



A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Sven Amann, M.Sc.

geboren in Darmstadt (Hessen).

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr.-Ing. Andreas Zeller
Datum der Einreichung:	20. März 2018
Datum der mündlichen Prüfung:	07. Mai 2018

Erscheinungsjahr 2018

Darmstädter Dissertationen

D17

Amann, Sven : A Systematic Approach to Benchmark and Improve Automated Static
Detection of Java-API Misuses
Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUpriints: 2018
URN: urn:nbn:de:tuda-tuprints-74222
URL: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/7422>

Tag der mündlichen Prüfung: 07.05.2018

Veröffentlicht unter CC BY-SA 4.0 International
<https://creativecommons.org/licenses/>

Preface

I love solving challenging problems. Maybe this is why I became interested in programming. The very idea of solving problems by writing down executable solution routines fascinates me. And the additional challenge of developing and maintaining high-quality solutions keeps me hooked.

During my studies, I kept coming back to practices and tools that support software quality, such as testing and code-analysis tools. Finally, in my Master's thesis, I developed a code recommender system based on implicit user feedback, to assist developers in writing high-quality code. This thesis was supervised by Marcel Bruch and Andreas Sewe from Prof. Mira Mezini's Software Technology Group (STG) at the Technische Universität Darmstadt. Marcel and Andreas introduced me to academic software-engineering research. They also introduced me to Sebastian Proksch, a PhD student at the STG, who asked me to join the research project KaVE, where we researched and developed recommender systems for software engineering. What followed were five years of continuous learning, hard work, and many many ups and downs. These years ultimately led to the thesis you hold before you. And since I did not walk this path alone, what follows is an attempt to thank all the people who accompanied me.

First, I would like to thank my supervisor, Prof. Mira Mezini, for the opportunity to do my PhD and for the liberty to pursue my own projects and ideas during this time. I highly appreciate that you trusted me to find my way, that you supported me in following this way, and that you acknowledged where it has led me.

I also thank Prof. Andreas Zeller for being the second examiner of my thesis. I am grateful for the time you spent on carefully reviewing my thesis and for your honest feedback.

Next, I want to thank Sebastian Proksch, with whom I collaborated very closely during the first half of my PhD. We shared many ups and downs during this time, and I am happy that the work we did back then ultimately contributed to your PhD thesis. I learned a great deal working with you and I am very glad to have had the opportunity.

Another person without whom I would not be where I am today is Sarah Nadi, who joined the STG as a PostDoc during my second year as a PhD student. Your guidance and example strongly influenced how I work, as well as my thoughts and beliefs about research and academia as a whole. I am deeply grateful for your advice and your reliability, even after you had long left to become a professor at the University of Alberta.

Soon after I started the work presented in this thesis, I had the privilege to get in contact with Hoan Anh Nguyen and Prof. Tien N. Nguyen, two experts in the field of recommender systems for software engineering. I am grateful that you two took the chance to work with a complete stranger, who you would only ever meet on Skype for almost a year to come. I highly appreciate all the guidance and assistance you put into

our work and that you continued to believe in me, despite all the bad luck we had, in addition to Reviewer 2.

One of the many things I can look back on is my research project Eko that was funded by the German Ministry of Education and Research (BMBF). It was a great honor to receive funding at this early career stage. It enabled me to lead a research team and back my work up with working prototypes, which we released along with the publications. The motivated people involved in this project were Dr. Sarah Nadi, the PhD student Leonid Glanz, and the undergraduate students Mattis Manfred Kämmerer, Jonas Schlitzer, Simon Weiler, Manuel Benz, and Govind Singh. We grew as a team over the course of the project and still collaborate on new topics.

In this very project, I had the chance to meet Willi Weiers and Joachim Heldmann from DHL IT Services, who mentored me over the course of the project. I am very grateful for your time and advice during that time. Our meetings provided me with ideas, confidence, and insights to tackle all obstacles on the path to a successful project, as well as this thesis.

My research would have been impossible without the people who went before me and who openly shared their work, data, and tools with me, no questions asked. These are Prof. Martin Monperrus, Prof. Michael Pradel, Andrzej Wasylkowski, and Prof. Andreas Zeller. May your example inspire many generations of researchers to come.

Over the years, I was happy to work with a number of excellent student assistants, namely Mattis Manfred Kämmerer, Jonas Schlitzer, David Albrecht, Uli Fahrner, and Andreas Bauer. Your hard work and dedication to high-quality software engineering enabled both development and maintenance of the many research prototypes that we published and contributed to over the years. You did a great service to the research community and myself, for which I am truly grateful. I would also like to include the students I had the pleasure to supervise in the last years. These are Michael Kutschke, David Dahlen, Oliver Abt, Waldemar Graf, Markus Zimmermann, Carina Oberle, Manuel Benz, Simon Weiler, Mattis Manfred Kämmerer, Govind Singh, Rossana Bermúdez De La Hoz, and Vidyashree Nanjunde Gowda.

I would also like to thank the people who have provided their help in proof reading this thesis. First and foremost there is Andreas Sewe, whose amazingly detailed and constructive feedback has—over and over again—brought my thinking to higher levels of clarity. Second, there is Ben Hermann, who I am convinced was an extraordinary salesperson in a prior life. Furthermore, I would like to thank the anonymous reviewers of all my submitted publications (including Reviewer 2). Though I sometimes disagreed with your opinions, I always came to value your criticism and did my best to consider it in my work. Like probably all PhD students, I came to loath the peer-review system at times, when a reject deprived me of so much of what little time I had. I sincerely hope that the community will find ways to continue providing insightful and constructive reviews in the face of the growing numbers of the submissions.

The past five years would not have been half as fun without my brilliant colleagues Andi Bejleri, Oliver Bracevac, Ervina Cergani, Joscha Drechsler, Michael Eichberg, Matthis Eichholz, Sebastian Erdweg, Leonid Glanz, Sylvia Grewe, Dominik Helm, Ben Hermann, Sven Keidel, Mirko Köhler, Florian Kübler, Edlira Kuci, Johannes Lerch, Ingo

Maier, Ralf Mitschke, Ragner Mogk, Patrick Müller, Sarah Nadi, Sebastian Proksch, Michael Reif, Guido Salvaneschi, Jan Sinschek, Jurgen van Ham, Manuel Weiel, Pascal Weisenburger, and Anna-Katharina Wickert.

Ultimately, my PhD would have come to a grinding halt many times without the invaluable support of Gudrun Harris. You certainly are the single most important person along the journey towards a doctoral degree at the STG. Your rigorous work ensures that we are well funded, fulfill regulations, and do not lose ourselves in paperwork. Your open ears and your humor ensure that we stay on our paths, with both feet on the ground, and free of illusions regarding our baking skills. I cannot thank you enough.

And of course, I would not have made it without the support of my girlfriend, my friends, and my family. I am deeply grateful to every single one of you, for accepting me for who I am and for having shared so much with me—the good, the bad, and the ugly.

Editorial notice: Throughout this thesis I use the term “we” and “us” to describe my work. This is meant to underline that research is always a cooperative effort and that I would have much less (if something at all) to present here, if other people had not taken the time off of their own work to review, discuss, and contribute to mine. I am deeply grateful for their effort.

Abstract

Today’s software industry relies heavily on the reuse of existing software libraries. Such libraries provide the building blocks for modern software products. Reusing them allow developers to focus on innovation, while standing on the shoulders of giants. To use libraries effectively, developers need to know the Application Programming Interfaces (APIs) through which they communicate with the libraries. This includes both the APIs’ semantics and the (implicit) usage constraints that come with them. In the face of the rapidly growing and evolving supply of software libraries, this has become a challenging task. As a result, incorrect usages of APIs, or API misuses, are a prevalent cause of software bugs, crashes, and vulnerabilities.

In reaction to this problem, researchers have proposed a multitude of developer-assistance tools. One particular class of such tools automates the detection of API misuses in software code. We call these tools API-misuse detectors. Existing misuse detectors address different aspects of API misuse. However, no attempt has been made to systematically define the problem space of API misuse and to assess the prevalence of API misuses compared to other types of bugs. This makes it impossible to judge the relevance of research on API-misuse detection. Moreover, previous empirical evaluations of misuse detectors commonly measure the detectors’ precision. However, since the studies use different datasets, it is unclear to which extent the results are comparable. It is also unclear where the detectors make trade-offs between their precision and their recall.

In this thesis, we first present a systematic analysis of the problem of API misuse. We find that API misuse causes 9.1% of all software bugs in real-world projects, including many critical issues, such as program crashes, data loss, and security vulnerabilities. Then, we survey the literature to consolidate over a decade of research on API-misuse detection and build MUBENCH, a public automated benchmark for API-misuse detectors. This enables us to conduct the first-ever qualitative and quantitative comparison of existing misuse detectors. We find that these detectors have the potential to discover many API misuses, but suffer from extremely low precision and recall in practice. Finally, we systematically design MUDETECT, a new API-misuse detector that addresses many of the problems of existing detectors. Using MUBENCH, we demonstrate that MUDETECT clearly outperforms existing detectors with respect to both precision and recall. Our results provide strong evidence that, following our systematic approach, we can develop API-misuse detectors that are fit for practical application.

Zusammenfassung

Die Wiederverwendung bestehender Softwarebibliotheken ist eine Grundpfeiler der heutigen Softwareindustrie. Solche Bibliotheken stellen Bausteine für moderne Softwareprodukte zur Verfügung. Ihre Verwendung erlaubt Softwareentwicklern sich auf innovative Aspekte der Software zu fokussieren, anstatt andauernd das Rad neu zu erfinden. Um Softwarebibliotheken effektiv einzusetzen, müssen Entwickler die Semantik und die (teilweise impliziten) Nutzungsbedingungen der Programmierschnittstellen (APIs) kennen, über die sie mit den Bibliotheken interagieren. Angesichts der hohen Geschwindigkeit mit der neue Softwarebibliotheken entstehen und bestehende Bibliotheken weiterentwickelt werden, ist dies zu einer immensen Herausforderung geworden. Aus diesem Grund sind inkorrekte Verwendungen von APIs, sogenannte API Misuses, heute weit verbreitet und verursachen Probleme wie Programmabstürze und Sicherheitslücken.

In Reaktion auf derartige Probleme haben Forscher eine Vielzahl von Assistenzwerkzeugen für Softwareentwickler vorgeschlagen. Eine Kategorie solcher Werkzeuge automatisiert die Identifikation von API Misuses in Software-Quelltext. Diese Werkzeuge werden API Misuse-Detektoren genannt. Bestehende Misuse-Detektoren adressieren unterschiedliche Aspekte von API Misuse. Es wurde jedoch bisher kein Versuch unternommen, das Problem von API Misuse systematisch zu definieren und die relative Häufigkeit von API Misuses in der Menge aller Softwarefehler zu erfassen. Daher ist es unmöglich, die Relevanz von Forschungsarbeit bzgl. API Misuse abzuschätzen. Bisherige empirische Untersuchungen von API Misuse-Detektoren haben deren Precision gemessen. Da diesen Untersuchungen jedoch stets unterschiedliche Datensätze zugrunde lagen, ist es unklar inwieweit die entsprechenden Ergebnisse vergleichbar sind. Ebenso ist unklar, inwieweit die verschiedenen Detektoren niedrigeren Recall zugunsten von höherer Precision in Kauf nehmen.

In der vorliegenden Arbeit präsentieren wir zunächst die Ergebnisse einer systematischen Analyse von API Misuse in Softwareprojekten. Wir belegen, dass API Misuse für 9.1% aller Softwarefehler verantwortlich ist. Viele dieser Fehler haben kritische Auswirkungen, wie Programmabstürze, Datenverlust, oder Sicherheitslücken. Anschließend präsentieren wir eine Literaturübersicht über mehr als 10 Jahre Forschungsarbeit an API Misuse-Detektoren und entwickeln MUBENCH, einen öffentlichen, automatisierten Benchmark für API Misuse-Detektoren. Diese Schritte ermöglichen uns den ersten qualitative und quantitative Vergleich von API Misuse-Detektoren. Wir zeigen, dass die Detektoren potentiell viele API Misuses identifizieren können, in der praktischen Anwendung aber sowohl im Hinblick auf Precision als auch im Hinblick auf Recall sehr schlecht abschneiden. Zuletzt stellen wir unseren neuen API Misuse-Detektor MUDTECT vor, der viele Probleme von anderen Detektoren gezielt vermeidet. Mithilfe von MUBENCH zeigen wir, dass MUDTECT im Vergleich zu anderen Detektoren sowohl

eine deutlich höhere Precision also auch einen deutlich höheren Recall erreicht. Unsere Ergebnisse weisen darauf hin, dass wir mit unserem systematischen Ansatz Detektoren entwickeln können, die für den praktischen Einsatz in der Softwareentwicklung geeignet sind.

Contents

Preface	3
1. Introduction	17
1.1. Problem Statement	18
1.2. Contributions of this Thesis	19
1.3. Publications	20
1.4. Structure of this Thesis	21
I. API-Misuse Detection	23
A History of API Misuse and API-Misuse Detection	25
2. API Usage and Misuse	27
2.1. API Usage Directives	27
2.2. Misuses in Bug Reports	29
2.3. Misuses at Development Time	30
2.4. Misuses Causing Vulnerabilities	31
2.5. Summary	33
2.6. Limitations	33
2.7. Related Work	34
3. A Taxonomy of API Misuses	35
3.1. Motivation	35
3.2. The Classification	36
3.2.1. Method Calls	37
3.2.2. Conditions	37
3.2.3. Iteration	38
3.2.4. Exception Handling	39
3.3. Limitations	39
4. A Survey of API-Misuse Detectors	41
4.1. Methodology	41
4.2. API-Misuse Detectors	42
4.2.1. PR-MINER	43
4.2.2. CHRONICLER	44
4.2.3. COLIBRI/ML	45
4.2.4. JADET	45

4.2.5. RGJ07	46
4.2.6. LKL08	46
4.2.7. ALATTIN	47
4.2.8. AX09	47
4.2.9. CAR-MINER	48
4.2.10. GROUMINER	48
4.2.11. OCD	49
4.2.12. DMMC	50
4.2.13. SPECHECK	50
4.2.14. RRFINDER	51
4.2.15. TIKANGA	51
4.2.16. PJAG12	52
4.2.17. PG12	53
4.2.18. DROIDASSIST	53
4.2.19. SALENTO	53
4.3. Discussion	54
4.3.1. Static Misuse Detectors	55
4.3.2. Dynamic Misuse Detectors	56
4.3.3. Coverage of the Problem Space	57
4.3.4. Evaluation	57
II. MuBench	61
A Systematic Evaluation of Static API-Misuse Detectors	63
5. Evaluation Setup	67
5.1. Subject Detectors	67
5.2. API-Misuse Dataset	68
5.3. Experiment P	70
5.4. Experiment RUB	73
5.5. Experiment R	75
5.6. Limitations	75
6. Experiment Pipeline	77
6.1. Representation	77
6.2. Benchmark Automation	78
6.3. Reproducibility and Traceability	81
6.4. Limitations	82
7. An Empirical Study of API-Misuse Detectors	83
7.1. Experiment P: Precision of the Detectors	83
7.2. Experiment RUB: Recall Upper Bound of the Detectors	88
7.3. Experiment R: Recall of the Detectors	93

7.4. Vulnerability Detection	95
7.5. User Experience	97
7.6. Call to Action	98
7.7. Threats to Validity	99
8. Extension and Further Use	101
8.1. Dataset Extensions	101
8.1.1. Misuses from a Runtime-Verification Study	102
8.1.2. Misuses from Previous Evaluations of Misuse Detectors	103
8.1.3. Java 8 Misuses	103
8.1.4. Misuse from the Original Misuse Collection	104
8.2. Comparison to Further Detectors	104
8.2.1. A Study of Using FINDBUGS for Misuse Detection	105
8.2.2. Integration of Salento	107
9. Related Work	109
III. MuDetect	113
The Next Step in Static API-Misuse Detection	115
10. Motivation	117
11. A New Detector	121
11.1. API-Usage Graphs	121
11.1.1. Usage Actions	121
11.1.2. Data Entities	123
11.1.3. Control Flow and Data Flow	123
11.1.4. Building API-Usage Graphs	124
11.2. Pattern Mining	126
11.2.1. Apriori-based Mining	126
11.2.2. Code Semantics	128
11.2.3. Greedy Exploration	128
11.3. Violation Detection	129
11.3.1. Graph Matching	129
11.3.2. Alternative-Pattern Instances	131
11.3.3. Ranking Violations	131
11.3.4. Alternative Violations	131
11.4. Ranking	132
11.5. Per-project and Cross-project Settings	133
12. Evaluation Setup	135
12.1. Detectors and Dataset	135

12.2. Experimental Setup	136
12.2.1. Experiment P	137
12.2.2. Experiment RUB	137
12.2.3. Experiment R	137
12.2.4. Experiment RNK	138
12.2.5. Experiment XP	138
13. Evaluation Results	139
13.1. Experiment RNK: Ranking Violations	139
13.2. Experiment P: Precision	139
13.3. Experiment RUB: Recall Upper Bound of the Detectors	142
13.4. Experiment R: Recall of the Detectors	143
13.5. Experiment XP: Cross-project Misuse Detection	144
13.6. Discussion	145
13.7. Reviewer Agreement	147
13.7.1. Agreement in Experiment P	147
13.7.2. Agreement in Experiment RUB	148
13.7.3. Agreement in Experiment R	149
14. Threats to Validity	151
15. Related Work	153
IV. Conclusion and Outlook	155
16. Conclusion	157
16.1. Summary of Results	157
16.2. Closing Discussion	159
17. Future Work	161
17.1. Benchmarking API-Misuse Detectors	161
17.2. Advancing API-Misuse Detection	163
Contributed Implementations and Data	167
Bibliography	169
Appendix	181
A. Six Basic Rules for Safe Usage of the Java Cryptographic Architecture APIs	183

B. BOA API Usage	185
C. BOA Cipher Usages	187

1. Introduction

Over the last few decades, software has become ubiquitous in our every day life. Every day, software unlocks new insights from the world around us and brings to life the devices and services that enrich our lives. Software.org¹ estimates that, in the US alone, the software industry contributed more than \$1.14 trillion to the total value-added Gross Domestic Product (GDP) and accounted for over 10.5 million jobs in 2016 (including indirect and induced impacts); an increase of about 6.5 percent on both scales, over the last two years.²

This enormous industry is based on an ever-growing number of software products. Its growth requires software companies to continuously speed up their development processes and reduce the time-to-market with every new product. The tremendous speed of the software economy shows, for example, in the domain of mobile apps: The Google Play Store grew by more than one app per minute, on average, between February and December 2016.³ Also, more and more of the big players, such as Facebook or Amazon, adopt Continuous Deployment techniques that allow them to release updates to their products 10, 100, or sometimes even 1000 times per day [SDG⁺16].

