implementation uses its own API and breaks patterns that are adhered to by its clients. In many cases, such false positives are eliminated, yet this is limited if the implementation forms part of a subtype. In this case, the AUG instead encodes the base type of the receiver of the calls to the method containing the self-usage, and the actual method is never inlined. This leads to our inter-procedural analysis not being able to detect larger usages, and thus not being able to remove false positives. This problem could be alleviated by using more intelligent static type analysis.

We also identify that common causes of false positives and false negatives are due to representational issues. For example, AUGs, and by extension iAUGs, do not capture concrete values, which are often a source of misuses but cannot be detected. Additionally, due to operator abstraction, our approach cannot detect misuses related to flipped conditions. Another source of false positives is uncommon alternative usage scenarios, for example, a code snippet that directly checks the size of a collection rather than checking whether an iterator has a next element. Since our approach removes first-party method calls, it is also unable to detect misuses of a first-party API, which is another prevalent type of misuse in the dataset.

We theorise that all four of these issues can partially be alleviated by using a richer representation. For constant and literal data, we can instead capture the actual value rather than just the abstract type. However, we then need to ensure that patterns remain general. Thus, one option would be to capture such values as sets of values for each supporting pattern fragment, and then performing additional mining, such as frequent itemset mining, to determine which of the values are legal. For operators involved in a condition, we could incorporate path conditions into the graph, where condition edges are additionally labelled with the actual condition that selects the branch. Furthermore, to alleviate issues of alternative ways to specify the same behaviour, the representation should be made more abstract and focus on capturing program behaviour rather than actual implementation. For example, instead of specifying that `hasNext` must be called on an iterator before calling `next`, such a representation could specify that a collection must be checked for the existence of an element before accessing this element. The way in which this check is performed or the element is accessed does not make a difference, it is more important to ensure such a check takes place. Finally, to alleviate the issue of first-party API usages, we need to re-introduce method boundaries. However, this would directly negate the method boundary-agnostic manner in which our patterns are represented, resulting in less general patterns. Thus, future work should instead opt to use a dual representation, which captures method boundaries when the call targets a first-party API, but erases such boundaries if the called method is a helper method.

A final issue we identify is that an absolute minimum support value is often insufficient to mine for representative patterns depending on the size of the project. For example, for a small project containing little API usage, a high minimum support value will not identify any meaningful patterns. For large projects containing a lot of API usage, too low of a minimum support value may instead overfit towards a specific usage. Thus, we theorise that our approach may benefit from using a relative minimum support value, which can scale to detect meaningful patterns in API usages in projects of arbitrary sizes.

In the presence of these challenges, we still observe that our detector correctly identifies many misuses of third-party APIs. We assess that our inter-procedural pattern mining pays off, since we are able to detect previously unknown misuses,

and detect more misuses than our tool with inlining disabled. We also determine our inter-procedural filter to be beneficial, since when inlining is enabled, our tool never reports a missing program element which is present in the callers or callees of a method.

6 Conclusion

Many of the state-of-the-art API misuse detection tools report false positive findings due to a lack of inter-procedural analysis in their detection. Additionally, they may miss actual misuses due to solely mining intra-procedural patterns, leading to an increase in false negatives. Both of these limitations have a significant impact on the tool's precision and recall, rendering them difficult to use in practice.

In this dissertation, we presented our two main contributions to the field of static API misuse detection to alleviate these issues: Graph inlining, which allows mining for inter-procedural method boundary-agnostic patterns, and an inter-procedural violation filtering algorithm that effectively removes false positives that occur due to missing pattern elements that are present in the call context surrounding a method. We additionally presented six inlining strategies that customise the manner in which a method is extended to include the method bodies of its callees, differing in their handling of recursive calls, duplicate calls, and calls that contain no direct API usage.

Our inlining algorithm transforms a set of intra-procedurally generated API usage graphs into a set of inlined API usage graphs. For each such source graph, it identifies calls to first-party methods, and selects calls to inline based on the provided inlining strategy. Inlining a call consists of replacing the node that represents the call in the graph by the graph of the called method, and relinking all of the edges connecting to and from the original call node to the newly inlined graph in a code semantics aware fashion. Then, we mine such inlined API usage graphs for patterns using a-priori frequent subgraph mining, which provides us inter-procedural patterns. To allow instances of these patterns to be scattered arbitrarily among implementation methods, we erase method boundaries during inlining. Thus, the obtained patterns are general, and can be used in various contexts.

We then use these patterns to perform violation detection. We apply a pattern growth algorithm to explore a pattern and a target method to identify maximum alternative overlaps, and consider an overlap to be a violation of the pattern if it is an imperfect instance. Since patterns are inter-procedural, we may detect inter-procedural violations, which may be false positives. Thus, we apply our general inter-procedural violation filtering algorithm, which inspects a violation method's callers and callees to remove false positives and duplicate violations. Thus, in the end, we obtain a list of likely violations that may be considered API misuses.

We empirically evaluated our approach on a public dataset of known API misuses. Here, we select nine projects, five of which are known to cause inter-procedural false positives and false negatives. We find that our approach effectively mines inter-procedural patterns, allowing it to identify previously unknown misuses. We also observe that it correctly removes false positives when a missing pattern element is present in the caller or callee of a method. However, our experimental evaluation also

identifies a number of important limitations of our approach.

Most importantly, our approach is unable to identify misuses in usages of first-party APIs, since such calls are inlined rather than captured abstractly. Although our inter-procedural filtering algorithm is able to remove false positives that are present in other detectors, it fails to identify a new type of false positive, where our detector reports missing elements that it would be able to detect in an inlined API usage graph, but the call containing these elements is not present since it has not been selected for inlining. We additionally determine a common cause of false positives to be API usages that are correct, but implement their behaviour differently from the mined patterns. A final, practical limitation of our approach is its fairly high cost. Since frequent subgraph mining suffers from combinatorial explosion, detecting misuses may take a long time for large graphs. Since inlining can grow graphs significantly, this issue is more prevalent. We consider these limitations, among others, to be prime subjects of subsequent research.

6.1 Open Research Avenues

As a final section in this dissertation, we present a number of possible directions future work can take to improve upon our approach. Concretely, we identify the need for more advanced filtering to further reduce the number of false positives, including the new false positives introduced by our inlining, which we detail in Section 6.1.1. We additionally identify that many other false positives and false negatives can be considered representational issues, and urge for a more behaviour-oriented representation in Section 6.1.2. In Section 6.1.3, we explore alternatives to using an absolute minimum support threshold, to allow mining for patterns if the usage is infrequent. We consider our detector's long running times at higher inlining depths in Section 6.1.4. We additionally identify that our experiments are limited and may not generalise to other projects or datasets, and thus in Section 6.1.5, we recommend to extend this evaluation to consider a larger dataset, more evaluation subjects, and a more in-depth investigation. Finally, we explore the applicability of our approach in other fields in Section 6.1.6.

6.1.1 Extending Inter-Procedural Filtering

Our current approach still struggles with eliminating certain types of inter-procedural false positives from its findings. Concretely, there are some cases where it cannot detect self-usages as false positives, and some cases where it incorrectly reports a violation of a missing element if a method call has not been inlined.

The edge cases of self-usages that cannot be detected are caused by imprecise static type information. It often occurs that a program constructs an instance of an anonymous class, or a private inner class that extends a certain API class. Due to impreciseness in typing information, our approach cannot identify calls to these methods as the corresponding first-party method, and thus does not inline the call. This means that it cannot inspect callers during inter-procedural filtering, and cannot detect the partial usage in the self-usage to be extended by a caller.

The limitation of inlining depth leads to false positives that occur if a method is not inlined into an indirect caller. Such cases are not filtered out by our algorithm

since it does not consider the case where callees may extend a partial pattern present in the caller, since it assumes this is naturally handled by inlining. When inlining is performed at a too shallow depth, this assumption breaks, and false positives occur. To alleviate these issues, the violation filter should be adapted to consider these cases. Alternatively, a dynamic inlining technique may be used during violation detection, which would inline a method on demand if it fails to find an expected element.