At the same time, today’s software systems become increasingly complex. A vital ingredient to keep up the development of such systems at market speed is the ability to reuse existing software components [Boe99, HDG⁺11, SE13], commonly referred to as software libraries. Such libraries provide the building blocks for modern software products. Reusing them allows developers to stand on the shoulders of giants while focusing on innovation, rather than reinventing the wheel. More specifically, the components provide Application Programming Interfaces (APIs) for developers to interact with. Metaphorically speaking, such APIs provide the vocabulary for developers to express their solutions to a task at hand.

As with any language, to use APIs effectively, developers need to know the available vocabulary, its semantics, and the usage constraints that come with it. In the face of the rapidly growing and evolving supply of software libraries, this has become a challenging task. A recent study shows, for example, that developers perceive understanding how to use cryptographic APIs as the biggest obstacle for using these APIs [NKMB16]. As a result, incorrect usages of APIs, or *API misuses*, are a prevalent cause of software bugs, crashes, and vulnerabilities [FHM⁺12, GIJ⁺12, MM13, EBFK13, LCWZ14,

¹An independent and non-partisan international research organization, dedicated to help policymakers and the broader public better understand the impact that software has on our lives, our economy, and our society. <https://software.org/> (checked on Oct 10, 2017)

²<https://software.org/wp-content/uploads/2017.Software.Economic.Impact.Report.pdf> (checked on Dec 05, 2017)

³<https://www.statista.com/statistics/266210/> (checked on Mar 20, 2018)

1. Introduction

SHA15, NKMB16, LHX⁺16].

Documenting the functionality and usage constraints of APIs is a natural way to help developer prevent API misuse, by providing a centralized source of information per API that they can easily check upon and refer to. Unfortunately, creating and maintaining high-quality documentation is a challenging task in itself; for many APIs, documentation is unavailable, incomplete, ambiguous, or incorrect [UR15, ZGC⁺17]. This sometimes even keeps developer from reusing a particular software component [UR15].

In reaction to these problems, researchers have proposed methods to automatically generate documentation, e.g., from source code [BW08, WPVS17] or developer communications [PADP⁺12]; and to automatically discover mistakes in documentation, such as mismatches between code and documentation [ZGC⁺17] or inconsistencies within documentation [ZS13]. However, even if all APIs had complete and correct documentation, this would not solve the problem entirely, since developers are often unaware even of documented usage constraints [DH09] and prefer informal references, such as STACKOVERFLOW,⁴ over official API documentation, even though the former contain more mistakes [ABF⁺16].

A more proactive approach to help developers work with APIs more efficiently and to prevent API misuse are a family of developer-assistance tools, often referred to as Recommender Systems for Software Engineering (RSSEs) [RWZ10, RMWZ14]. The key idea behind RSSEs is to automatically obtain information items estimated to be valuable for a software engineering task in a given context and to provide them to developers, often directly in their Integrated Development Environments (IDEs).

One category of RSSEs assists developers in using APIs correctly, for example, by automatically retrieving relevant code examples [HM05, HWM05], documentation fragments [DH09, TR16], or STACKOVERFLOW discussions [PBDP⁺14]; or by promoting likely proposals of the IDE’s code completion [BMM09, PLM15]—all based on the current editor content. Another category of RSSEs helps developers identify existing API misuses [WZL07, NNP⁺09b, MBM10, WZ11]. Such automated *API-misuse detectors* are the focus of this thesis.

1.1. Problem Statement

Over the last decade, the automated detection of API misuses has received much attention [LZ05, RGJ07b, RGJ07a, WZL07, AX09, NNP⁺09b, TX09b, TX09a, MBM10, GWZ10, WZ11, NPVN15, MCJ17]. Existing approaches address different aspects of API misuse, such as missing method calls [MM13], missing preconditions of method calls [TX09b], or wrong method-call order [WZL07]. However, to the best of our knowledge, no attempt has been made to systematically define the problem space of API misuse and to assess the prevalence of API misuses compared to other types of bugs. This makes it impossible to judge the impact of this type of bug and, consequently, the relevance of research on API-misuse detection in general.

⁴<https://stackoverflow.com/> (checked on Mar 20, 2018)

The performance of many existing misuse detectors has been demonstrated in empirical studies, which commonly measure the precision of detectors, i.e., which fraction of their findings are actual misuses. Precision is important, since developers often reject analysis tools that produce many false positives [FLL⁺02, BBC⁺10, JS13]. The studies show that the detectors’ precision varies greatly—between 5% and 100%. However, since studies use different datasets, it is unclear to which extent the results are comparable. Furthermore, it is unclear to what extent there is a trade-off between the kinds of misuse that a detector identifies and the precision that it achieves. Therefore—and because developers need to know which problems a particular detector finds and how reliable it does so—we argue that it is important to also assess the recall of misuse detectors, i.e., which fraction of all misuses they identify.

We, hence, need a systematic analysis of the problem of API misuse and a systematic assessment of the state of the art in API-misuse detection, its strengths, and its weaknesses, in order to advance misuse detectors and to make them ready for practical application.

1.2. Contributions of this Thesis

This work contributes to the area of API-misuse detection. Our systematic survey of existing misuse-detection literature and our qualitative and quantitative analysis of respective approaches provide the first-ever holistic view on the state of the art in the field. Our automated benchmark allows researchers to systematically advance API-misuse detection and to immediately compare new approaches to existing ones with little additional effort. Our pattern-mining, violation-detection, and ranking approaches advance the state of the art in misuse detection, bringing it one step closer the practical applicability.

The Problem Space of API Misuse In our first contribution, we systematically analyze the problem space of API misuse. First, we present our definition of API misuse and contrast it with other types of bugs. Then, we estimate how big a threat API misuse is and analyze how prevalently it appears as a cause of issues reported on Open Source software products. Furthermore, we demonstrate that API misuse causes many critical issues, such as program crashes, data loss, and security vulnerabilities and provide evidence that API misuse is a serious obstacle to developers in their day-to-day work. Based on these insights, we create the API MISUSE CLASSIFICATION (MUC), as both a taxonomy of API misuses and a framework to assess the conceptual capabilities of API-misuse detectors.

The State-of-the-art in API-Misuse Detection Our second contribution is a systematic literature review of existing work on API-misuse detection. We summarize existing approaches and compare them with regard to the underlying techniques and the types of API misuses they may identify. We find that both static and dynamic analysis techniques have been used to identify misuses and that all techniques focus on only a few

1. Introduction

types of API misuses. Furthermore, we provide an overview of the evaluation strategies that have been used to assess the performance of API-misuse detectors empirically, uncovering that evaluations have been focusing on the precision of detectors, neglecting their recall.

A Systematic Evaluation of Static API-Misuse Detectors In our third contribution, we created MUBENCH, the first-ever automated benchmark that enables the empirical evaluation and comparison of API-misuse detectors with regard to both their precision and recall. We carefully design MUBENCH to allow reproducible and comparable results across detectors. Then we use MUBENCH to evaluate four state-of-the-art misuse detectors JADET, GROUMINER, DMMC, and TIKANGA, uncovering their poor performance with regard to both precision and recall. Through a systematic analysis of the root causes of the detectors’ false positives and false negatives, we identify a set of common problems, which limit their performance. We published MUBENCH and our experiment results, to allow other researchers to reproduce our results and to evaluate further detectors on our benchmark and compare the respective results.

The Next Step in API-Misuse Detection In our fourth contribution, we present MUDTECT, a new API-misuse detector that addresses many of the problems that we identified with state-of-the-art detectors. We introduce a new graph-based representation of API usages designed to cover all prevalent types of misuses in MUC. We present respective pattern-mining and violation-detection algorithms that efficiently and effectively identify misuses. We use MUBENCH to demonstrate that our new detector outperforms the state of the art, bringing misuse detection one step closer to practical use. We also demonstrate that both the precision and the recall of misuse detectors greatly improves, when we provide them with cross-project usage examples for pattern mining, showing a further advance API-misuse detection.

1.3. Publications

Many of the contributions presented in this thesis have previously been published to software engineering conferences or journals. This section gives an overview over these publications and the respective parts of this thesis. The thesis parts may contain verbatim content of the publications.

MuBench: A Benchmark for API-Misuse Detectors [ANN⁺16]. The MSR’16 data paper presents an early version of the collection of Java API-Misuses that we describe in Chapter 2. This collection contained 89 Java API misuses collected by reviewing issues from general bug datasets (Section 2.2), conducting a developer survey (Section 2.3), and mining misuses from the version-control histories of Open Source project (Section 2.4). The paper also presents an early version of the data format that we developed for processing the misuse examples in our automated benchmark MUBENCH (Section 6.1).

A Systematic Evaluation of API-Misuse Detectors [ANN⁺18b]. The TSE’18 journal article presents an extension of the API-misuse dataset by misuse examples from studies on API-usage directives (Section 2.1). It introduces the API-Misuse Classification MUC (Chapter 3) and presents the 2016 version of our survey of API-misuse detectors (Chapter 4). The article also presents the automated misuse-detector benchmark MUBENCH (Chapter 5 and Chapter 6) and the empirical evaluation and comparison of the four state-of-the-art misuse detectors JADET [WZL07], GROUMINER [NNP⁺09b], DMMC [MBM10], and TIKANGA [WZ11] (Chapter 7).

MuDetect: The Next Step in API-Misuse Detection [ANN⁺18a] (*under review*). The paper presents the new API-misuse detector MUDETECT and the empirical evaluation and comparison of MUDETECT, JADET [WZL07], GROUMINER [NNP⁺09b], DMMC [MBM10], and TIKANGA [WZ11] (Part III).

1.4. Structure of this Thesis

This thesis is organized as follows. In Part I, we introduce our terminology with regard to API misuse and API-misuse detection. We estimate the potential for API misuses based on usage directives in API documentation and investigate the prevalence of API misuse in bug reports on released software products and during development time (Chapter 2). Based on the examples of API misuse we identify in this process, we propose the API-Misuse Classification (MUC), both as a taxonomy of API misuses and a framework to assess the conceptual capabilities of API-misuse detectors (Chapter 3). Then, we conduct a systematic literature survey to identify existing work on API-misuse detection and assess and compare the conceptual capabilities of respective approaches with respect to MUC (Chapter 4).

In Part II, we present MUBENCH, our automated benchmark for API-misuse detectors. We discuss how we create a ground-truth dataset of API misuses and how we design three experiments to assess the performance of such misuse detectors with respect to their precision and recall (Chapter 5). Through automating most of the evaluation process with MUBENCH, we enable reproducible and comparable results and keep the effort of manual reviews of detector findings low (Chapter 6). We use MUBENCH to assess and compare the performance of four state-of-the-art misuse detectors, analyze their findings in detail, and uncover several root causes for their false positives and false negatives, which we use to formulate a roadmap for the next steps in API-misuse detection and to call researchers to action (Chapter 7). To demonstrate the potential for future use of MUBENCH, we present examples of how we extended the benchmark dataset and of further studies that use the benchmark (Chapter 8). To conclude, we give an overview of related benchmarking datasets and approaches, as well as empirical studies comparing misuse detectors (Chapter 9).

In Part III, we present MUDETECT, our new API-misuse detector that advances the state of the art by solving many of the challenges identified in the previous part of this thesis. We first give a high-level overview on the problems MUDETECT addresses

1. Introduction

and how it does so (Chapter 10). Then we present our detector’s pattern-mining and violation-detection algorithms and its ranking strategy in detail (Chapter 11). Subsequently, we describe the evaluation setup we use to assess MUDetect’s performance using MUBENCH (Chapter 12), evaluate the detector and compare it to the other detectors evaluated in the previous part of this thesis (Chapter 13). To conclude, we discuss the threats to the validity of our findings (Chapter 14) and give an overview of related work (Chapter 15).

In Part IV, we summarize this thesis and present an outlook to further future work on API-misuse detection.

Part I.

API-Misuse Detection

A History of API Misuse and API-Misuse Detection

Incorrect usages of an Application Programming Interface (API), or *API misuses*, are violations of (implicit) *usage constraints* of the API. An example of a usage constraint is having to check that `hasNext()` returns `true` before calling `next()` on an `Iterator`, in order to avoid a `NoSuchElementException` at runtime. API misuse causes software bugs, crashes, and vulnerabilities [FHM⁺12, GIJ⁺12, MM13, EBFK13, SHA15, NKMB16, LHX⁺16].

While high-quality documentation of an API’s usage-constraints could help to mitigate API misuse, it is often not enough to solve the problem, because developers often remain unaware even of documented constraints [DH09]. Moreover, a recent empirical study shows that Android developers prefer informal references, such as `STACK-OVERFLOW`, over official API documentation, even though the former contain more mistakes [ABF⁺16].

Ideally, development environments should assist developers in implementing correct usages and in finding and fixing existing misuses. Prevention of API misuse has been approached in various ways, e.g., through actively notifying developers about relevant API usage constraints [DH09] or recommending correct usage [HM05, BMM09, PLM15]. In this thesis, we focus on tools that automatically identify misuses in a given codebase, referred to as *API-misuse detectors*.

Despite the vast amount of work on API-misuse detection, API misuses still exist in practice, as recent studies show [LHX⁺16, ABF⁺16]. In order to advance the state of the art in API-misuse detection, we need to understand how existing approaches compare to each other, and what their current capabilities and limitations are. We need a general definition of API-misuse to assess each detector’s capabilities and to systematically address the problem space. This would allow researchers to improve API-misuse detection tools by enhancing the strengths and overcoming the weaknesses of current detectors.

To address these needs, in this part of the thesis, we present a conceptual analysis of the state of the art in API-misuse detection. In Chapter 2, we present our definition of API misuse as violations of API usage constraints. Using this definition, we estimate the potential for API misuse on the basis of existing empirical studies on API usage directives [METM12, ZGC⁺17]. We find that up to two thirds of all API elements come with usage constraints, and that the documentation of roughly every second API usage constraint is incorrect, even in the mature `JAVA CLASS LIBRARY`. We then conduct a review of over 1.200 bug reports from four general bug datasets and a developer survey to identify API misuses as they appear in software releases and during development time.

While we find that API misuses make up only about a tenth of all software bugs reported in production, these bugs also cause program crashes in more than four out of five cases. Furthermore, we find indicators that developers spend considerable time struggling to solve API misuses, even for APIs with thoroughly documented usage constraints; and that developers rarely fix misuses that do not cause crashes, but have less-obvious effects, such as vulnerabilities.

In Chapter 3, we propose the *API-Misuse Classification* (MuC) as both a taxonomy for API misuses and a framework to assess the capabilities of misuse detectors. We derive MuC from the 164 examples of API misuse that we previously identified.

In Chapter 4, we present the results of a systematic literature review to identify existing API-misuse detectors. We identify both static and dynamic detection approaches. Using MuC, we qualitatively compare the descriptions of 18 existing detectors and identify their conceptual capabilities and shortcomings. For example, we find that few detectors detect misuses related to conditions or exception handling. We confirm this assessment with the detectors' authors.

The terminology and the techniques presented in this part are helpful to understand the contributions of this thesis. The insights about the prevalence and impact of API misuses and the conceptual analysis of the state of the art in API-misuse detection motivate our work presented in the following parts of the thesis.

2. API Usage and Misuse

An *API usage* (*usage*, for short) is a piece of code that uses a given API to accomplish some task. The *API*, in an object-oriented language such as Java, consists of the interfaces of one or more types. The usage is a combination of basic *program elements*, such as method calls (both on instances of the respective types and to their static methods), control structures, or arithmetic operations. The combination of such elements in an API usage is subject to constraints, which depend on the nature of the API. We call such constraints usage constraints *usage constraints*. For example, two methods may need to be called in a specific order, division may not be used with a divisor of zero, and a file resource needs to be released along all execution paths. When a usage violates one or more of these constraints, we call it a *misuse*, otherwise a *correct usage*. We subsequently present four studies that demonstrate why API misuses occur, how prevalent they are, and why it is important to address this particular type of bug.

2.1. API Usage Directives

API misuses violate API usage constraints. We distinguish constraints that are enforced by the compiler, such as correct typing, from constraints that are not, i.e., whose violation may lead to problems and errors at runtime. Such constraints are often declared as *API usage directives* in the API documentation. Monperrus *et al.* [METM12] empirically investigated such directives and their prevalence in a study of three Java libraries, namely, the JAVA CLASS LIBRARY, JFACE, and the APACHE COMMONS COLLECTIONS. From their taxonomy of API usage directives, we take that API misuses violate usage directives from one of the following categories (in order of prevalence):

Parameter Directives, which mandate restrictions on parameters that are not enforced by the type system, such as non-`null` requirements, string format requirements, number range requirements, type requirements, or parameter-correlation requirements. In total, 22.2% of all analyzed API elements have such a directive.

Restrictions, which mandate actions to happen in a certain context, e.g., when a method must only be called from the UI thread. In total, 6.6% of all analyzed API elements have such a directive.

Protocol Directives, which mandate (a specific number of) calls to certain methods or a specific method-call order. In total, 6.2% of all analyzed API elements have such a directive.

2. API Usage and Misuse

Table 2.1.: The Prevalence of API Misuse by Source of Software Defects.

Source	Candidates	Reviewed	Misuse	Crash
BUGCLASSIFY	2,914	294 (10.1%)	24 (8.2%)	14 (58.3%)
DEFECTS4J	357	357 (100.0%)	17 (4.8%)	15 (88.2%)
iBUGS	390	390 (100.0%)	40 (10.3%)	35 (87.5%)
QACRASHFIX	24	24 (100.0%)	16 (66.7%)	16 (100.0%)
DEVELOPER SURVEY	18	18 (100.0%)	18 (100.0%)	12 (66.7%)
JCA Misuses	15,752	38 (2.4%)	31 (81.6%)	0 (0.0%)
JCA Fixes (SF)	130	130 (100.0%)	11 (8.5%)	2 (18.2%)
JCA Fixes (GH)	2,660	78 (2.9%)	3 (3.9%)	2 (66.7%)
USAGE DIRECTIVES	10	10 (100.0%)	10 (100.0%)	5 (50.0%)
Total	22,255	1,339 (6.0%)	170 (12.7%)	96 (56.5%)

Return-Value Directives, which define properties of return values that are not enforced by the type system, e.g., that the value may be `null` or have a certain state. In total, 5.1% of all analyzed API elements have such a directive.

Exception Directives, which explain the exceptional behavior of a method that usages need to deal with, such as a possible connection timeout when connecting to a server. In total, 4.1% of all analyzed API elements have such a directive. Note that this category includes exceptional behavior in case of invalid parameter values, which usually co-occurs with parameter directives.

Synchronization Directives, which mandate the use of synchronization in usages. In total, 3.9% of all analyzed API elements have such a directive. Note that this category includes only directive that formulate requirements for thread-safe use and excludes declarations of thread-safety.

Limitations, which explain what a method does not do, but that usages are expected to take care of, e.g., that adding elements to a UI panel does not make them visible, until the usage calls a dedicated update method. In total, 0.9% of all analyzed API elements have such a directive.

Monperrus *et al.* report that, overall, 66.5% of all analyzed API elements have at least one directive. Note, however, that this also includes further directives that do not formulate usage constraints, such as *Null-Allowed Directives*, which inform that a parameter may be `null` or *Alternative Directives* that inform about alternative ways to achieve the same goal (with potentially different trade-offs). These other directives appear on at least 24.8% of the analyzed API elements, which means that the number of methods with directives that formulate usage constraints is somewhere between 41.7% and 66.5%. In any case, the study shows that a significant fraction of API elements comes with usage constraints and may, thus, potentially be misused.

Dekel *et al.* [DH09] find that developers are often unaware of documented directives and that they, therefore, miss to consider them, unless they are actively pointed at the directives. Zhou *et al.* [ZGC⁺17] demonstrate that from a randomized sample of 1,975 usage constraints mined in the `java.awt` and `javax.swing` libraries, 1,413 (71.5%) are incorrectly documented and from a second randomized sample of 2057 usage constraints mined in the `java.lang`, `java.util`, `java.security`, `java(x).sql`, `javax.management`, and `javax.xml` libraries, 772 (37.5%) are incorrectly documented. Overall, this means that more than half (51.2%) of all directives mined in the JAVA CLASS LIBRARY are incorrect. Hence, even if developers would be aware of all documented directives, they might still often inadvertently misuse APIs.

2.2. Misuses in Bug Reports

Previous work shows that API misuses causing crashes or data loss are prevalent in today’s software [MM13, LHX⁺16, MCJ17]. However, to the best of our knowledge, no work empirically compares the prevalence of API misuses to that of other types of bugs. This makes it hard to judge the impact of research that addresses API misuse.

To address this problem, we manually review the bugs in four existing bug datasets. These datasets were originally constructed to evaluate approaches to problems such as bug triaging [HJZ13], fault localization [DZ07, JJE14], and automated program repair [GZW⁺15, JJE14]. They encompass bugs reported in various projects’ issue trackers or fixed in their version-control history. Identifying the API misuses in these general bug datasets allows us to assess the prevalence of API misuses among all bugs reported from production environments. The first four rows of Table 2.1 summarize the results, which we discuss subsequently.

BugClassify This dataset by Herzig *et al.* [HJZ13] consists of 7,401 tickets from the issue trackers of five Open Source projects. They manually classified 2,914 of these tickets as reporting bugs. We randomly selected 10% of those tickets for each of the five projects, a total of 294 tickets, from which we identified 24 API misuses (8.2%). We found that most tickets report logic errors, such as wrong calculations or missing handling of certain cases. Other categories are mistakes in configuration files and multi-threading issues, such as race conditions.

Defects4J This defect dataset by Just *et al.* [JJE14] consists of 357 source-code bugs from six Open Source projects. Each defect is reported in an issue tracker, fixed in a single commit, and had at least one accompanying test case that failed before and passed after the fix. From all of these cases, we identified 17 API misuses (4.8%).

iBugs This dataset by Dallmeier and Zimmerman [DZ07] consists of 390 fixing commits from three Open Source projects. They selected the commits through heuristics on the commit messages. From all of these cases, we identified 40 API misuses (10.3%). Many

2. API Usage and Misuse

of the other issues were unrelated to API usage and often even appeared in non-code files, such as configuration or documentation.

QACrashFix This dataset by Gao *et al.* [GZW⁺15] consists of 24 source-code bugs from 16 Open Source projects. They selected the bugs by, first, mining the issue trackers of the projects for crash reports related to the Android API and, second, reviewing the resulting candidates manually. From all of these cases, we identified 16 API misuses (66.7%). Interestingly, a very large part of these crash bugs are API misuses.

Overall, we find that only 9.1% of the bugs reported on the respective projects are due to API misuses. However, many of those misuses cause crashes (82.5%), which stresses the importance of mitigating this kind of bug. Moreover, we found that many of the fixing commits resolve multiple misuses, because often the same mistake occurs also in other usages of the same API. Such consistency in the API misuse suggests that the original authors were generally unaware of the respective usage constraints, rather than momentarily inattentive, and that they would make the same mistakes again in the future.

2.3. Misuses at Development Time

Previous work suggests that developers struggle with the correct usage of APIs during development [SHA15, NKMB16]. Since many API misuses lead to obviously spurious behavior, such as program crashes (see Section 2.2), we hypothesize that developers might introduce and fix many misuses in their day-to-day work. We might not ever encounter such misuses in code repositories or released software products, which is why we call them *transient misuses*.

To identify examples of transient misuses, we conducted a survey, which we promoted via colleagues, friends, and TWITTER, to reach developers. The middle segment of Table 2.1 summarizes the results. Within nine days, we collected 18 responses naming as many distinct API misuses. We provide the questionnaire and all responses on an artifact page.¹ While the size of this survey is limited, the responses still allow for some interesting observations.

First, we find examples of only three of these 18 reported misuses in the bug reports reviewed in Section 2.2. This suggests that transient misuses are indeed different from the misuses we encounter in code repositories or released software products.

Second, many respondents point out multiple possible fixes for a single misuse. For example, to ensure a resource gets closed, one may explicitly invoke `close()` on the resource, or pass the resource to `IOUtils.closeQuietly()` from `APACHE COMMONS`, or use the try-with-resources statement to have Java automatically close the resource. This suggests that there are multiple correct alternatives for using an API to implement a specific task.

¹<http://www.st.informatik.tu-darmstadt.de/artifacts/api-misuse-survey/>

While the survey results suggest that many API misuses are resolved early on in the development process, fixing them may still distract developers from their original tasks and slow down development. To get an impression of the potential impact, we conduct a small experiment: one of the survey respondents describes the problem of getting a `ConcurrentModificationException` when using an `Iterator` after modifying the underlying collection. To assess how frequently developers face this problem, we search `STACKOVERFLOW` for the keyword `ConcurrentModificationException`. This search returns 2854 threads,² of which we manually review the top-151 threads, according to `STACKOVERFLOW`’s relevance ranking.³ We find that 88 of these threads (58.3%) ask how to fix some code that contains exactly the misuse in question and 5 more threads (3.3%) ask about best practices for iterating collections while modifying them. Another 39 threads (25.8%) ask about related issues in multi-threaded environments and six threads (4.0%) about respective exceptions being thrown by 3rd-party libraries. For the remaining 13 threads (8.6%) we cannot determine the ultimate cause of the problem from the discussions. These results suggest that, even for a widely known and well-documented API, developers frequently have to go as far as asking for help online—a rather slow problem-resolution strategy. Therefore, we argue that it is important to help developers find and fix such misuses or to avoid them in the first place.

2.4. Misuses Causing Vulnerabilities

A study of 269 Common Vulnerabilities & Exposures (CVE) by Lazar *et al.* [LCWZ14] reports that 83% of all bugs are misuses of cryptographic libraries by individual applications. Such misuses lead to plaintext disclosure [EBFK13, LCWZ14] and insecure network communication [FHM⁺12, GIJ⁺12, LCWZ14], or facilitate brute-force attacks [LCWZ14]. Egele *et al.* [EBFK13] report that 88% (10,327 of 11,748) Android apps on the Google Play marketplace violate at least one of six basic rules for safe usage of the Java Cryptographic Architecture (JCA) APIs (see Appendix A). A survey by Nadi *et al.* [NKMB16] reveals that developers indeed lack necessary knowledge to use the JCA APIs and that they would welcome assistance in doing so correctly. This again underlines the criticality of software bugs related to API misuse.

To find examples of such misuse, we mine JCA usages as follows:

1. We query the repository-mining infrastructure BOA [DNRN13] (GitHub September, 2015 full snapshot) for projects that use the JCA encryption API `Cipher`, i.e., whose latest source code contains imports of either `javax.crypto.Cipher` directly or the entire `javax.crypto` package. The respective query script is provided as Appendix B.
2. We generate GitHub links to the source files containing the respective usages.

²As of February 1, 2016.

³Artifact Page: `STACKOVERFLOW` Study on `ConcurrentModificationException`
(<http://www.st.informatik.tu-darmstadt.de/artifacts/stackoverflow-cme/>)

2. API Usage and Misuse

3. We manually verify the correctness of the respective candidate usages with respect to the specification provided by Krüger *et al.* [KSA⁺18].

The third-last row in Table 2.1 summarizes the results of this process. BOA identifies 15,752 `Cipher` usages in 1,573 `GITHUB` projects. This corresponds to 0.02% of the 7,830,023 projects in the entire BOA dataset, which suggests that only a comparably small number of applications uses the JCA at all. Nevertheless, reviewing all candidate projects is still practically infeasible. Therefore, we randomly sample projects and review the respective usages, until we identify at least 30 misuses. In total, we reviewed 38 usages in 26 projects and identified 31 misuses (81.6%) in 19 (73.1%) of these projects. We note that the misuse ratio of 81.6% is close to the ratio of 88% that Egele *et al.* [EBFK13] report for Android apps. The most common misuse (fourteen misuses) is to specify an insecure algorithm for encryption; the second-most common (ten misuses) is to rely on a potentially insecure default configuration of the crypto provider. This suggests that projects that do use the `Cipher` API indeed commonly misuse it.

As opposed to many of the misuses we identified in Section 2.2, the JCA misuses we identified do not cause obviously spurious behavior, such as program crashes. In fact, specifying an insecure algorithm for encryption works perfectly fine. To get an idea as to whether and how such misuses get fixed, we mine bug-fixing changes in JCA usages as follows:

1. We query the repository-mining infrastructure BOA [DNRN13] (`GITHUB` September, 2015 full snapshot) for projects that use the JCA APIs, i.e., whose latest source code contains imports from the `javax.crypto` package.
2. We identify potentially bug-fixing changes by matching commit messages against keywords that indicate bug fixes, using the approach from Zimmerman *et al.* [ZPZ07].
3. For each fixing change, we extract the actual source-code change [NNW⁺10] and analyzed the changes to the abstract syntax tree [NNP⁺12] to find changes to the usages of a JCA type.
4. We manually verify whether the respective candidate usages violate the specification provided by Krüger *et al.* [KSA⁺18] and whether the change resolves this.

The last two rows in Table 2.1 summarize the results of this process. We extracted 130 candidates from `SOURCEFORGE`, from which we identified 11 misuses (8.5%), and 2660 candidates from `GITHUB`, from which we reviewed all 78 candidates from a random sample of 15 projects and identified 3 misuses (3.9%). Interestingly, only one change fixes an unsafe algorithm configuration. All other changes fix misuses that lead to spurious behavior, the most common (10 cases) being crashes due to invalid input, such as an incompatible encoding for the data to encrypt or a cryptographic key that does not match the encryption algorithm. This suggests that developers are less likely to discover and fix inconspicuous misuses, which underlines the need for tools that identify such problems.

2.5. Summary

We reviewed 1,339 bugs and bug candidates from nine sources. We identify API misuses in 23 out of 30 real-world projects (76.7%) from the bug datasets, which demonstrates that the problem is indeed prevalent. While only a relatively small fraction of all bugs that appear in production are misuses (9.1%), many of these cause application crashes (82.5%). Furthermore, developers seem to struggle with API misuses that cause obvious misbehavior in their daily work, which may considerably slow down development, although respective misuses never appear in production. And finally, we identify vulnerable JCA usages in 19 out of 26 projects (73.1%), which confirms previous findings regarding the prevalence of API misuses causing vulnerabilities [FHM⁺12, GIJ⁺12, EBFK13, LCWZ14]. Yet, we find fixes of such vulnerabilities in only 8 out of 65 projects (12.3%). This suggests that vulnerabilities are less likely to get fixed than other misuses, possibly because they cause no obvious misbehavior.

In the studies on API-usage directives [DH09, METM12], we identify 10 additional examples of API misuse, of which we did not find concrete instances via any of the other sources. This suggests that the problem of potential API misuse extends to a large number of APIs.

Overall, we identified 170 API misuses in 50 projects, the survey responses, and two studies on API-usage directives. This suggests that providing developers with assistance to prevent, identify, and resolve API misuses is very important to increase the overall quality, stability, and security of software products.

2.6. Limitations

In our work, we focus on misuse of Java APIs. Our findings may not generalize to API misuse in other programming languages.

The sample of API misuses we identified may not be representative for Java API misuses in the wild. We manually reviewed more than 1,200 bugs from established general bug datasets, to identify examples of API misuses (Section 2.2). This leads us to believe that our sample covers a broad variety of misuses as they manifest in software projects. Our developer survey provides evidence that there are other misuses that may not ever appear in code repositories, but that developer nonetheless struggle with. Future work should investigate whether there are distinctive properties of these two categories of misuses and how this impacts research on API misuse.

The review of bug datasets (Section 2.2) and survey responses (Section 2.3) was initially done by the author of this thesis alone; the review of the JCA usages by one of his colleagues, who previously did research on the usability of cryptographic APIs [NKMB16]. In both cases, only a single person reviewed the candidates. It is possible that we missed examples of misuse in this process and that the actual prevalence of API misuses is higher than we reported.

It is also possible that we identified examples that are not actually API misuses. We published all examples in May 2016, to allow others to review and use them for their

2. API Usage and Misuse

work, to mitigate this threat. Since the original publication, the author of this thesis created a dataset from the examples and used it to benchmark API-misuse detectors (see Part II of this thesis). In this process, each misuse example was revisited and discussed repeatedly between him and at least two of his colleagues. A single false positive was discovered along the way and excluded from the dataset. The data presented in this part of this thesis was updated to reflect this decision.

The size of the developer survey we conducted (Section 2.3) is small. The responses are unlikely to present a representative picture of the API misuses that developers face in their day-to-day work. We conducted this preliminary survey to get a first impression of possible differences between API misuses that occur during development time and misuses that cause bug reports. Future work should conduct more comprehensive studies to investigate the impact and nature of API misuse at development time.

2.7. Related Work

Lazar *et al.* [LCWZ14] study 269 Common Vulnerabilities & Exposures (CVE) and find that 83% of all bugs are misuses of cryptographic libraries by individual applications. Their work motivated us to study misuses of the JCA APIs as an example of API misuses with severe consequences other than application crashes. We find that usages of the respective APIs are relatively rare, at least on GITHUB. However, from the usages we reviewed, we indeed classified 81.2% as misuses.

Zhong *et al.* [ZS15] studied over 9,000 bug fixes from six Open Source Java projects to empirically validate assumptions underlying program repair techniques. They report that developers make API repair actions in half of the source files involved in fixes, i.e., they add, modify, or delete a statement that contains an API element, e.g., add a check on the result of an API call. They define API as any code element declared outside the target project itself. However, they do not distinguish whether the respective bugs are violations of API usage constraints or different kinds of problems. We find that API misuses indeed only amount to about 10% of all bugs. Zhong *et al.* also find that fixes more often add statements than remove them, suggesting that missing-element violations are more prevalent than redundant-element violations. Our findings confirm this (see Table 3.1).

Dekel *et al.* [DH09], Sushine *et al.* [SHA15], and Nadi *et al.* [NKMB16] provide evidence that developers struggle with API misuse already during development time. Motivated by their observations, we conducted our developer survey to identify concrete examples of APIs and usage constraints that developers struggle with. We find that these problems are mostly distinct from those we identified in code repositories and that developers apparently spent much time resolving respective issues, resorting to ask for help online even for misuses of well-documented APIs.

3. A Taxonomy of API Misuses

Past research on API misuses shows that there are different kinds of misuses: For example, Monperrus *et al.* [MM13] report that issues related to missing method calls are prevalent in bug trackers, forums, newsgroups, commit messages, and source-code comments. Thummalapenta *et al.* [TX09b] specifically target the detection of missing preconditions of method calls, and Wasylkowski *et al.* [WZL07] investigate problems where methods are called in the wrong order. However, to the best of our knowledge, no work systematically defined the problem space of API misuse. This prevents us from assessing which aspects of API misuse have been addressed or may have been neglected by existing approaches, and to compare these approaches to one another.

To improve on this situation, in this chapter, we introduce the *API-Misuse Classification* (MUC), a taxonomy of API misuses and framework for the evaluation and comparison of the capabilities of API-misuse detectors. In Chapter 4, we use MUC to qualitatively compare the capabilities of existing API-misuse detectors. In Chapter 7, we use MUC to define our expectations on the detectors' performance in an empirical evaluation.

3.1. Motivation

The IEEE has a standard for classifying defects [IEE10], which served as the basis for IBM's ORTHOGONAL DEFECT CLASSIFICATION (ODC) [CBC⁺92]. The ODC uses the defect type as one of the aspects from which to classify the defect. The defect type is composed from a conceptual element of the program, such as a function, check, assignment, documentation, or algorithm, and a violation type, i.e., either *missing* or *incorrect*. El Emam *et al.* [EW98] presented an adaptation of the ODC to a particular project setting through removal of some defect types and addition of others. More recently, Beller *et al.* [BBMZ16] presented the GENERAL DEFECT CLASSIFICATION (GCD), a remote ODC-descendant, tailored to compare the capabilities of automated static-analysis tools. Either classification captures the entire domain of all software defects. To compare the capabilities of API-misuse detectors, we need a more fine-grained differentiation of a subset of the categories they encompass.

Past work presented empirical studies and taxonomies of API-usage directives [DH09, METM12]. Many of these directives can be thought of as usage constraints in our terminology and their violations, consequently, as misuses. Other directives, however, do not formulate constraints. Examples are directives that explicitly allow `null` to be passed as a parameter and directives that inform about alternative ways to achieve a behavior (possibly with different trade-offs). Therefore, we cannot directly convert a taxonomy of usage directives into a taxonomy of misuses. Instead, we create a taxonomy

3. A Taxonomy of API Misuses

of API misuses based on the examples of API misuses we previously collected (see Chapter 2). We consider the usage directives that can be viewed as usage constraint in this process, through the hand-crafted examples of misuses that we derives from the examples presented in the studies.