6.1.2 Behaviour-Oriented Representations

We find that many false negatives are caused by the AUG representation not capturing concrete constant or literal values, or due to the abstraction over operators involved in conditions. Additionally, the majority of false positives are caused by uncommon but incorrect alternative usages, many of which involve an alternative way of implementing behaviour that is equivalent to the pattern. Finally, since we do not capture calls to first-party APIs in the iAUG representation, we are unable to detect a number of misuses. We therefore believe future work should extend the AUG representation to consider more such factors.

Capturing concrete values could potentially lead to multiple distinct patterns that only differ in their data values. Thus, we should instead retain the abstract type representation already present, but additionally capture the concrete values separately. For instance, the values could be captured as itemsets, and legal values or combinations of values could be mined using frequent itemset mining or association rule mining, respectively. This would allow a detector to identify misuses that are rooted in the argument values to a call.

In terms of operators, we find that the abstraction of relational and arithmetic operators leads to a number of false negatives whose expected misuses involve flipped conditions. We theorise that using a sort of path conditions in the labels of condition edges could help identify such misuses, by distinguishing which branch corresponds to which outcome of the condition.

For uncommon, but correct usages, we believe that future representations should abstract over the concrete implementation of program behaviour, and instead focus on this behaviour itself. In essence, we believe that a pattern stating that a collection must always be submitted to a size check before an element is retrieved, is more meaningful, general, and useful than a pattern stating that an iterator must always be verified to have a next element before this element is retrieved. The former statement would generalize to other collections, and to different ways of performing this check, whereas the latter requires specifically the use of iterator methods, and could lead to false positives in case of infrequent alternative usages.

Finally, to allow inlined AUGs to capture first-party method calls that form part of an API, the method boundaries surrounding this API call should be re-introduced into the representation. This leads to the difficult task of determining whether a method is part of an API, or a helper method. However, a dual representation containing both first-party API calls with method boundaries, as well as the inlined implementations of these calls, would allow us to detect violations of both the first-party API, as well as any underlying APIs.

6.1.3 Relative Support Threshold

We find that in some projects, there is too little API usage for our tool to mine a respective pattern. This is largely due to a too high minimum support value, which prevents the pattern from being mined. To alleviate this, future work should look into the adoption of a relative minimum support threshold that can scale to the size of the project at hand and the number of API usages in this project. One potential implementation of such a relative threshold could influence the miner to only extend candidate patterns if at least $x\%$ of its supporting fragments can be extended in an identical way. Not only would this then allow the tool to mine patterns when there are too few correct usage examples, it could also identify infrequent alternative patterns and thus reduce false positives.

6.1.4 Performance Improvements

There exist a lot of opportunities for optimisation in our tool. For instance, the construction phase, which is notoriously slow, could potentially benefit from more intelligent data structure usages and caching of information. More importantly, the mining and detection phases take an increasingly long time to perform their tasks as graphs grow. This becomes problematic when inlining at larger depths, and thus, if inlining at such larger depths is determined to be beneficial to the accuracy of mined patterns, these phases should be improved.

In terms of mining, one solution would be to cluster the graphs of methods according to the API calls made in the methods, which leads to a smaller corpus of methods being provided to the miner, and thus, less work to perform. Additionally, the size of the graphs could be decreased by using program slicing. Both of these approaches are implemented in BigGroum (Mover et al., 2018), which efficiently mined patterns from a large corpus of sources relating to a large API surface.

To improve the performance of violation detection, we could focus on detecting violations intra-procedurally, and later have the filtering algorithm remove false positives that it can detect inter-procedurally. Another option would be to perform “puzzling”, by detecting overlaps intra-procedurally and later inlining the overlaps in a similar manner to the inlining performed to construct iAUGs.