3.2. The Classification

We developed MUC using a variation of Grounded Theory [GS67]: Following our notion of API misuses as API usages with one or more violations of usage constraints, the first author of this work went through all the misuse examples identified in Chapter 2 and came up with labels for the characteristics of the respective violations, until each misuse was tagged with at least one label. Subsequently, the author and three of his colleagues iteratively revisited the labeled misuses to unify semantically equivalent labels and group related labels, until we had a consistent taxonomy. In the end, we had two dimensions whose intersection describes all violations in the examples: the type of the involved API-usage element and the type of the violation.

Violation A *violation* is a pair of a violation type and an API-usage element.

API-Usage Element An *API-usage element* is a program element that appears in API usages. The following elements are involved in the misuses in the misuse examples we identified in Chapter 2: *method calls*, *conditions*, *iterations*, and *exception handling*. Note that we consider primitive operators, such as arithmetic operators, as methods. For conditions, we further distinguish *null checks*, *synchronization conditions*, *context conditions*, and other *value or state conditions*, because of their distinct properties.

Violation Type The *violation type* describes how a usage violates a given usage constraint with respect to a given usage element. Among the misuse examples we identified in Chapter 2, we find two violation types: *missing* and *redundant*. Violations of the missing type come from constraints that mandate the presence of a usage element. They generally cause program errors. An example of such a violation is a “missing method call.” Violations of the redundant type come from constraints that mandate the absence of a usage element or declare the presence of a usage element unnecessary. Note that in either case the repetition of an element may have undesired effects, such as errors or decreased performance. An example of such a redundant violation is a “redundant method call.”

Table 3.1 shows a summary of MUC. The numbers in the cells denote how many misuses with a respective violation among the misuse examples we identified in Chapter 2. Note that since a single misuse may have multiple violations, the individual cells in the table sum up to more than the 170 misuses we collected. The table shows that missing

Table 3.1.: The Misuse Classification (MuC), with the Number of Misuses with a Particular Violation among the Examples Identified in Chapter 2, Ordered by Prevalence.

		Violation Type	
		Missing	Redundant
API-Usage Element	Condition	109	7
	Value or State	56	1
	<code>null</code> Check	51	4
	Synchronization	1	1
	Context	1	1
	Method Call	45	17
	Exception Handling	16	1
	Iteration	1	2

value or state conditions and missing `null` checks are the most prevalent violations, followed by missing method calls. Redundant calls and missing exception handling are less frequent, but still prevalent, while we have only few examples for the other violations.

We now discuss the different violation categories shown in Table 3.1, grouped by the API-usage element involved.

3.2.1. Method Calls

Method calls are the most prominent elements of API usages, as they are the primary means of communication between client code and the API.

One violation category is *missing method calls*, which occur if a usage does not call a certain method that is mandated by the API usage constraints. For example, if a usage does not call `validate()` on a `JFrame` after adding elements to it, which is required for the change to become visible.

The other case is *redundant method calls*, which occur if a usage calls a certain method that is restricted by the API usage constraints. For example, if a usage calls `remove()` on a list that is currently being iterated over, which causes an exception in the subsequent iteration. Or, as another example, if a usage calls `finalize()` on an `Object`, which should never be done from any user-defined code.

3.2.2. Conditions

Client code often needs to ensure conditions for valid communication to an API, in order to adhere to the API's usage constraints. There are often alternative ways to ensure such conditions. For example, to ensure that a collection is not empty one may check `isEmpty()`, check its `size()`, or add an element to it. Note that checks, in particular, are also a means for the client code to vary usages depending on program inputs.

3. A Taxonomy of API Misuses

One violation category is *missing conditions*, which occur if a usage does not ensure certain conditions that are mandated by the API usage constraints. One case is *missing null checks*, e.g., if a usage fails to ensure that a receiver or a parameter of a call is not `null`. Another case is *missing value or state conditions*, e.g., if a usage fails to ensure that a `Map` contains a certain key before using the key to access the `Map`. In multi-threaded environments, *missing synchronization conditions* may occur, e.g., if a usage does not obtain a lock before updating a `HashMap` that is accessed from multiple threads [METM12]. Finally, *missing context conditions* may also occur, e.g., if a usage fails to ensure that GUI components in SWING are updated on the Event Dispatching Thread (EDT) [DH09].

The other case is *redundant conditions*, where a condition prevents a necessary part of a usage, e.g., a method call, from being executed along certain execution paths or is simply redundant. One case is *redundant null checks*, e.g., if the usage checks nullness only after a method has been invoked on the respective object. Another case is *redundant value or state conditions*, e.g., if the usage checks `isEmpty` on a collection that's guaranteed to contain an element. In multi-threaded environments, *redundant synchronization conditions* may occur, e.g., if the usage requests a lock that it already holds, which may cause a deadlock. Finally, *redundant context conditions* may also occur, e.g., if a JUNIT assertion is executed on another thread, where its failing cannot be captured by the JUNIT framework.

3.2.3. Iteration

Iteration is another means of interacting with APIs, used, in particular, with collections and IO streams. It takes the form of loops and recursive methods. Note that respective usage constraints are about (not) repeating (part of) a usage, rather than about the condition that controls the execution.

One violation category is *missing iterations*, which occur if a usage does not repeatedly check a condition that the API usage constraints mandate must be checked again after executing part of the usage. For example, the Java documentation states that calls to `Object.wait()` must always happen in a loop. A usage calls `wait()` to pause the current thread until it receives an interrupt. It does this to wait until a certain condition holds, e.g., until a resource becomes available. Since interrupts may occur any time, the resumed usage must check the condition and, if it does not hold, call `wait()` again. Note that this may happen an arbitrary number of times before the condition becomes true. Thus, a usage that checks the condition with an `if` still violated the usage constraint.

The other case is *redundant iterations*, which occur if part of a usage is reiterated that the API usage constraints mandate may be executed not more than once or that is imply redundant. For example, a `Cipher` instance might be reused in a loop to encrypt a collection of values, but its initialization through calling `init()` must happen exactly once, i.e., before the loop. Note that, if `init()` is called inside the loop, there is exactly one call to the method—as required by the usage constraints—but its inclusion in an iteration causes a violation.

3.2.4. Exception Handling

Exceptions are a way for APIs to communicate errors to client code. The handling of different errors often depends on the specific API.

One violation category is *missing exception handling*, which occurs if a usage does not take actions to recover from a possible error, as mandated by the API usage constraints. For example, when initializing a `Cipher` with an externally provided cryptographic key, one should handle `InvalidKeyException`. Another example is resources that need to be closed after use, also in case of an exception. Such guarantees are often implemented by a `finally` block, but also using the `try-with-resources` construct or even respective handling in multiple `catch` blocks.

The other case is *redundant exception handling*, which occurs if a usage intercepts exceptions that should not be caught or handled explicitly. For example, catching `Throwable` when executing a command in an application might suppress a `CancellationException`, preventing the user from cancelling the command.

3.3. Limitations

We derive MUC from the misuse examples identified in Chapter 2, i.e., from 147 API misuses identified in 50 real-world projects and 17 further misuses identified through a developer survey. These examples may not be representative for API misuses in general, hence, MUC may miss some violation categories. However, the examples were identified through a review of over 1,200 reports from state-of-the-art bug datasets as well as developer input and we find that a relatively small number of criteria suffices to characterize all these misuses. We see this as an indicator that we covered a large fraction of the different kinds of API misuses. Meanwhile, we also classified the capabilities of 18 existing misuse detectors according to MUC (see Chapter 4) and found that all these detectors' capabilities are covered by MUC. This makes it unlikely that we miss a prevalent violation category. The classification enables us, for the first time, to gather empirical data about API misuses.

4. A Survey of API-Misuse Detectors

Figure 4.1 depicts the solution space for the detection of API-misuses. The detection may be approached through *static analyses* of source code or binaries and through *dynamic analyses*, i.e., runtime monitoring or analysis of runtime data, such as traces or logs. In either case, the detection requires either specifications of correct API usage to find violations of or specifications of misuses to find instances of. Such specification may be *crafted manually* by experts or *inferred automatically* by algorithms. Automatic specification inference (or *mining*) may, again, be approached both *statically*, e.g., based on code samples or documentation, and *dynamically*, e.g., based on traces or logs. Statically inferred specifications are often referred to as *patterns*.

Since manually crafting and maintaining specifications is costly, in this work, we focus on automated detectors. We call such tools *API-misuse detectors*. In the literature, we find *static misuse detectors* that statically mine specifications and detect misuse through static analysis, e.g., [WZL07, NNP⁺09b, MM13]; *dynamic misuse detectors* that dynamically mine specifications and detect misuses through dynamic analysis, e.g., [PG12, LZL⁺14]; and *hybrid misuse detectors* that, for example, combine dynamic specification mining with static detection [PJAG12].

To advance the state of the art of API-misuse detection, we need to understand the capabilities and short-comings of existing misuse detectors. Therefore, we conduct a systematic literature review to identify existing API-misuse detectors and assess their underlying approach, their conceptual capabilities with respect to MUC, and the evaluation setting they were tested in (Section 4.2). Then we compare the different detectors and the respective evaluations to establish the big picture of the state of the art (Section 4.3).

4.1. Methodology

We performed a systematic literature survey in two rounds: The first round was conducted in early 2016 and started from a survey of automated API-property inference techniques by Robillard *et al.* [RBK⁺13] and the 2013 to 2015 proceedings of the ICSE, FSE, and ASE conferences (and their respective colocated events). The second round was conducted in late 2017 and started from the 2016 and 2017 proceedings of the same conferences. In both rounds, the author of this thesis proceeded as follows:

1. He filtered the proceedings for publications whose title or abstract contain one of the keywords **API**, **usage**, **error**, **mine**, **mining**, **specification**, **verification**, or **bug**.

4. A Survey of API-Misuse Detectors

			Detection	
			Static	Dynamic
Specifications	Mined	Static	API-Misuse Detectors	
		Dynamic		
	Hand-crafted		Specification Verifiers	

Figure 4.1.: Solution Space for the Detection of API Misuses.

2. He manually reviewed the title and abstract of filtered publications to identify those about either API-usage specifications, specification mining, program verification, or bug detection.
3. He checked any such publication for approaches to API-misuse detection. The first round included only approaches based on static specification inference and verification. The second round included also approaches based on dynamic analyses or manual specifications or both.
4. If a publication presents such an approach and that approach targets Java APIs, he added all references to other publications that supposedly present a misuse detector to the list of publications to check.

We present the misuse detectors identified in this survey process chronologically by their publication date. We mark approaches identified in the second round of the survey with *, because subsequent parts of this thesis base on the results of the first round only.

We use MUC for a qualitative comparison of detectors, i.e., we assess the *conceptual capabilities* of each detector with respect to MUC to obtain a *conceptual classification* of the existing detectors. We use the published description and evaluation results of each detector to identify which of MUC categories they can, conceptually, detect. To reduce subjectivity, we confirmed our capability assessment and the detector descriptions with the respective authors, except for PR-MINER and COLIBRI/ML, whose authors did not respond.¹ We also describe the strategies used to evaluate each detector and summarize those in Table 4.2.

4.2. API-Misuse Detectors

We subsequently present the results of our survey. Table 4.1 summarizes the capabilities of each detector with respect to MUC. Table 4.2 summarizes the empirical evaluation

¹ Note that we did this only for the detectors identified in the first round of our survey in 2016.

Table 4.1.: Capabilities of API-Misuse Detectors. ● denotes the capability to detect a violation. ● denotes the capability to detect a violation under special conditions. ○ denotes the inability to detect a violation.

	Target Language	Method Calls		Conditions				Ex. Handl.		Iteration		
		Missing	Redundant	Missing null	Missing Val./State	Missing Sync.	Missing Context	Redundant	Missing	Redundant	Missing	Redundant
Misuses Detector												
PR-MINER [LZ05]	C	●	○	○	○	○	○	○	○	○	○	○
CHRONICLER [RGJ07a]	C	●	○	○	○	○	○	○	○	○	○	○
COLIBRI/ML [Lin07]	C	●	○	○	○	○	○	○	○	○	○	○
JADET [WZL07]	Java	●	○	○	○	○	○	○	○	○	●	○
RGJ07 [RGJ07b]	C	●	○	●	●	○	○	○	○	○	○	○
LKL08 [LKL08]	Java	●	○	○	○	○	○	○	○	○	○	○
ALATTIN [TX09a]	Java	●	○	●	●	○	○	○	○	○	○	○
AX09 [AX09]	C	●	○	●	●	○	○	○	●	○	○	○
CAR-MINER [TX09b]	C++/Java	●	○	○	○	○	○	○	●	○	○	○
GROUMINER [NNP ⁺ 09b]	Java	●	○	●	●	○	○	○	○	○	●	○
OCD [GS10]	Java	●	○	○	○	○	○	○	○	○	●	○
DMMC [MBM10]	Java	●	○	○	○	○	○	○	○	○	○	○
SPECHECK [NK11]	Java	●	○	○	○	○	○	○	○	○	○	○
RRFINDER [WLW ⁺ 11]	Java	●	○	○	○	○	○	○	○	○	○	○
TIKANGA [WZ11]	Java	●	○	○	○	○	○	○	○	○	●	○
PJAG12 [PJAG12]	Java	●	●	○	●	○	○	○	○	○	○	○
PG12 [PG12]	Java	●	●	○	●	○	○	○	○	○	○	○
DROIDASSIST [NPVN15]	Java	●	●	○	○	○	○	○	○	○	○	○
SALENTO [MCJ17]	Java	●	●	○	○	○	○	○	○	○	○	○

methodology and results for each of the detectors in our survey. The first column names the detector and the respective publication presenting the evaluation. The second column shows the number of projects the detector was applied to in the evaluation. The third column shows which sample of the detector’s findings the authors reviewed, to determine the detector’s precision. The last column shows the detector’s precision. If the precision was reported per project, we report a range. We discuss the individual results for each of the detectors below.

4.2.1. PR-Miner

PR-MINER [LZ05] is a misuse detector for C. It encodes usages as the set of all function names called within the same function and then employs frequent-itemset mining to find patterns with a minimum support of 15 usages.

Violations here are strict subsets of a pattern that occur at least ten times less frequently than the pattern. To prune false positives, PR-MINER applies inter-procedural

4. A Survey of API-Misuse Detectors

Table 4.2.: Evaluation Summary of Surveyed API-Misuse Detectors with the Number of Target Projects (#TP) and the Number of Reviewed Findings (#RF).

Detector	Eval. Setting	#TP	#RF	Precision (Range)
PR-MINER [LZ05]	per-project	3	Top 60	18.1% (10-27%)
CHRONICLER [RGJ07a]	per-project	5	example-based	
COLIBRI/ML [Lin07]	per-project	5	example-based	
JADET [WZL07]	per-project	5	Top 10/project	6.5% (0-13%)
JADET [GWZ10]	multi-project	20	Top 25% (50)	8.0% (0-100%)
RGJ07 [RGJ07b]	per-project	1	example-based	
LKL08 [LKL08]	per-project	1	example-based	
ALATTIN [TX09a]	cross-project	6	Top 10/project	29.5% (13-100%)
AX09 [AX09]	per-project	3	All (292)	90.4% (50-94%)
CAR-MINER [TX09b]	cross-project	5	Top 10/project	60.1% (41-82%)
GROUMINER [NNP ⁺ 09b]	per-project	9	Top 10/project	5.4% (0-8%)
OCD [GS10]	per-project	10	4 of 7	25.0%
OCD [GS10]	per-project	2	All (15)	55.0% (40-70%)
DMMC [MBM10]	per-project	1	All (19)	73.7%
DMMC [MM13]	per-project	3	Top 30	56.7%
SPECHECK [NK11]	per-project	7	All (24)	54.2% (0-100%)
RRFINDER [WLW ⁺ 11]	cross-project	n/s	example-based	
TIKANGA [WZ11]	per-project	6	Top 25% (121)	9.9% (0-33%)
PJAG12 [PJAG12]	per-project	12	All (81)	32.1% (0-100%)
PG12 [PG12]	per-project	10	All (54)	100% (100%)
DROIDASSIST [NPVN15]	not evaluated			
SALENTO [MCJ17]	cross-project	250	Top 8% (48)	75%

analysis, i.e., for each occurrence of a violation, it checks whether the missing calls occur within a transitively called method. This analysis follows the call path for at most three levels. The reported violations are ranked by the respective pattern’s support.

PR-MINER detects missing method calls.

The evaluation by the authors of PR-MINER applied the detector to three target projects individually, thereby finding violations of project-specific patterns. The authors reviewed the top-60 violations reported across all projects and found 18.1% true positives (26.7%, 10.0%, and 14.3% on the individual projects).

4.2.2. Chronicler

CHRONICLER [RGJ07a] is a misuse detector for C. It mines frequent call-precedence relations from an inter-procedural control-flow graph. A relation is considered frequent, if it holds on at least 80% of all execution paths. Paths where such relations do not hold are reported as violations.

CHRONICLER detects missing method calls. Since loops are unrolled exactly once, it cannot detect missing iteration.

The evaluation by the authors of CHRONICLER applied the detector to five projects individually, thereby finding violations of project-specific patterns. The authors compare the identified protocols with the documented protocols for one API and discuss a few examples of actual bugs found by their tool, but report no statistics on the quality of the detector’s findings.

4.2.3. Colibri/ML

COLIBRI/ML [Lin07] is another misuse detector for C. It reimplements PR-MINER using *Formal Concept Analysis* [GW97] to strengthen the theoretical foundation of the approach. Consequently, its capabilities are the same as PR-MINER’s.

The evaluation by the authors of COLIBRI/ML applied the detector to five target projects individually, thereby finding violations of project-specific patterns. The authors present some detected violations in the paper, but report no statistics on the quality of the detector’s findings.

4.2.4. Jadet

JADET [WZL07] is a misuse detector for Java. It uses COLIBRI/ML [Lin07], but instead of only method names, it encodes method-call order and call receivers in usages. Therefore, it first builds a directed graph whose nodes represent method calls on a given object and whose edges represent control flows. From this graph, it then derives a pair of calls for each call-order relationship, e.g., $m() \prec n()$. Each usage is represented by the set of these pairs. These sets of call pairs form the input to the mining, which identifies patterns, i.e., sets of pairs, with a minimum support of 20.

A violation may miss at most two properties of the violated pattern and needs to occur at least ten times less frequently than the pattern. Detected violations are ranked by $u \times s/v$, where s is the violated pattern’s support, v is the number of violations of the pattern, and u is a uniqueness factor of the pattern. To compute u JADET counts for every API in the pattern the number of violations involving that API and takes the inverse of the largest such number. Intuitively, if an API is involved in more violations, any particular violation involving it is less likely to be problematic.

The encoding of call-order relations allows JADET to detect missing calls, even if the respective call appears at a different place in the usage. It may detect missing loops as a missing call-order relation from a method call in the loop header to itself. However, it cannot detect violations of patterns that consist of only two calls since such a pattern would be encoded as a set of a single pair of method calls. The only strict subset of such a pattern is the empty set, which is by definition not a violation.

The evaluation by the authors of JADET applied the detector to five target projects, thereby finding violations of project-specific patterns. The authors reviewed the top-10 violations reported per project and found 6.5% true positives (0%, 0%, 7.7%, 10.5%, and 13.3% on the individual projects). Other findings were classified as code smells (6.5%) or hints² (35.0%).

²Hints point at code that could be improved with respect to readability or maintainability [WZL07].

4. A Survey of API-Misuse Detectors

A later study [GWZ10] applied JADET to 6,097 projects at once (multi-project setting), using a minimum pattern support of 200. The authors considered the findings on a random sample of 20 of these target projects; a total of 136 findings. The authors reviewed the top-25% findings per project, a total of 50 findings, and found 8% true positives. Other findings were classified as code smells (14.0%).

4.2.5. RGJ07

RGJ07 [RGJ07b] is a misuse detector for C. It encodes usages as sets of properties for each variable v . Properties are comparisons to literals, e.g., (\neq, null) , if v was checked to be not `null`, argument positions in function calls, e.g., $(\text{arg}(2), f)$ if v was passed as the second argument to a function f , and assignments, e.g., $(:=, \text{res}(f))$ if the v was assigned the result of a call to f . For each call, RGJ07 creates a group of the property sets of the call's arguments. To all groups for a particular function, it applies sequence mining to learn common sequences of control-flow properties and frequent-itemset mining to identify all common sets of all other property types. In both cases, the mining uses a confidence threshold of 70%.

While mining patterns, RGJ07 also identifies violations of the common property sequences and sets (patterns), i.e., usages that cause the confidence for a particular pattern to be less than 100%.

RGJ07 is designed to detect missing conditions. From the properties it encodes, it can detect missing `null` checks and missing value/state conditions. Since patterns contain preceding calls on arguments, it may also detect missing calls, if the respective call shares an argument with another call in the pattern.

The evaluation by the authors of RGJ07 applied the detector to a single project, thereby finding violations of project-specific patterns. The authors discuss several examples of actual bugs their approach detects, but report no statistics on the quality of the detector's findings.

4.2.6. LKL08*

LKL08 [LKL08] is a misuse detector for Java. It mines specifications of the form **consequences** \hookrightarrow **premises** from method-call traces, where both the premises and consequences are sets of method calls. It requires that the respective traces contain only calls to methods of the relevant API(s) and searches for specifications with a minimum support of 15 and a confidence of 90% across all traces. The authors then manually transform the mined specifications into LTL formulae and use the PRISM model checker [HKNP06] to find respective violations.

LKL08 detects missing method calls and wrong call order.

The evaluation by the authors of LKL08 applied the detector to a single projects containing four known bugs. An existing test harness was used to generate traces. The authors reviewed the violations reported by LKL08 and found that the detector identifies three of the four known bugs.

* We identified this approach in the second round of our survey in 2017.

4.2.7. Alattin

ALATTIN [TX09a] is a misuse detector for Java, specialized in alternative patterns for condition checks. For each target method m , it queries the code-search engine GOOGLE CODE SEARCH³ to find example usages. From each example, it extracts a set of rules about pre- and post-condition checks on the receiver, the arguments, and the return value of m , e.g., “boolean check on return of `Iterator.hasNext` before `Iterator.next`” or “const check on return of `ArrayList.size` before `Iterator.next`.” It then applies frequent-itemset mining on the sets of these rules to obtain patterns with a minimum support of 40%. For each such pattern, it extracts the rule sets that do not adhere to the pattern and repeats mining on these, to obtain infrequent patterns with a minimum support of 20%. Finally, it combines all frequent and infrequent patterns for m by disjunction.

An analyzed method has a violation, if the set of rules that hold in it is not a superset of any of the alternative patterns. Violations are ranked by the support of the respective pattern.

ALATTIN detects missing null-checks and missing value/state conditions that are ensured by checks and that do not involve literals. It may also detect missing method-calls that occur in checks.

The evaluation by the authors of ALATTIN applied the detector to six projects. Since it queries a code-search engine for usage examples, it detects violations of cross-project patterns. The authors manually reviewed all violations of the top-10 patterns per project, a total of 532 findings, and confirmed that 29.5% identify missing condition checks (12.5%, 26.2%, 28.1%, 32.7%, 52.6%, and 100% for the individual projects). Considering frequent alternative patterns reduced false positives by 15.2% on average, which increases precision to 33.3%. Considering both frequent and infrequent alternatives even reduced false positives by 28.1% on average, leading to a precision of 37.8%, but introduced 1.5% additional false negatives, because misuses that occur multiple times are mistaken for infrequent patterns.

4.2.8. AX09

AX09 [AX09] is a misuse detector for C, specialized in detecting wrong error handling, realized through returning (and checking for) error codes. To this end, it distinguishes normal paths, i.e., execution paths from the beginning of the `main` function to its end, from error paths, i.e., paths from the beginning of the `main` function to an `exit` or `return` statement in an error-handling block. It uses push-down model checking to generate such paths as sequences of method calls and applies frequent-subsequence mining to find patterns with a minimum support of 80% (but at least 5 usages).

AX09 uses push-down model checking also to verify adherence to patterns and to identify respective violations. It then filters false positives by tracking variable values and excluding error cases that cannot occur, for example, if a usage does not close a

³ https://en.wikipedia.org/wiki/Google_Code_Search (checked on Dec 18, 2017)

4. A Survey of API-Misuse Detectors

file along the error path that is taken when the file could not be opened, i.e., when the respective file handle is `null`.

AX09 detects missing error-handling as well as missing method calls among error-handling functions. Since it identifies error-handling blocks through a predefined set of checks, it also detects missing `null`-checks and missing value/state conditions in the case of missing error-handling blocks.

The evaluation by the authors of AX09 applied the detector to three projects individually, thereby finding violations of project-specific patterns. The authors manually reviewed all 292 findings and confirmed 90.4% true positives (50.0%, 90.3%, and 93.5% on the individual projects).

4.2.9. CAR-Miner

CAR-MINER [TX09b] is a misuse detector for C++ and Java, specialized in detecting wrong error handling. For each analyzed method `m` in a given code corpus, it queries the code-search engine `GOOGLE CODE SEARCH`⁴ to find example usages. From the examples, it builds an *Exception Flow Graph* (EFG), i.e., a control-flow graph with additional edges for exceptional flow to and within `catch` and `finally` blocks. From the EFG, it generates *normal* call sequences that lead to the currently analyzed call and *exception* call sequences that lead from the call along exceptional edges. Subsequently, it mines association rules between normal sequences and exception sequences, with a minimum support of 40%. It ranks association rules by their support.

To detect violations, CAR-MINER extracts the normal call sequence and the exception call sequence for each target method call. It then uses the learned association rules to determine the expected exception handling and reports a violation if the actual sequence does not include it.

CAR-MINER detects missing exception-handling as well as missing method calls among error-handling functions.

The evaluation by the authors of CAR-MINER applied the detector to five projects. Since it queries a code-search engine for usage examples, it detects violations of cross-project patterns. The authors manually reviewed all violations of the top-10 association rules for each project, a total of 264 violations, and confirmed that 60.1% identify wrong error handling (41.1%, 54.5%, 68.2%, 68.4%, and 82.3% on the individual projects). Other findings were classified as hints (3.0%).

4.2.10. GROUMiner

GROUMINER [NNP⁺09b] is a misuse detector for Java. It creates a graph-based object-usage representation (GROUM) for each target method. A GROUM is a directed acyclic graph whose nodes represent method calls, branchings, and loops and whose edges encode control and data flows. GROUMINER uses an a-priori-based algorithm to detect frequent-subgraphs [RP15] on sets of such GROUMs, to detect recurring usage patterns with a minimum support of six. A-priori-based algorithms start from frequent

⁴ https://en.wikipedia.org/wiki/Google_Code_Search (checked on Dec 18, 2017)

single-node subgraphs and recursively extend known, frequent subgraphs by frequently adjacent neighbor nodes.

When at least 90% of all occurrences of a sub-pattern can be extended to a larger pattern, but some cannot, those *rare* inextensible occurrences are considered as violations. Note that such violations have always exactly one node less than a pattern. The detection of patterns and violations happens at the same time. Violations are ranked by their *rareness*, i.e., the support of the pattern over the support of the violation.

GROUMINER detects missing method calls. It also detects missing conditions and loops at the granularity of a missing branching or loop node. However, it cannot consider the actual condition.

The evaluation by the authors of GROUMINER applied the detector to nine projects individually, thereby finding violations of project-specific patterns. The authors reviewed the top-10 violations per project, a total of 184 findings, and found 5.4% true positives (three times 0%, five times 6.7%, and once 7.8% on the individual projects). Other findings were classified as code smells (7.6%) or hints (6.0%).

4.2.11. OCD*

OCD [GS10] is a misuse detector for Java. To mine and check temporal patterns, OCD observes a window of 25 events from the method-call traces and identifies pairs of subsequent calls to the same receiver. If no second call occurs within the window, it considers the first call as isolated. Both types of occurrences serve as evidence (or counter-evidence) for temporal patterns, based on a predefined set of pattern templates for sequential calls (ab), loop begin and end (ab^+ and a^+b), pre- and post-conditions ($ab?$ and $a?b$), and association rules ($(ab|ba)$).

OCD uses multiple thresholds to decide—based on the collected evidence and counter-evidence—whether a pattern should be enforced, i.e., the respective violations be reported, or not. It self-tunes these thresholds such that it reports only around ten violations. A ranking strategy is mentioned, but not discussed in the publication.

OCD detects missing method calls and wrong call order. Based on the pattern templates, we assume that it should be able to identify missing iteration in some cases.

The evaluation by the authors of OCD is twofold: First, it applied the detector to ten projects from the DCAPO benchmark suite [BGH⁺06] individually, thereby finding violations of project-specific specifications. The original test harnesses of the projects were used for execution. The authors reviewed the three violations of JAVA CLASS LIBRARY APIs reported by OCD and found one (33.3%) true positive. They also reviewed one of the four violations of other APIs reported by OCD, but found it to be a (non-obvious) false positive. Second, the evaluation applied OCD to the usages of JAVA CLASS LIBRARY APIs in two further projects individually. Both projects are applications that the authors manually interacted with, to generate inputs. The authors reviewed all violations reported by OCD and found 55% true positives (7 of 10 and 2 of 5 on the individual projects).

* We identified this approach in the second round of our survey in 2017.

4.2.12. DMMC

DMMC [MBM10] is a misuse detector for Java, specialized in missing method calls. It does not mine patterns, but rather computes a likelihood for every usage to be a misuse. This calculation is based on type usages, i.e., sets of methods called on a given receiver type in a given method, and the usage context, i.e., the signature of the method that the usage occurs in. Two usages are considered *exactly similar* if their respective sets match and *almost similar* if one of them contains exactly one additional method. DMMC assumes that violations should have only few exactly-similar usages, but many almost-similar ones. The likelihood of a usage x being a violation is expressed in the *strangeness score* $= 1 - |E(x)|/(|E(x)| + |A(X)|)$, where $E(x)$ is the set of usages that are exactly similar to x and $A(x)$ the set of those that are almost similar to x . Violations are ranked by their strangeness score.

DMMC detects misuses with exactly one missing method-call.

The evaluation by the authors of DMMC applied the detector to a single project, thereby finding project-specific violations for the STANDARD WIDGET TOOLKIT (SWT) of ECLIPSE. The authors manually reviewed all findings with a strangeness score above 97%, a total of 19 findings, and confirmed 73.7% as true positives, for which they submitted respective patches to the ECLIPSE project. Eleven of these patches were accepted, one was rejected, and the remaining two remained unanswered.

A later study [MM13] applied DMMC to three projects individually, thereby finding project-specific violations for a predefined set of APIs. The authors manually reviewed approximately 30 findings,⁵ and confirmed 17 ($\approx 56.7\%$) as true positives, for which they submitted respective patches to the projects. Eleven of these patches were accepted, two were rejected, two were ignored, because the respective code was no longer maintained, and two remained unanswered.

4.2.13. SpecCheck*

SPECHECK [NK11] is a misuse detector for Java. It uses the LM miner [DLMK10] to obtain specifications of the form **consequences** \leftrightarrow **premises** from method-call traces, where both the premises and consequences are sets of method calls. In this step, SPECHECK uses different values for support and confidence, depending on the size of the trace (higher values for larger traces, in discrete steps).

To determine the subset of significant specifications, SPECHECK removes the premise method calls from the instances of the specification in the training codebase and executes the mutated program. If this causes an exception at a consequence method call of a specification, for at least one instance of the specification, SPECHECK consider this specification as signification. Otherwise, it drops the specification. For specification with multiple premise method calls, SPECHECK repeats this check with a leave-one-premise-out strategy and drops premises whose omission does not indicate significance

⁵This is the quantity as reported in the publication.

*We identified this approach in the second round of our survey in 2017.

of the specification. Finally, SPECHECK combines multiple specification with the same consequences by conjunction.

To find misuses, SPECHECK uses JFTA [DKM⁺10], a static typestate verifier.

SPECHECK detects missing method calls and wrong call order.

The evaluation by the authors of SPECHECK applied the detector to seven projects individually, thereby finding violations of project-specific patterns. All projects were taken from the DACAPO benchmark suite [BGH⁺06] and the respective test harnesses were used for execution. The authors reviewed all violations reported by SPECHECK and found 54.2% true positives (0%, 33.3%, 33.3%, 50%, 62.5%, 66.7%, and 100% on the individual projects). Including the insignificant specifications produces no additional true positives, but 613 (55.7 times) more false positives.

4.2.14. RRFinder^{*}

RRFINDER [WLW⁺11] is a misuse detector for Java, specialized in detecting resource leaks. It first uses a classifier to identify resource-releasing (RR) methods in the implementation code of APIs that manipulate resources. This classifier uses several features of methods, such as keywords appearing in the method name or documentation, whether the class is overriding a known RR method, the percentage of statement in the method's body known to perform RR actions, and the percentage of other method in the class the fail after the method was invoked. Since some of these features depend on the classification of other methods, RRFINDER implements an iterative classification approach.

For each identified RR method $m()$, RRFINDER then heuristically determines respective resource-acquiring (RA) methods, by search for public methods that either assign a field that is set to `null` in $m()$ or invoke an RA method that corresponds to RR method invoked in $m()$. If no such method exists, RRFINDER assumes that the constructor of the class declaring $m()$ is the corresponding RA method. Each pair of an RR method and the set of corresponding RA methods for a RR specification.

The verification mechanism of RRFINDER is not detailed in the publication.

RRFINDER detects misuses with missing RA-method calls.

The evaluation trained the RR-method classifier on a manually curated dataset of 19,080 method from the JAVA CLASS LIBRARY. It then applied RRFINDER to an unspecified set of open source projects, using resource-release specifications mined for the APIs from eight popular libraries. The authors present one example of an actual bug that their approach detects, but report not statistics on the quality of the detector's findings.

4.2.15. Tikanga

TIKANGA [WZ11] is a misuse detector for Java that builds on the same algorithm as JADET. It replaces JADET's simple call-order properties by general Computation Tree Logic formulae on object usages. Specifically, it uses formulae that require a certain call to occur in a usage, formulae that require two calls in a certain order, and formulae that

^{*} We identified this approach in the second round of our survey in 2017.

4. A Survey of API-Misuse Detectors

require a certain call to happen after another. It uses model checking to determine the subset of all those formulae with a minimum support of 20 in the codebase. It then applies Formal Concept Analysis [GW97] to obtain patterns and violations at the same time. Violations are ranked by the *conviction* measure [BMUT97] of the association between the set of present formulae and the set of missing formulae in the violating usage.

TIKANGA’s capabilities are the same as JADET’s, but it also detects violations of patterns with only two calls.

The evaluation by the authors of TIKANGA applied the detector to six projects individually, finding violations of project-specific patterns. The authors manually reviewed the top-25% of findings per project, a total of 121 findings, and confirmed 9.9% as true positives (twice 0%, 8.3%, 20.0%, 21.4%, and 33.3% on the individual projects). Other findings were classified as code smells (29.8%).

4.2.16. PJAG12*

PJAG12 [PJAG12] is a misuse detector for Java, specialized on multi-object method-call protocols. It uses a dynamic specification miner [PG09, PBG10] to obtain multi-object specifications from method-call traces. The resulting specifications are finite-state automata (FSA), where transitions are method calls and states represent the respective objects’ state.

PJAG12 transforms the mined FSA specifications into FUSION specifications [JA09], i.e., into triples of a set of relationships between the involved objects, a set of call pre-conditions, and a set of call effects on relationships and object states. FUSION performs a static inter-procedural analysis to verify the specifications on a given target codebase.

To avoid false positives, PJAG12 prunes specifications using a support threshold and violations using thresholds on the number of illegal method calls in a violation divided by the number of all methods calls in the respective violated specification and on the number of calls to a method that are found to be illegal divided by the overall number of calls to this method.

PJAG12 detects missing and redundant method calls. It may also implicitly identify missing state conditions, if it finds a certain call to be illegal in the receiver’s current state, according to the respective FSA specification.

The evaluation by the authors of PJAG12 applied the detector to twelve projects from the DACAPO benchmark suite [BGH⁺06] individually, thereby finding violations of project-specific patterns. The training data for the specification miner was taken from prior work of the same authors [PBG10]. The authors reviewed all violations reported by PJAG12 and found 32.1% true positives (0%, 0%, 0%, 0%, 13.3%, 13.3%, 23.1%, 28.6%, 61.5%, 69.2%, 100%, and 100% on the individual projects) and additional 18.5% code smells.

* We identified this approach in the second round of our survey in 2017.

4.2.17. PG12*

PG12 [PG12] is a misuse detector for Java, specialized on method-call protocols. Given a target program and an API, PG12 first uses RANDOOP [PLEB07], a feedback-directed random test generator, to automatically execute the target code, prioritizing parts that use the API in question. Second, PG12 uses a dynamic specification miner [PG09, PBG10] (same as in PJAG12) to obtain specifications from the method-call traces of succeeding generated test runs, i.e., runs that did not terminate with an exception. Last, PG12 filters the traces of failing generated test runs to those that violate one of the mined specifications and where the violating code does not explicitly declare the respective exceptional case, and reports them as violations. Each violation is accompanied by a test case that provokes a respective crash.