6.1.5 Extended Evaluation

We only evaluated precision and recall on nine specifically selected projects. Thus, this evaluation should be extended to include more projects to obtain more general insights into the viability of our approach. It would also be interesting to include other detectors into this evaluation, to obtain more points of comparison. A future evaluation should also determine which scenarios lead to better results for each inlining strategy, and investigate if and how they could be merged to retain the strengths of multiple strategies.

6.1.6 General Applicability

The contributions presented in this dissertation are general. Thus, it would be interesting to apply them in other contexts. For example, inter-procedural method boundary-agnostic API usage graph patterns may be a desirable representation in API usage mining and recommendation. Similarly, our inter-procedural violation filtering algorithm may be applied to other tools that have radically different representations.

6.2 Closing Statement

In this dissertation, we have presented two main contributions. Our inter-procedural graph inlining algorithm, which embeds the graph-based representation of a callee method in the graph of its callers in a code semantics-aware manner; and our general inter-procedural violation filtering algorithm, which is able to remove inter-procedural false positives and duplicate violations from the findings of API misuse detectors. Experimental evaluation shows that our graph inlining is effective at mining inter-procedural patterns, and that our filtering algorithm can correctly remove false positive violations relating to missing API elements that are present in the call context surrounding a method.

References

Acharya, M. & Xie, T. (2009). Mining API Error-Handling Specifications from Source Code. In M. Chechik & M. Wirsing (Eds.), *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09),* March 22–29, 2009, York, UK (pp. 370–384). Heidelberg, Berlin, Germany: Springer. doi: 10.1007/978-3-642-00593-0_25

Agrawal, R., Imieliński, T. & Swami, A. (1993). Mining Association Rules between Sets of Items in Large Databases. In P. Buneman & S. Jajodia (Eds.), *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93),* May 26–28, 1993, Washington, DC, USA (pp. 207–216). New York, NY, USA: ACM. doi: 10.1145/170035.170072

Agrawal, R. & Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. In J. B. Bocca, M. Jarke & C. Zaniolo (Eds.), *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94),* September 12–15, 1994, Santiago de Chile, Chile (pp. 487–499). San Francisco, CA, USA: Morgan Kaufmann.

Agrawal, R. & Srikant, R. (1995). Mining Sequential Patterns. In P. S. Yu & A. L. P. Chen (Eds.), *Proceedings of the 11th International Conference on Data Engineering (ICDE '95),* March 6–10, 1995, Taipei, Taiwan (pp. 3–14). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/ICDE.1995.380415

Amann, S. (2019). *MUDetect Artifact Page.* Retrieved 2019-08-21, from `http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/`

Amann, S., Nadi, S., Nguyen, H. A., Nguyen, T. N. & Mezini, M. (2016). MUBench: a Benchmark for API-Misuse Detectors. In M. Kim, R. Robbes & C. Bird (Eds.), *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16),* May 14–22, 2016, Austin, TX, USA (pp. 464–467). New York, NY, USA: ACM. doi: 10.1145/2901739.2903506

Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N. & Mezini, M. (2017). A Systematic Evaluation of API-Misuse Detectors. *Computing Research Repository, abs/1712.00242v3.*

Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N. & Mezini, M. (2019). Investigating Next-Steps in Static API-Misuse Detection. In M.-A. D. Storey, B. Adams & S. Haiduc (Eds.), *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19),* May 26–27, 2019, Montreal, QC, Canada (pp. 265–275). Los Alamitos, CA, USA: IEEE Computer Society.

Bielik, P., Raychev, V. & Vechev, M. (2016). PHOG: Probabilistic Model for Code. In M.-F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd International Conference on Machine Learning (ICML '16),* June 19–24, 2016, New York City, NY, USA (pp. 2933–2942). PMLR.