PG12 detects missing and redundant method calls, if the respective violation causes an exception. Like PJAG12, it may also implicitly identify missing state conditions, if it finds a certain call to be illegal in the receivers current state (and this call causes an exception).

The evaluation by the authors of PG12 applied the detector to ten projects from the DACAPO benchmark suite [BGH⁺06] (the same projects that were used in the evaluation of OCD) individually, thereby finding violations of project-specific specifications. The authors reviewed all 54 violations reported by PG12 and found only true positives.

4.2.18. DroidAssist

DROIDASSIST [NPVN15] is a detector for Dalvik Bytecode (Android Java). It generates method-call sequences from source code and learns a Hidden Markov Model from them, using a modified version of the Baum-Welch algorithm, to compute the likelihood of a particular call sequence. If the likelihood is too small, the sequence is considered a violation. DROIDASSIST then explores different modifications of the sequence (adding, replacing, and removing calls) to find a slightly modified, more likely sequence. This allows it to detect missing and redundant method calls and even to suggest solutions for them. The authors of DROIDASSIST present no evaluation of this mechanism in the respective paper.

4.2.19. Salento*

SALENTO [MCJ17] is a detector for Dalvik Bytecode (Android Java). To detect misuses, SALENTO takes a dataset with training code, a dataset with target code, and a set of APIs to analyze. It uses symbolic execution to identify objects of the APIs' types and encodes each respective usage as a bag of all method calls on such an object. For each method call, SALENTO also encodes boolean predicates that capture constraints on call parameters, whether the call throws an exception, or other properties. Details on these predicates are not provided in the paper.

* We identified this approach in the second round of our survey in 2017.

* We identified this approach in the second round of our survey in 2017.

4. A Survey of API-Misuse Detectors

SALENTO assumes that each usage U conforms to a specification Z , which is unknown. It further assumes that the method calls X_U appearing in U inform about Z . The uncertainty about Z is formalized as $P(Z||X = X_U)$, where Z is a random variable over specifications and X is a random variable over method calls in usages. Moreover, SALENTO expresses the uncertainty regarding the behaviors Y of U as $P_U(Y)$, where Y is a random variable over behaviors, and allows for a distribution $P(Y||Z = Z)$ over the behaviors of usages that implement a given specification Z . In this framework, SALENTO learns a joint distribution $P(X, Y, Z)$ from the usage examples in the training code.

To detect misuses, SALENTO computes for each usage U in the target code the *anomaly score* as the statistical distance between $P(Y||X = X_U)$ and $P_F(Y)$. SALENTO reports U as a violation, if the usage’s anomaly score exceeds a threshold. The reported violations are ranked according to this score.

SALENTO detects missing and redundant method calls. It may be able to detect other kinds of misuses, depending on the properties it captures in its boolean predicates. However, the paper presents no evidence for SALENTO detecting other kinds of misuses.

The evaluation trained SALENTO on a dataset of 500 apps and applied it to detect misuses in a disjoint dataset of 250 apps. The first author reviewed the top-ranked violations for each target project with a violations in the top-10% of violations reported across the entire target dataset. They chose this sampling to avoid counting multiple findings that identify the same problem. SALENTO reports all its true positives in the top-8% of these findings (48 violations), which leads to a precision of 75% in the top-8% findings.

4.3. Discussion

Overall, we identified 19 API-misuse detectors in the literature. These detectors target either C, C++, or Java. Only two detectors have been applied across multiple languages: (1) CAR-MINER, which analyzes C++ and Java, and (2) COLIBRI/ML, which itself analyzes only C, but is also the foundation of JADET and TIKANGA, which analyze Java. Five detectors use dynamic analyses, while the other 14 exclusively use static analyses.

All detectors approach the detection of misuses through the detection of *deviant code* [ECH⁺01] (or deviant execution traces, in case of dynamic detection). The key idea is that mistakes violate constraints that usages should adhere to and that, given sufficiently many usage examples, such violations appear as *anomalies*. This follows two assumptions:

1. It assumes that usages that occur frequent correspond to correct usage or, in other words, that *the majority of usages is correct*. This seems intuitive, considering that misuses often cause spurious behavior (see Chapter 2), which would likely be noticed if it were the rule rather than the exception.
2. It assumes that anomalies with respect to frequent usages are misuses. This seems less intuitive, since such anomalies are, first of all, simply rare usages, which does

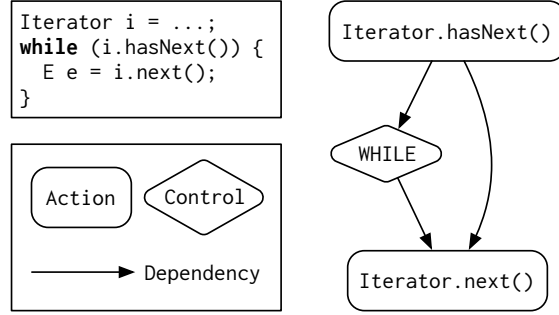


Figure 4.2.: A Correct Iterator Usage in GROUMINER’s Graph Representation

not imply anything regarding their correctness per se. Therefore, misuse detectors usually refine this assumption using some measure of distance between usages, i.e., they consider only anomalies that are very similar to a frequent usage as mistakes.

Previous evaluations show that respective approaches can successfully detect mistakes in the usage of popular libraries [ECH⁺01, LZ05, WZL07, NNP⁺09b, WZ11, MM13]. However, the low precision reported for many detectors also suggests that many anomalies with respect to correct usages are themselves correct usages.

4.3.1. Static Misuse Detectors

The static detectors all use code (snippets) as training and verification input. Some require the code in a compiled format, such as Java Bytecode, while others directly work on source code. They typically represent usages as sets, sequences, or graphs and mine patterns through frequent-itemset/subsequence/subgraph mining, according to their usage representation. Many static detectors perform pattern mining and anomaly detection at the same time, since violations are simply in-extensible parts of patterns that are themselves observed infrequently. Since many detectors produce a large number of findings (including many false positives), they propose a variety of ranking strategies to ensure true positives are reported first.

Two exceptions to the typical design of static detectors are DMMC and DROIDASSIST. DMMC computes the probability of each usage missing a method call based on all observed usages of the same type, i.e., it does not mine patterns. DROIDASSIST learns a HIDDEN MARKOV MODEL [RJ86] that allows it to compute the likelihood of method-call sequences to be correct. Consequently, it performs mining first and detection later.

The static detectors use both absolute and relative minimum-support thresholds to identify patterns. Relative thresholds are defined with respect to the absolute number of usage examples provided as input. The exceptions are, again, DMMC and DROIDASSIST, which use fixed thresholds on the respective probabilities.

For ranking violations, most static detectors rely mainly on pattern support, but some use different concepts, such as *confidence*, *rareness*, *strangeness*, or *conviction*. A comparison of different ranking strategies is unavailable in the literature.

4. A Survey of API-Misuse Detectors

A strength of static detectors is that their representations quite naturally encompass different usage elements and their relations, since they encode usages as abstractions from how they appear in code. GROUMINER, for example, represents a correct `Iterator` usage as depicted in Figure 4.2, encoding the two respective method calls, the iterative check on the result of `hasNext()`, and that it must precede the call to `next()`. Generally, graph representations seem promising for simultaneously encoding different usage elements, order, and data-flow relations.

Another strength of static approaches is that they can train on code examples from various sources, such as documentation, code-search engines, or Q&A sites, even if these are not compilable or executable. This makes it easier to obtain sufficiently many usage example for different APIs and enables cross-comparison of examples from different sources, which might help to mitigated biases. None of the approaches from the literature makes use of more than one source of usage examples, however.

A major limiting factor for the usefulness of static detectors is the availability of exhaustive examples of correct usage for pattern mining. Additionally, these examples need to appear sufficiently often, in order to be considered frequent, i.e., patterns. Insufficient training data is likely one of the main reasons for the large number of false positives that many static detectors report. It appears that the detectors that focus on specific violations, such as error handling (AX09 and CAR-MINER) or missing method calls (DMMC), have higher precision, possibly because they abstract more from the usage code and, therefore, need fewer examples to mine good patterns.

An inherent limitation of many current static detectors is that they define violations as in-extensible parts of patterns. As a consequence, they cannot detect redundant elements, as such an element is never part of any pattern. DROIDASSIST shows an alternative approach, using a probabilistic model, which might identify redundant calls as being unlikely.

4.3.2. Dynamic Misuse Detectors

The dynamic detectors all use method-call traces as training input. They learn either association rules between (sets of) method calls or finite-state automata representing object states. With the exception of OCD, they all conduct specification learning and validation in two separate phases, which allows them to prune both specifications and violations separately. Validation of mined specifications happens either again on execution traces or via static verification. All dynamic detectors report relatively few findings, compared to the static detectors. Therefore, evaluations simply consider all their findings and none of the approaches presents a ranking strategy.

A strengths of dynamic detectors is that they can verify whether a misuse is problematic in the sense that it may cause an exception. This allows a significant reduction of false positives (98.2% in the case of SPECHECK) or even their complete exclusion (in case of PG12). Since reducing false positives is important for developers to adopt misuse detectors [FLL⁺02, BBC⁺10, JS13] and many misuses cause exceptions (see Section 2.2), this is an important achievement. However, there are many misuses that do not cause exceptions, but nonetheless have severe consequences, such as vulnerabilities

(see Section 2.4) and should not be neglected.

A major limiting factor for the usefulness of dynamic detectors is the availability of exhaustive execution traces [PBG10]. Similar to static detectors, specifications need to manifest in sufficiently many traces in order to be mined. In addition, if a misuse is not triggered in any execution, it cannot be detected through verification of execution traces. Most dynamic detectors rely on existing test harnesses—which is generally unreliable—or manually curated training datasets—which is not scalable. The exception is PG12, which uses random test generation to maximize the trace coverage. However, to make this scale, PG12 focuses on code parts that call methods from a pre-specified set of APIs.

An inherent limitation of all current dynamic approaches is that they do not consider usage elements other than method calls. Method-call traces do not show whether checks happened on the result of a call or, more generally, which subsequences of a trace is control-dependent on which preceding calls. Therefore, the detectors cannot analyze control dependencies and exception handling, which make up for a significant part of all API misuses (see Chapter 3).

Another limitation is that current dynamic approaches can only search misuses of a predefined set of APIs. This set may be defined explicitly [PG12] or implicitly, e.g., as all APIs from a certain package [PJAG12] or library [LKL08, GS10, NK11]. They need this, in order to decide whether a method call `m()` should itself become an event in the call trace or whether the execution of `m()` should be tracked to potentially add transitive calls to the trace. Therefore, dynamic detectors usually focus on detecting misuses of widely used APIs, e.g., from the `JAVA CLASS LIBRARY`, and neglect less-commonly-used APIs and project-specific APIs.

4.3.3. Coverage of the Problem Space

Table 4.1 shows that existing detectors cover only a small subset of all API-misuse categories. While all detectors can, to some degree, identify missing method calls, only four detectors can identify missing conditions, only four can identify missing iterations in some cases, and only two can identify missing exception handling. None of the detectors targets all of these categories. Only three of the detectors can identify any redundant usage elements—method calls, in all cases. All other detectors cannot identify such elements, by design of their usage representation and violation-detection strategy. Table 3.1 shows that the redundant violation type is much less common than the missing violation type, which might, in part, explain this design choice. Overall, these findings suggests that future work should develop detectors that target the neglected misuse categories or cover more misuse categories at once or both. Part III of this thesis presents such a detector.

4.3.4. Evaluation

Table 4.2 summarizes the empirical results of the surveyed detectors, as reported in their original papers. Detectors are evaluated on one to twenty projects (average 5.7; median

4. A Survey of API-Misuse Detectors

five). The concrete projects samples are all distinct and mostly even disjoint, except in the evaluations of dynamic detectors that chose target projects from the DACAPO benchmark suite [BGH⁺06].

Most evaluations apply detectors to projects individually. In this setting, the detectors learn project-specific patterns and identify respective violations, assuming that correct usages occur frequently within projects. If some API is, for example, only used once in a project, the detectors cannot determine whether this usage is correct or not, even if that particular usage is obviously incorrect. The exceptions to this are (1) JADET, which was also evaluated in a multi-project setting where it was trained on usages from 6,097 projects to detect misuses in 20 of these projects, (2) CAR-MINER and ALATTIN, which mine patterns from examples retrieved via a code search engine, (3) RRFINDER, which was trained on a manually curated dataset of correct usage examples, and (4) PJAG12, which was trained on a public dataset of execution traces. While JADET shows a slightly better performance in the multi-project setting, the evaluations indicate no general superiority of either approach.

To assess the detection performance, most authors review the top-X findings of their detectors in their experiments, where X is either a fixed number or a percentage (sometimes 100%, i.e., all findings). They then either present anecdotal evidence of true positives or measure the precision of detectors. Many evaluations also present additional categories of findings, such as code smells, to distinguish false positives from other non-misuse findings that may still be valuable to developers. The definitions of when a finding belongs to which category—if provided—differ between publications, even if they use the same label, e.g., “bug” or “code smell.”

We argue that reviewing a fixed number of findings more realistically mirrors the end-user scenario, where a developer is presented with a list of findings for review. Developers are unlikely to review an arbitrarily large number of findings, especially if many of them are false positives [FLL⁺02, BBC⁺10, JS13]. It is more likely that they focus on the findings that are presented to them first. As a fixed number, authors reviewed 10-60 findings (average 24; median 10). Considering that many detectors present a significantly larger number of findings, it becomes important that detectors effectively rank more severe findings before others, e.g., crash bugs before code smells.

Precision is an important metric for API-misuse detectors, since large numbers of false positives is one of the main reasons why developers reject code-analysis tools [FLL⁺02, BBC⁺10, JS13]. However, we are also interested in the detectors’ recall, since our conceptual assessment in Table 4.1 suggests that detectors miss many misuses. We argue that it is important for developers to know which kinds of mistakes a detector misses, in order to decide about additional quality measures. Past studies widely ignored the recall of API-misuse detectors. One exception is LKL08, whose recall was measured with respect to four known bugs in a single target project. Further exceptions are PJAG12 and SALENTO, whose recall was measured with respect randomly mutated usages. No study in our survey measured recall with respect to a larger quantity of real-world usages.

Overall, the selection of different target-project samples and review samples sizes and the varying definitions of finding categories make a direct comparison of the detectors, solely based on their reported empirical results, unreliable. The only concrete comparison

between detectors is presented by Pradel *et al.* [PJAG12], who compare the findings of PJAG12 to those of GROUMINER and TIKANGA and who discuss which misuses are identified across detectors and which only by one of them. This calls for more standardized evaluation of misuse detectors and for putting more effort into the cross-comparison of detectors. Part II of this thesis such an evaluation and comparison.

Part II.

MuBench

A Systematic Evaluation of Static API-Misuse Detectors

To mitigate API misuse, researchers have proposed several *API-misuse detectors* that are able to find misuses through static analysis. These detectors commonly analyze *API usages*, i.e., code snippets that use a given API. They mine *usage patterns*, i.e., equivalent API usages that occur frequently, and then report deviations from these patterns as potential misuses. The detectors’ underlying techniques differ, especially with respect to how they encode API usages and frequency, as well as in how they identify patterns and violations thereof. Chapter 4 of this thesis presents a survey and qualitative comparison of these detectors. In this part, we compare them quantitatively.

Our survey in Chapter 4 shows that previous empirical studies generally evaluated detectors on different sets of target projects, such that the respective results are hardly comparable. In many cases the exact versions of the projects are not reported or became unavailable, which makes it impossible to reproduce results and to evaluate other detectors on the same project versions. Moreover, we find that all studies focus on the precision of detectors. Precision is an important metric, since large numbers of false positives is one of the main reasons why developers do not use code-analysis tools [FLL⁺02, BBC⁺10, JS13]. However, we are also interested in the detectors’ recall, since our conceptual assessment in Chapter 4 suggests that detectors miss many misuses. We argue that it is important for developers to know which kinds of mistakes a detector misses, in order to decide about additional quality measures.

Comparing different misuse detectors with respect to both their precision and recall is a challenging task, due to the different underlying mechanisms and representations used by each detector. Moreover, it is very difficult to design a unified evaluation setup that fairly compares both static and dynamic techniques, without resorting to compare apples and oranges, since both generally require different input (code examples vs. execution traces). Therefore, in this part of the thesis, we focus on static API-misuse detectors.

To achieve a reproducible empirical comparison of static misuse detectors, we build MUBENCH, the first automated benchmark for Java-API-misuse detectors (Chapter 5). We integrate the four state-of-the-art misuse detectors JADET, GROUMINER, DMMC, and TIKANGA into the benchmark (Section 5.1). We exclude the other eight static detectors that we identified in our survey in Chapter 4, because two rely on the discontinued Google Code Search, five target C/C++ code, and one targets Dalvik Bytecode, while our benchmark contains Java misuses.⁶ We use the misuse examples from Chapter 2 to create a ground-truth dataset with 64 misuses from 13 real-world projects (Section 5.2).

⁶ Note that we selected these detectors in 2016, based on the results of the first round of our survey. Therefore, the detectors that were published later were not initially considered for the benchmark.

Then we design three experiments, to measure both the detectors’ precision and recall:

Experiment P (Section 5.3) measures the precision of the detectors in a per-project setting, where they mine patterns and detect violations in individual projects from MUBENCH. This is the same setting we find in most existing empirical evaluations (see Table 4.2).

Experiment RUB (Section 5.4) determines upper bounds to the recall of the detectors with respect to the known misuses in MUBENCH. We take the possibility of insufficient training data out of the equation, by providing the detectors with crafted examples of correct usages for them to mine required patterns.

Experiment R (Section 5.5) measures the recall of the detectors against both the misuses from the MUBENCH dataset and the detectors’ own confirmed findings from Experiment P in a per-project setting.

In Chapter 6, we present the MUBENCHPIPE, a pipeline that automates all parts of the evaluation process, except for a manual review of the subset of a detector’s findings that may potentially identify a misuse. MUBENCH pre-filters such candidates based on misuse locations, to reduce the manual effort of evaluations and ease reproduction of benchmark experiments. Moreover, MUBENCHPIPE enables full traceability of experiments and review decisions through a ready-to-use artifact website.