Burdick, D., Calimlim, M. & Gehrke, J. (2001). MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In D. Georgakopoulos & A. Buchmann (Eds.), *Proceedings of the 17th International Conference on Data Engineering (ICDE*

'01), April 2–6, 2001, Heidelberg, Germany (pp. 443–452). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/ICDE.2001.914857

Chang, R.-y., Podgurski, A. & Yang, J. (2007). Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In D. S. Rosenblum & S. G. Elbaum (Eds.), *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '07)*, July 9–12, 2007, London, UK (pp. 163–173). New York, NY, USA: ACM. doi: 10.1.1.122.5821

Clarke, E. M., Emerson, E. A. & Sistla, A. P. (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *8*(2), 244–263. doi: 10.1145/5397.5399

Egele, M., Brumley, D., Fratantonio, Y. & Kruegel, C. (2013). An Empirical Study of Cryptographic Misuse in Android Applications. In A.-R. Sadeghi, V. Gligor & M. Yung (Eds.), *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*, November 4–8, 2013, Berlin, Germany (pp. 73–84). New York, NY, USA: ACM. doi: 10.1145/2508859.2516693

Engler, D., Chen, D. Y., Hallem, S., Chou, A. & Chelf, B. (2001). Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In K. Marzullo & M. Satyanarayanan (Eds.), *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, October 21–24, 2001, Banff, AB, Canada (pp. 57–72). New York, NY, USA: ACM. doi: 10.1145/502034.502041

Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L. & Freisleben, B. (2012). Why Eve and Mallory love Android: an Analysis of Android SSL (In)Security. In T. Yu, G. Danezis & V. Gligor (Eds.), *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, October 16–18, 2012, Raleigh, NC, USA (pp. 50–61). New York, NY, USA: ACM. doi: 10.1145/2382196.2382205

Ferrante, J., Ottenstein, K. J. & Warren, J. D. (1987). The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *9*(3), 319–349. doi: 10.1145/24039.24041

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B. & Stata, R. (2002). Extended Static Checking for Java. In J. Knoop & L. J. Hendren (Eds.), *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, June 17–19, 2002, Berlin, Germany (pp. 234–245). New York, NY, USA: ACM. doi: 10.1145/512529.512558

Ganter, B. & Wille, R. (1997). *Formal Concept Analysis: Mathematical Foundations* (1st ed.). Heidelberg, Berlin, Germany: Springer.

Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D. & Shmatikov, V. (2012). The Most Dangerous Code in the World. In T. Yu, G. Danezis & V. Gligor (Eds.), *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, October 16–18, 2012, Raleigh, NC, USA (pp. 38–49). New York, NY, USA: ACM. doi: 10.1145/2382196.2382204

Grahne, G. & Zhu, J. (2003). Efficiently Using Prefix-trees in Mining Frequent Itemsets. In B. Goethals & M. J. Zaki (Eds.), *Proceedings of the 2003 IEEE International Conference on Data Mining Workshop on Frequent Itemset Mining Implementations (FIMI '03)*, November 19, 2003, Melbourne, FL, USA. Aachen, Germany: CEUR Workshop Proceedings.

Hovemeyer, D. & Pugh, W. (2004). Finding Bugs is Easy. *SIGPLAN Notices*, *39*(12), 92–106. doi: 10.1145/1052883.1052895

Inokuchi, A., Washio, T. & Motoda, H. (2000). An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In D. A. Zighed, H. J. Komorowski & J. M. Zytkow (Eds.), *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD '00)*, September 13–16, 2000, Lyon, France (pp. 13–23). Heidelberg, Berlin, Germany: Springer. doi: 10.1007/3-540-45372-5_2

Li, Z. & Zhou, Y. (2005). PR-Miner: Automatically Extracting Implicit Programming

Rules and Detecting Violations in Large Software Code. In M. Wermelinger & H. C. Gall (Eds.), *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05),* September 5–9, 2005, Lisbon, Portugal (pp. 306–315). New York, NY, USA: ACM. doi: 10.1145/1081706.1081755

Lindig, C. (2016). Mining Patterns and Violations Using Concept Analysis. In C. Bird, T. Menzies & T. Zimmermann (Eds.), *The art and science of analyzing software data* (pp. 17–38). Boston, MA, USA: Morgan Kaufmann. (First available as a technical report in 2007, Universität des Saarlandes, Saarbrücken, Germany) doi: 10.1016/B978-0-12-411519-4.00002-1