In Chapter 7, we use MUBENCH to empirically evaluate and compare the four state-of-the-art misuse detectors that we integrated. Our quantitative results show that these detectors are practically capable of detecting misuses, when provided with correct usages to mine patterns from, i.e., when they successfully mine the respective patterns. However, they suffer from extremely low precision and recall in a realistic setting. We identify four root causes for false negatives and seven root causes for false positives. Most importantly, to improve precision, detectors need to go beyond the naive assumption that a deviation from the most-frequent usage corresponds to a misuse, for example, by building probabilistic models to reason about the likelihood of usages in their respective context. To improve recall, before all else, detectors need to learn better models of API usage, for example, by obtaining more correct usage examples from different sources, such as code-search engines, and considering program semantics, such as type hierarchies and implicit dependencies between API usages. These novel insights are made possible by MUBENCH. Our empirical results present a wake-up call, unveiling serious practical limitations of tools and evaluation strategies from the field, especially with respect to detectors’ recall, which is typically not evaluated, and the application of detectors to individual projects, which do not seem to give them sufficient data to learn good models of correct API usage.

In Chapter 8, we discuss the extensibility and reusability of MUBENCH. The benchmark eases the integration of new misuse examples and of further misuse detectors, to increase generalizability of experiment results and encourage cross-detector comparison. We demonstrate extensions to the dataset, e.g., from the findings of a study of

runtime-verification techniques [LHX⁺16] and integrate the static bug finder FINDBUGS⁷ to compare its capabilities to those of API-misuse detectors.

In Chapter 9, to conclude this part of the thesis, we provide an overview over related work on both benchmarking bug-detection tools and evaluating API-misuse detectors.

⁷<http://findbugs.sourceforge.net/> (checked on Mar 20, 2018)

5. Evaluation Setup

In this chapter, we describe the evaluation setup we use to empirically compare the capabilities of API-misuse detectors. We design three experiments, to measure both the detectors' precision and recall. As the basis of these experiments, we assemble a ground-truth dataset from the misuse examples we identified in Chapter 2. This enables us to compare all detectors on the same target projects and with respect to the same known misuses.

5.1. Subject Detectors

To evaluate detectors, ideally, we use the exact same implementations as were used for the original evaluations presented in the respective publications. Compared to reimplementations, this avoids bias that might come from misunderstandings of the approaches or diverging decisions with respect to implementation details that are not described in the publications. Moreover, it reduces the effort to obtain implementations of multiple detectors. To obtain the original implementations, we contact the authors of existing API-misuse detectors and ask them to provide the prototypes they used in the original empirical evaluations. We focus on misuse detectors for Java APIs, in order to evaluate them using the examples of Java-API misuse we presented in Chapter 2. Our survey in Chapter 4 identifies seven such detectors.¹ We contacted the respective authors and got responses from all of them:

- The authors of JADET [WZL07], GROUMINER [NNP⁺09b], DMMC [MBM10], and TIKANGA [WZ11] provided their implementations and assisted us in setting them up for execution on our benchmark.
- The authors of DROIDASSIST [NPVN15] also provided their implementation, but we found that it supports only Dalvik Bytecode,² while the examples in our dataset originate from general Java projects, which compile to Java Bytecode. Therefore, we exclude DROIDASSIST from our experiments.
- The authors of ALATTIN [TX09a] and CAR-MINER [TX09b] informed us that we can no longer run their respective implementations, because they both depend

¹Note that this bases on the results of the first round of our survey in 2016. Meanwhile, new detector were published.

²A bytecode format developed by Google, which is optimized for the characteristics of mobile operating systems (especially for the Android platform).

5. Evaluation Setup

on Google Code Search to retrieve usage examples, a service that is no longer available.³

This leaves us with the four detectors JADET, GROUMINER, DMMC, and TIKANGA.

5.2. API-Misuse Dataset

Our goal is to empirically evaluate and compare API-misuse detectors. To this end, we need a dataset with targets to run all these detectors on. We use the misuse examples that we identified in Chapter 2 as a starting point to create such a ground-truth dataset, turning each example into a candidate for inclusion as follows.

For the misuses we identified in real-world projects (see Section 2.2 and Section 2.4), we locate the actual occurrences from the respective codebases. For each misuse, we manually identify the respective project’s version-control system and extract the revision Id of a project version that contains the misuse, as well as the path of the file and signature of the method that the misuse occurs in, and the name of the API that is misused. In addition, if the misuse was fixed in the project history, we extract the revision Id of the version immediately after the fix. Finally, we assemble a description from the respective issue report, the fix, and the API documentation. In total, we obtain such data about 103 misuses from 49 projects. Note that we had to exclude the misuse examples from the ASPECTJ project, as identified by the IBUGS dataset, at this point, because we could not locate the version-control revisions designated in the dataset in the project’s version-control system, which has been migrated since the release of IBUGS.

For the misuses we identified through the developer survey (see Section 2.3) and the misuses we identified in studies on API-usage directives (see Section 2.1), we manually create respective code examples and add them to our dataset. We aim to create minimal-yet-realistic examples, as they might appear in a real codebase, if the respective usage has been factored out into a single method. We do not create any call sites for the methods containing the misuses, since any additional code beyond the API usage in question would be arbitrary and, thus, might introduce bias. As for the misuses from real-world projects, we create a description of each of the misuse examples from the survey. This gives us an additional 28 misuse examples.

Following this process gives us in total 131 misuse examples as candidates for inclusion in our ground-truth dataset. For each candidate, we now possess a respective occurrence in source code. However, while GROUMINER works on source code, JADET, TIKANGA, and DMMC require Java Bytecode as input. Thus, we can only compare these detectors on misuse examples for which we have both source code and Bytecode.

Since Bytecode is not readily available for most of our misuse examples, we resort to compiling them ourselves. For the hand-crafted examples this is relatively easy, since the examples are small and we can easily create a minimal GRADLE build configuration to compile them. For the examples from real-world projects, we need to compile the

³ <https://google-opensource.blogspot.com/2015/03/farewell-to-google-code.html> (checked on Mar 09, 2018)

respective project versions that contain the misuses, by adding necessary build files and fixing any dependency issues. Unfortunately, this task cannot be entirely automated. To make it at least reproducible, we manually determine a sequence of commands that compile a project version starting from a clean checkout. To bound the manual effort in determining these commands, we define a procedure that we follow for each project version:

1. If the project version uses a build configuration, such as ANT, GRADLE, or MAVEN, we add a respective compilation command and run it.
2. If running the build fails for local reasons, such as a build directory that needs to be created, we add respective preparation commands to resolve the problems.
3. If running the build fails because the configuration is outdated or erroneous, we try to fix it and add respective transformation commands. If we cannot fix the configuration within one day, we continue with the next step.
4. If there is no build configuration or we cannot get it to run, we try to manually construct a minimal GRADLE configuration to compile the project version. If we cannot construct a configuration within one day, we exclude the project version from our dataset.
5. If running the build fails because required dependencies are not included in the project version and cannot automatically be resolved, we manually search for the respective JAR files and add them to a dedicated Maven repository for use in the benchmark compilation. If we cannot manually resolve some dependencies within one hour, we manually create stubs for the types required by the project version, if this requires to stub at most ten types. Otherwise, we exclude the project version from our dataset.
6. If the project version contains compilation errors with an obvious fix, we add respective transformation commands to fix them. For example, in one project version, an instance of `java.security.CodeSource` is created with `null` as the second constructor argument. With Java 1.4 this was valid, because `CodeSource` had only a single constructor. However, Java 1.5 introduced a second constructor, which makes the call ambiguous. To fix this problem, we insert a cast of `null` to `java.security.CodeSignature[]`, the type it was implicitly cast to before. If we cannot fix all compilation problems after one day or we encounter an error without an obvious fix, we exclude the project version from our dataset.

Following this process we obtain 29 compilable versions of 13 projects. We have to exclude 29 versions of 19 projects, because 22 versions contain compile errors that we cannot fix and seven versions miss dependencies that we cannot resolve. Furthermore, we have to exclude 17 versions from as many ANDROID apps, because ANDROID has its own bytecode format, DALVIK,⁴ which we cannot feed to misuse detectors that read Java

⁴ [https://en.wikipedia.org/wiki/Dalvik_\(software\)](https://en.wikipedia.org/wiki/Dalvik_(software)) (checked on Nov 16, 2017)

5. Evaluation Setup

Table 5.1.: Datasets Used Throughout Part II of This Thesis, with the Number of Hand-crafted Misuses (#HM), the Number of Real-world Projects (#P), the Number of Project Versions (#PV), the Number of Misuses in These Project Versions (#PVM), and the Total Number of Misuses (#M). “n/a” Denotes that the Number Is Irrelevant for the Use of the Dataset.

	Dataset	#HM	#P	#PV	#PVM	#M
1	Experiment P	n/a	5	5	n/a	n/a
2	Experiment RUB	25	13	29	39	64
3	Experiment R	0	13	29	53	53

Bytecode. We proceed likewise for three misuses examples from the developer survey that are specific to the ANDROID standard library.

In the end, we have 64 misuses in total for our experiments; 39 misuses from the 29 versions of 13 real-world projects and 25 hand-crafted examples. Note that some project versions contain multiple misuses. Table 5.1 describes the subsets of this dataset that we use in the individual experiments. We publish the dataset⁵ for others to use in future studies.

5.3. Experiment P

We design Experiment P to assess the precision of detectors.

Motivation Past studies show that developers rarely use analysis tools that produce many false positives [FLL⁺02, BBC⁺10, JS13]. Therefore, for a detector to be adopted in practice, it needs a high precision.

Setup To measure precision, we follow the most-common experimental setting we found in the literature (cf. Table 4.2), i.e., the per-project setting. In this setting, detectors mine patterns and detect violations on a per-project basis. First, we run detectors on individual project versions. Second, we manually validate the top-20 findings per detector on each version, as determined by the respective detector’s ranking strategies. We limit the number of findings, because it seems likely that developers would only consider a fixed number of findings, rather than all of a potentially very large number of findings. Hence, the precision in a detector’s top findings is likely crucial for tool adoption. Also, we need to limit the effort of reviewing findings of multiple detectors on each project version.

Dataset Since manually reviewing findings of all detectors on all project versions is infeasible, we sample five project versions. To ensure a fair selection of projects, we first run all detectors on all project versions. For practical reasons, we timeout each detector on an individual project version after two hours. The run statistics are summarized in

⁵<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

Table 5.2.

JADET and TIKANGA fail on one project version and DMMC fails on four project versions, since the Bytecode contains constructs that the detectors’ respective Bytecode toolkits do not support. GROUMINER times out on eight project versions and produces an error on one other version. We exclude any project version where a detector fails.

For the remaining 15 versions, we observe that the total number of findings correlates across detectors. Table 5.3 shows that the pairwise correlation (Pearson’s r) is strong (≥ 0.75) or medium (≥ 0.5) for all pairs of detectors, except for JADET and GROUMINER ($r = 0.49$). This means that either all detectors report a relatively large or a relatively small number of findings on any given project version. We hypothesize that the total number of findings might be related to the detectors’ ability to precisely identify misuses in a given project version. Therefore, we sample project versions according to the average normalized number of findings across all detector. We normalize the number of findings per detector on all project versions by the maximum number of findings of that detector on any project version. We sample the two projects with the highest average normalized number of findings across all detectors (CLOSURE⁶ v319 and iTEXT⁷ v5091) and the two projects with the lowest average normalized number of findings across all detectors (JMRTD⁸ v51 and JODA-TIME⁹ v1231). Additionally, we randomly select one more project version (APACHE LUCENE¹⁰ v1918) from the remaining projects, to cover the middle ground. Note that we select at most one version from each distinct project, because different versions of the same project may share a lot of code, such that detectors are likely to perform similarly on them. This dataset for Experiment P is summarized in Row 1 of Table 5.1.

Metrics We calculate the precision of the detector, i.e., the ratio between the number of true positives over the number of findings.

Review Process Two authors independently review each of the top-20 findings of the sampled project versions and mark it as a misuse or not. To determine this, they consider the logic and the documentation in the source code, the API’s documentation, and its implementation if publicly available. After the review, any disagreements between the reviewers are discussed until a consensus is reached. We report Cohen’s Kappa score as a measure of the reviewers’ agreement. Note that we follow a lenient reviewing process. For example, assume a usage misses a check `if (Iterator.hasNext())` before calling `Iterator.next()`. If the detector finds that `hasNext()` is missing, we mark the finding as a hit, even though this does not explicitly state that the call to `next()` should be guarded by a check on the return value of `hasNext()`. This follows our intuition that such findings may still provide a developer with a valuable hint about the problem.

⁶<https://developers.google.com/closure/compiler/> (checked on Feb 24, 2017)

⁷<https://sourceforge.net/projects/itext/> (checked on Feb 24, 2017)

⁸<http://jmrttd.org/> (checked on Feb 24, 2017)

⁹<http://www.joda.org/joda-time/> (checked on Feb 24, 2017)

¹⁰<https://lucene.apache.org/core/> (checked on Feb 24, 2017)

5. Evaluation Setup

Table 5.2.: Number of Findings per Detector on All Compilable Project Versions in MUBENCH. Detectors timeout after two hours. Experiment P includes the two projects with the highest number of findings (\sqcup), the two projects with the lowest number of findings (\sqcap), and one randomly selected project (\circ).

Project	Version	Number of Findings					Sample Criterion
		JADET	GROUMINER	TIKANGA	DMMC	Norm. Avg.	
APACHE COMMONS LANG	587	0	28	0	157	0.06	
APACHE COMMONS MATH	998	17	error	17	686	0.20	
ADEMPIERE	1312	0	27	0	116	0.05	
ALIBABA DUID	e10f28	17	timeout	5	520	0.13	
CLOSURE	114	113	101	24	1233	0.49	
CLOSURE	319	176	126	45	1945	0.74	\sqcup
CLOSURE	884	71	167	33	1966	0.63	
APACHE HTTPCLIENT	302	0	12	0	114	0.03	
APACHE HTTPCLIENT	444	0	15	0	110	0.03	
APACHE HTTPCLIENT	452	0	12	0	113	0.03	
ITEXT	5091	17	198	55	1138	0.55	\sqcup
APACHE JACKRABBIT	1601	12	186	22	error	0.41	
APACHE JACKRABBIT	1678	0	15	0	error	0.03	
APACHE JACKRABBIT	1694	13	186	22	error	0.41	
APACHE JACKRABBIT	1750	10	timeout	8	434	0.12	
JFREECHART	103	167	timeout	88	673	0.69	
JFREECHART	164	168	timeout	90	664	0.69	
JFREECHART	881	194	timeout	93	745	0.76	
JFREECHART	1025	194	timeout	93	747	0.76	
JFREECHART	2183	190	timeout	100	906	0.81	
JFREECHART	2266	195	timeout	102	913	0.82	
JMRD	51	0	11	0	29	0.02	\sqcap
JMRD	67	0	10	0	35	0.02	
JODA-TIME	1231	0	0	0	1	0.00	\sqcap
APACHE LUCENE	207	0	140	0	182	0.20	
APACHE LUCENE	754	0	54	0	265	0.10	
APACHE LUCENE	1251	2	62	0	error	0.11	
APACHE LUCENE	1918	2	88	4	583	0.20	\circ
MOZILLA RHINO	286251	error	55	error	257	0.20	

Table 5.3.: Correction of the Number of Findings per Project Version For All Pairs of Detectors (Pearson’s r). Strong correlation ($r \geq 0.75$) in **bold**. Medium correlation ($r \geq 0.5$) in *italic*.

	JADET	GROUMINER	DMMC	TIKANGA
JADET	1.00			
GROUMINER	0.49	1.00		
DMMC	0.85	0.78	1.00	
TIKANGA	<i>0.70</i>	0.82	0.88	1.00

5.4. Experiment RUB

We design Experiment RUB to assess the detection capabilities of our subject detectors, i.e., to measure an upper bound to their recall under the assumption that they always mine the required pattern.

Motivation We argue that it is important for developers to know which misuses a particular tool may or may not find, in order to decide whether the tool is adequate for their use case and whether they must take additional measures. Moreover, it is important for researchers to know which types of misuses existing detectors may identify, in order to direct future work. Therefore, we measure detectors’ recall while providing sufficiently many correct usages that would allow them to mine the required pattern.

Dataset For this experiment, we use all compilable project versions from the MUBENCH dataset with the respective known misuses, as well as the hand-crafted misuse examples. This dataset for Experiment RUB is summarized in Row 2 of Table 5.1.

Setup Recall that all our subject detectors mine patterns, i.e., frequently reoccurring API usages, and assume that these correspond to correct usages. They use these patterns to identify misuses. Recall further that each detector has a distinct representation of usages and patterns and its own mining and detection strategies. If a detector fails to identify a particular misuse, this may be due to (1) an inherent limitation of the detector, e.g., because it cannot represent some usage element such as conditions, or (2) a lack of examples of respective correct usage for pattern mining, i.e., a limitation of the training data. With Experiment RUB, we focus on (1), i.e., we take (2) out of the equation and assess the detectors’ general ability to identify misuses. To this end, we provide the detectors with sufficiently many examples of correct usage corresponding to the misuses in question. This guarantees that they could mine a respective pattern. If the detector is unable to identify a misuse in this setting, we know the problem lies with the detector itself.

We manually create a correct usage for each misuse in the dataset. For the misuse examples from the real-world projects, we derive the correct usages from the fix that we find in the respective project’s version-control history, if one exists. While gathering candidates for inclusion in our benchmark dataset in Section 5.2, we identified these

5. Evaluation Setup

fixes at the granularity of a commit. However, we find that these commits often contain many unrelated changes, such as refactorings or reformatting, in addition to the actual fix. To make comparison of the usage before and after the fix easier, we manually reduce the changeset to only those changes required to turn the misuse into a correct usage and apply only this reduced changeset to the code before the fix. Then we remove from the fixed code anything that does not impact the usage in question, such as other usages with not data or control dependencies to the usage in question. For the hand-crafted misuses examples derived from API usage directives (see Section 2.1) and the developer survey (see Section 2.3), we derive the correct usages by following the directives that we knowingly violated and from the problem descriptions of the survey respondents, respectively. As for the misuse examples themselves, we aim to create minimal-yet-realistic examples of correct usage. To this end, we take the misuse examples and apply only the necessary modifications to turn them into correct usages, just as a developer fixing the misuse might proceed. We store the code of this *crafted correct usage* in our dataset.

In the experiment, we run each detector once for each individual known misuse in the dataset. In each run, we provide the detector with the file that contains the known misuse and with 50 copies of the respective crafted correct usage. We ensure that the detector considers each copy as a distinct usage. We configure the detectors to mine patterns with a minimum support of 50, thereby ensuring that they mine patterns only from the code in the crafted correct usage. We chose 50 as a threshold, since it is high enough to ensure that no detector mines patterns from the code in the file with the misuse.