Livshits, B. & Zimmermann, T. (2005). DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In M. Wermelinger & H. C. Gall (Eds.), *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05),* September 5–9, 2005, Lisbon, Portugal (pp. 296–305). New York, NY, USA: ACM. doi: 10.1145/1081706.1081754

Michail, A. (1999). Data mining library reuse patterns in user-selected applications. In D. Setliff, R. J. Hall & E. Tyugu (Eds.), *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE '99),* October 12–15, 1999, Cocoa Beach, FL, USA (pp. 24–33). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1.1.40.3130

Michail, A. (2000). Data Mining Library Reuse Patterns using Generalized Association Rules. In C. Ghezzi, M. Jazayeri & A. L. Wolf (Eds.), *Proceedings of the 22nd International Conference on on Software Engineering (ICSE '00),* June 4–11, 2000, Limerick, Ireland (pp. 167–176). New York, NY, USA: ACM. doi: 10.1145/337180.337200

Monperrus, M. & Mezini, M. (2013). Detecting Missing Method Calls as Violations of the Majority Rule. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *22*(1), 7:1–7:25. doi: 10.1145/2430536.2430541

Mover, S., Sankaranarayanan, S., Olsen, R. B. P. & Chang, B.-Y. E. (2018). Mining Framework Usage Graphs from App Corpora. In R. Oliveto, M. Di Penta & D. C. Shepherd (Eds.), *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18),* March 20–23, 2018, Campobasso, Italy (pp. 277–289). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/SANER.2018.8330216

Nguyen, A. T. & Nguyen, T. N. (2015). Graph-Based Statistical Language Model for Code. In A. Bertolino, G. Canfora & S. Elbaum (Eds.), *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE '15),* May 16–24, 2015, Florence, Italy (Vol. 1, pp. 858–868). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/ICSE.2015.336

Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J. M. & Nguyen, T. N. (2009). Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In M. Chechik & M. Wirsing (Eds.), *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09),* March 22–29, 2009, York, UK (pp. 440–455). Heidelberg, Berlin, Germany: Springer. doi: 10.1007/978-3-642-00593-0_31

Nguyen, H. V., Nguyen, H. A., Nguyen, A. T. & Nguyen, T. N. (2014). Mining Interprocedural, Data-Oriented Usage Patterns in JavaScript Web Applications. In P. Jalote, L. C. Briand & A. van der Hoek (Eds.), *Proceedings of the 36th International Conference on Software Engineering (ICSE '14),* May 31–June 7, 2014, Hyderabad, India (pp. 791–802). New York, NY, USA: ACM. doi: 10.1145/2568225.2568302

Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M. & Nguyen, T. N. (2009). Graph-Based Mining of Multiple Object Usage Patterns. In H. van Vliet & V. Issarny (Eds.), *Proceedings of the 7th joint meeting of the European Software Engineering*

Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '09), August 24–28, 2009, Amsterdam, The Netherlands (pp. 383–392). New York, NY, USA: ACM. doi: 10.1145/1595696.1595767

Nguyen, T. T., Pham, H. V., Vu, P. M. & Nguyen, T. T. (2015). Recommending API Usages for Mobile Apps with Hidden Markov Model. In M. B. Cohen, L. Grunske & M. Whalen (Eds.), *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15),* November 9–13, 2015, Lincoln, NE, USA (pp. 795–800). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/ase.2015.109

Ramanathan, M. K., Grama, A. & Jagannathan, S. (2007a). Path-Sensitive Inference of Function Precedence Protocols. In J. Knight, W. Emmerich & G. Rothermel (Eds.), *Proceedings of the 29th International Conference on Software Engineering (ICSE '07),* May 20–26, 2007, Minneapolis, MN, USA (pp. 240–250). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/icse.2007.63

Ramanathan, M. K., Grama, A. & Jagannathan, S. (2007b). Static Specification Inference Using Predicate Mining. In K. S. M. Jeanne Ferrante (Ed.), *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07),* June 10–13, 2007, San Diego, CA, USA (pp. 123–134). New York, NY, USA: ACM. doi: 10.1145/1250734.1250749

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. & Anderson, T. E. (1997). Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In M. Banâtre, H. M. Levy & W. M. Waite (Eds.), *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP '97),* October 5–8, 1997, St. Malo, France (pp. 27–37). New York, NY, USA: ACM. doi: 10.1145/268998.266641

Scaffidi, C. (2006). Why Are APIs Difficult to Learn and Use? *ACM Crossroads*, *12*(4). doi: 10.1145/1144359.1144363

Sushine, J., Herbsleb, J. D. & Aldrich, J. (2015). Searching the State Space: A Qualitative Study of API Protocol Usability. In A. D. Lucia, C. Bird & R. Oliveto (Eds.), *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15),* May 18–19, 2015, Firenze, Italy (pp. 82–93). Los Alamitos, CA, USA: IEEE Computer Society.

Thummalapenta, S. & Xie, T. (2007). PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In K. Stirewalt, A. Egyed & B. Fischer (Eds.), *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07),* November 5–9, 2007, Atlanta, GA, USA (pp. 204–213). New York, NY, USA: ACM. doi: 10.1145/1321631.1321663

Thummalapenta, S. & Xie, T. (2009). Mining Exception-Handling Rules as Sequence Association Rules. In S. Fickas, J. Atlee & P. Inverardi (Eds.), *Proceedings of the 31st International Conference on Software Engineering (ICSE '09),* May 16–24, 2009, Vancouver, Canada (pp. 496–506). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/icse.2009.5070548

Thummalapenta, S. & Xie, T. (2011). Alattin: Mining Alternative Patterns for Defect Detection. *Automated Software Engineering*, *18*(3-4), 293–323. doi: 10.1007/s10515-011-0086-z

Wasylkowski, A. & Zeller, A. (2011). Mining temporal specifications from object usage. *Automated Software Engineering*, *18*(3-4), 263–292. doi: 10.1007/s10515-011-0084-1

Wasylkowski, A., Zeller, A. & Lindig, C. (2007). Detecting Object Usage Anomalies. In I. Crnkovic & A. Bertolino (Eds.), *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '07),* September 3–7, 2007, Dubrovnik, Croatia (p. 35-44). New York, NY, USA: ACM. doi: 10.1145/1287624.1287632

Weimer, W. & Necula, G. C. (2005). Mining Temporal Specifications for Error Detection. In N. Halbwachs & L. D. Zuck (Eds.), *Proceedings of the 11th International Conference*

on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05), April 4–8, 2005, Edinburgh, UK (pp. 461–476). Heidelberg, Berlin, Germany: Springer. doi: 10.1007/978-3-540-31980-1_30

Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.-C. & Su, Z. (2019). Exposing Library API Misuses via Mutation Analysis. In J. M. Atlee, T. Bultan & J. Whittle (Eds.), *Proceedings of the 41st International Conference on Software Engineering (ICSE '19),* May 25–31, 2019, Montreal, QC, Canada (pp. 866–877). Los Alamitos, CA, USA: IEEE Computer Society.

Xie, T. & Pei, J. (2006). MAPO: Mining API Usages from Open Source Repositories. In S. Diehl, H. Gall & A. E. Hassan (Eds.), *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06),* May 22–23, 2006, Shanghai, China (pp. 54–57). New York, NY, USA: ACM. doi: 10.1145/1137983.1137997

Zhong, H., Xie, T., Zhang, L., Pei, J. & Mei, H. (2009). MAPO: Mining and Recommending API Usage Patterns. In S. Drossopoulou (Ed.), *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09),* July 6–10, 2009, Genoa, Italy (pp. 318–343). Heidelberg, Berlin, Germany: Springer. doi: 10.1007/978-3-642-03013-0_15