Metrics We calculate two numbers for each detector. The first is its *conceptual recall upper bound*, which is the fraction of the known misuses in the dataset that match its capabilities from Table 4.1. Note that the conceptual recall upper bound is calculated offline, without running any experiments. The second is the detector’s *empirical recall upper bound*, which is the fraction of misuses a detector actually finds from all the known misuses in the dataset. An ideal detector should have an empirical recall upper bound equal to its conceptual recall upper bound. Otherwise, its practical capabilities do not match its conceptual capabilities. In such cases, we investigate the root causes for such mismatches. Note that we use the term “upper bound,” because neither metric reflects the detectors’ recall in a setting without guarantees on the number of correct usages for mining.

Review Process To evaluate the results, we review all *potential hits*, i.e., findings from each detector that identify violations in the same files and methods as known misuses. Two authors independently review each such potential hit to determine whether it actually identifies one of the known misuses. If at least one potential hit identifies a misuse, we count it as a *hit*. After the review, any disagreements between the reviewers are discussed until a consensus is reached. We report Cohen’s Kappa score as a measure of the reviewers’ agreement. We follow the same lenient review process as for Experiment P.

5.5. Experiment R

We design Experiment R to assess the recall of detectors.

Motivation While Experiment RUB gives us an upper bound to the recall of misuse detectors, we also want to assess their actual recall where we do not provide them with correct usages ourselves. Due to the lack of a ground-truth dataset, such an experiment has not been attempted before in any of the misuse-detection studies we surveyed in Chapter 4.

Dataset As the ground truth for this experiment, we use all known misuses from real-world projects in MUBENCH plus the true positives identified by any of the detectors in Experiment P. This means that Experiment R not only evaluates recall against the misuses of MUBENCH, but also practically cross-validates the detector capabilities against each other. We exclude the hand-crafted misuse examples from this experiment, since there is no corresponding code for the detectors to mine patterns from. The dataset we use for Experiment R is summarized in Row 3 of Table 5.1.

Setup We run all detectors on all projects versions individually, i.e., we use the same per-project setup as for Experiment P.

Metrics We calculate the recall of the detectors, i.e., the number of actual hits over the number of known misuses in the dataset.

Review Process We review all potential hits in the same process as for Experiment RUB. This gives us the detectors' recall with respect to a large number of known misuses from MUBENCH.

5.6. Limitations

Ideally, our experiments would include thousands of misuses from a large number of projects and in each individual project version, to give us greater confidence in the generalizability of our benchmark. However, currently, there is no such dataset. We invested several months of effort to collect and prepare the dataset in its current state, to make a first step towards such a dataset. Now that we have the infrastructure in place, it is straightforward to extend the dataset with misuse examples from different sources.

Our benchmark dataset is subject to the limitations of how we identified examples of API misuse in the wild (see Section 2.6). The benchmark dataset may not be representative for API misuses in the wild, especially, because we could only compile 29 (50%) of the project versions with misuses and, therefore, had to exclude 38 (37.2%) of the misuses, which appear in the other versions, from our experiments (see Section 5.2). Compiling arbitrary versions of projects from the source control history of project is a challenging task. The author of this thesis invested two full weeks and additional three months work of a student, to include as many project versions as possible. Still, losing the examples for which we could not compile the respective project versions may bias

5. Evaluation Setup

the benchmark. For example, it may be that the projects we could compile generally adhere to higher quality standards and that, therefore, the misuses they contain are more complex. Conversely, it is possible that they adhere to lower quality standards, since they contain relatively more misuses. Future work may mitigate this limitation by adding further project versions and misuse examples to the dataset. To make this as easy as possible, we build our benchmarking pipeline agnostic to the concrete dataset, such that the benchmarking experiments automatically consider new additions to the dataset.

The concrete misuses instances we include in our dataset may not be representative for all instances of the same misuse. For example, a certain misuse might often be distributed over multiple methods, while our dataset contains only occurrences in a single method or vice versa. This threat will likely become smaller, as the dataset increases in size.

We did not design an experiment to measure an upper bound to the *precision* of detectors. To measure the upper bound of recall, we provide detectors with the misuses and the corresponding fixed usages and check whether they can detect the difference. To measure the upper bound of precision, however, we would have to provide detectors with a *minimal* version of the fixed usage, because every additional element may cause false positives. It is unclear how to construct such minimal examples.

6. Experiment Pipeline

Following the idea of automated bug-detection benchmarks for C programs, such as BUG-BENCH [LLQ⁺05] and BEGBUNCH [CHK⁺09], we facilitate the task of benchmarking multiple detectors on our misuse dataset with an automated experiment pipeline. We call this pipeline MUBENCHPIPE. The goal of MUBENCHPIPE is (1) to automate as much as possible of the experimental setup presented in Chapter 5, to reduce the effort of evaluating API-misuse detectors, and (2) to make benchmarking experiments reproducible and extensible. The pipeline also enables adding new detectors to the comparison, as well as benchmarking with different or extended datasets, in the future. We publish MUBENCHPIPE¹ for future studies.

6.1. Representation

To enable automated processing of our dataset, we store it in the data schema depicted by Figure 6.1. Each misuse example that we collected appears in a particular project. More specifically, we identified each misuse in one particular version of a project, i.e., in one particular version-control revision. Usually, the same misuse also exists in other versions of the project, i.e., in all revisions from its introduction up until the revision immediately before it got fixed. To be able to count distinct misuses both per project version and per project, we represent all three as separate entities in our dataset, where a project has one or more versions and contains one or more misuses, and each of the project’s versions contains one or more of the project’s misuses. For the hand-crafted misuse examples, we create artificial project named **synthetic** and distinct project versions that contain the source code of one crafted example each.

For each project, the dataset records the project name, the project website, and the project repository. We use the repository to uniquely identify a project, because this allows us to uniquely identify project versions using their respective revision Id.

For each project version, the dataset records the revision Id, the relative path to the source files and class files (after build), as well as the list of build commands obtained in Section 5.2. This information suffices to checkout the project versions, compile them, and provide the source code or Bytecode or both to a misuse detector.

For each misuse, we store the source that we identified it from, the description of the misuse, its location in the project version’s source folder, and a reference to its fix. Listing 6.1 shows an example of such metadata, using the YAML format.²

¹<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

² <http://yaml.org/> (checked on Nov 24, 2017)

6. Experiment Pipeline

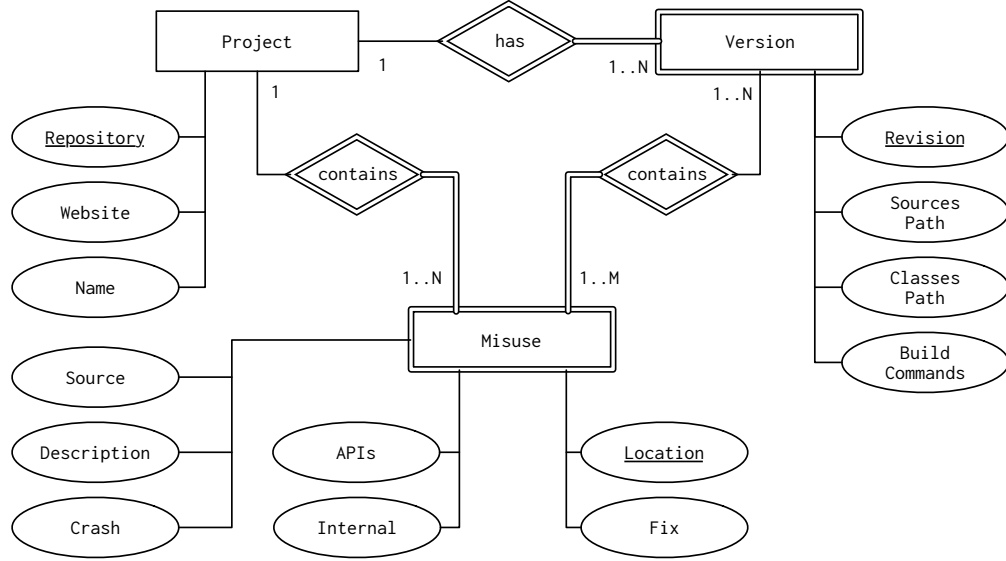


Figure 6.1.: Structure of the API-Misuse Dataset

The **source** designates where we found the misuse example. The **name** either identifies one of the existing datasets we analyzed (Section 2.2), the JCA usages or JCA fixes study (Section 2.4), our developer survey (Section 2.3), or the publication we took the misuse example from. We provide a **url** to the original dataset or publication, if applicable. The **report** designates a ticket—usually in an issue tracker—reporting the misuse.

The metadata contains a textual **description** of the misuse and its fix. Furthermore, we document whether the report indicates that the misuse may cause a **crash**, list all types that are involved in either the misuse or the respective correct usage (**api**), and document whether any of these types is declared in the project containing the misuse or whether they are declared by some dependency (**internal**).

The **location** points to the source-code location of the misuse in the project. We report the Java source **file** and the signature of the enclosing **method**. We use this information to uniquely identify a misuse. Additionally, we provide the **revision** Id for the respective **fix** and a URL to the fixing **commit**, if available from the report.

6.2. Benchmark Automation

MUBENCHPIPE automates many of our evaluation steps including retrieving and compiling the project versions' source code, running detectors, collecting their findings, and performing the manual reviews of potential hits. It provides a command-line interface to control these tasks. We subsequently describe the pipeline steps we implemented to facilitate our evaluation and to enable easy replication of our experiments.

```

1  source:
2    name: BugClassify
3    url: https://www.st.cs.uni-saarland.de/softevo/bugclassify/
4    report: https://bugzilla.mozilla.org/show\_bug.cgi?id=286251
5  description: >
6    IRFactory.initFunction() is called twice along one possible execution path,
7    which causes an infinite loop. This is fixed by removing the second call.
8  crash: yes
9  api:
10    - org.mozilla.javascript.IRFactory
11  internal: yes
12  location:
13    file: org/mozilla/javascript/Parser.java
14    method: function(int)
15  fix:
16    commit:
17      https://github.com/mozilla/rhino/commit/ed00a2e83de1e768918604a65def097895b71dd4
18    revision: ed00a2e83de1e768918604a65def097895b71dd4

```

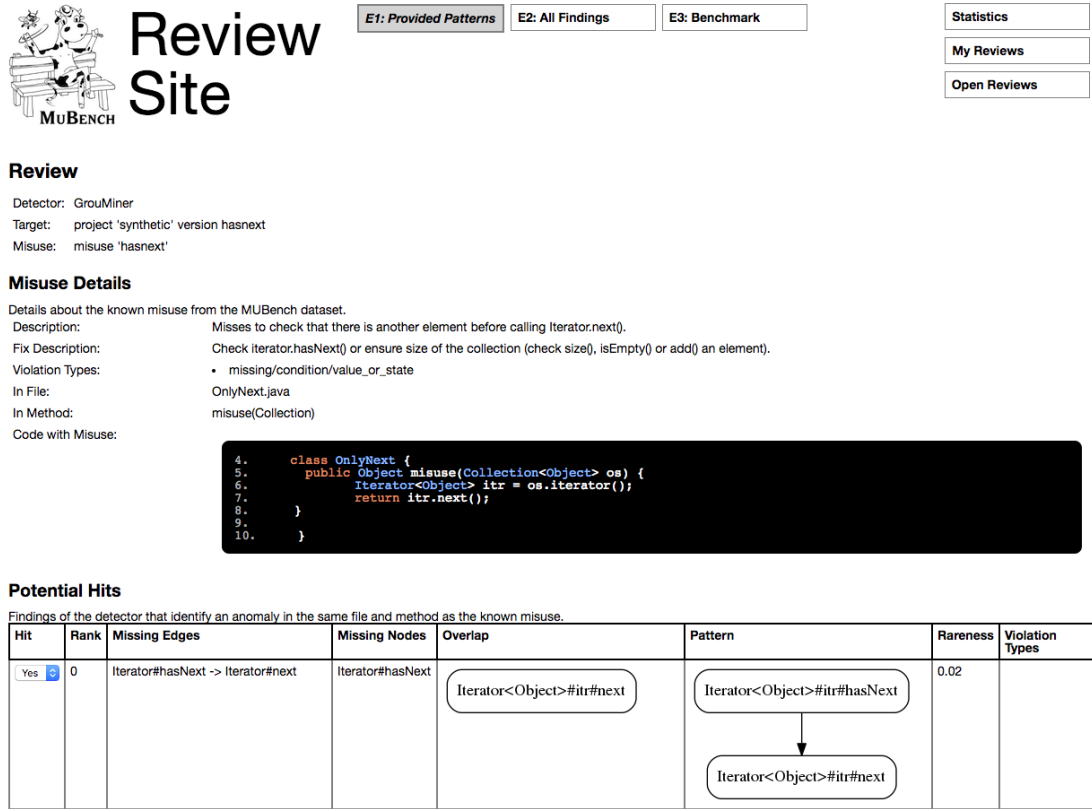
Listing 6.1.: Metadata of an API Misuse from the RHINO Project.

Checkout MUBENCHPIPE uses the recorded repository URL and commit Id from the dataset to obtain the source code of the respective project version. It supports SVN and GIT repositories, source archives (zip), as well as a special handling for the hand-crafted examples in the dataset.

Compile For every project version, MUBENCHPIPE first copies the entire project source code, the individual files containing known misuses, and the respective crafted correct usages for Experiment RUB each into a separate folder. Then, it uses the respective build configuration from the dataset to compile all Java sources to Bytecode. During compilation, MUBENCHPIPE captures all build dependencies from any MAVEN, GRADLE, and ANT execution and copies them into a dedicated folder. After compilation, it copies the entire project Bytecode, the Bytecode of the individual files containing known misuses, and the Bytecode of the respective crafted correct usages each into a separate folder. From the Bytecode of the entire project it additionally creates a Java archive (jar) file. This way, the pipeline can provide the detectors with the source code and Bytecode of each of these project parts individually, as well as with class paths that contains all respective dependencies.

Detect For each detector, we build a *runner* to have a unified command-line interface for all detectors. These runners invoke the detectors with the best configuration reported in the respective publication. To run a detector on a project version, MUBENCHPIPE invokes the detector’s runner with the paths to the respective source code, Bytecode, and dependencies. The runner uses these paths to provide the right input to the detector and to output its findings. The pipeline can be run in Experiment RUB, Experiment P, and Experiment R mode. From the detector’s point of view, running Experiment R is the

6. Experiment Pipeline



Review Site

Review

Detector: GrouMiner
 Target: project 'synthetic' version hasNext
 Misuse: misuse 'hasnext'

Misuse Details

Details about the known misuse from the MuBench dataset.
 Description: Misses to check that there is another element before calling `Iterator.next()`.
 Fix Description: Check `iterator.hasNext()` or ensure size of the collection (`checkSize()`, `isEmpty()` or `add()` an element).
 Violation Types:

- missing/condition/value_or_state

 In File: OnlyNext.java
 In Method: misuse(Collection)
 Code with Misuse:

```

4.  class OnlyNext {
5.      public Object misuse(Collection<Object> os) {
6.          Iterator<Object> itr = os.iterator();
7.          return itr.next();
8.      }
9.  }
10. }

```

Potential Hits

Findings of the detector that identify an anomaly in the same file and method as the known misuse.

Hit	Rank	Missing Edges	Missing Nodes	Overlap	Pattern	Rareness	Violation Types
Yes	0	Iterator#hasNext -> Iterator#next	Iterator#hasNext	<pre> graph LR A([Iterator<Object>#itr#next]) B([Iterator<Object>#itr#hasNext]) C([Iterator<Object>#itr#next]) A --> B B --> C </pre>	<pre> graph TD A([Iterator<Object>#itr#hasNext]) --> B([Iterator<Object>#itr#next]) </pre>	0.02	

Figure 6.2.: A Finding of GROUMINER Presented on the Review Site

same as running Experiment P. The difference comes in the reviewing process where only findings that match the known misuses are reviewed in Experiment R, while a sample of *all* findings is reviewed in Experiment P. After running the actual detector, the runner converts the detectors' findings into a unified format to facilitate the following validation step. For each finding, this format specifies the name of the file and the name of the method that the finding is in. In addition, the runners may add tool-specific data that helps with validation (e.g., the detector's confidence value, a description of the violated pattern, or a description of the violation).

Apart from adding some accessor methods that allow us to obtain the detectors' output, we leave the detector implementations unchanged. We sent the runner code to the original authors of the detector implementations and they confirmed that our invocations of their code are correct.

Validation To help with the manual review of findings, MuBENCHPIPE automatically publishes experiment results to a review website that shows for every detector finding the source code it is found in along with any metadata the detector provides, such as the violated pattern, the properties of the violation, and the detector's confidence. Figure 6.2

shows an example of this for a finding of the GROUMINER detector.

For Experiment RUB and Experiment R, MUBENCHPIPE automatically filters potential hits, by matching findings to known misuses by file and method name. On the review website, a reviewer sees the description of the known misuse as well as its fix, along with the set of potential hits that need to be reviewed. For Experiment P, MUBENCHPIPE shows all findings of the detector on the review site, each with the source code it is located in.

The review website allows reviewers to save an assessment and comment for each finding. It also ensures at least two reviews for each finding, before automatically computing the experiment statistics, such as the precision, recall, and KAPPA scores.

6.3. Reproducibility and Traceability

We publish MUBENCH³ and encourage others to use and contribute to the dataset and the automated pipeline to conduct and repeat experiments and to extend the benchmark itself. MUBENCH comes with a Docker⁴ container, which allows running reproducible experiments across platforms, without the need to ensure a proper environment setup.

For legal reasons, we do not include source code or binaries of target projects in the benchmark itself, but instead provide links to the respective version-control systems and tooling that automates the retrieval of respective checkouts and their compilation. This minimizes the effort to collect the dataset and ensures access to the exact same versions of the projects. The results of checkout and compilation are stored locally, such that the respective data remains available for subsequent use.

We provide binaries of the misuses detectors integrated into MUBENCH via our artifact page. Like the target code, MUBENCH retrieves them automatically upon first use and stores them locally for subsequent use. Multiple versions of a detector may be registered to MUBENCH, each with a version id and (optionally) a tag. This provides access to the specific detector versions used in a concrete experiment, even if the respective detectors have meanwhile been updated.

MUBENCH’s review website—based on PHP and MySQL, such that it can be hosted on any of-the-shelf webspace—facilitates independent reviews, even when researchers work from different locations, while ensuring review integrity using PHP Basic Auth. The website may directly be used as an artifact to publish review results and experiment statistics. While reviews may not be completely reproducible, due to the subjectivity of human reviewers, publishing the review details, including review comments justifying the decisions, makes them at least traceable. All detector findings, review, and statistical data may be exported from the review site in CSV format, for processing with other tooling.

³<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

⁴ <https://www.docker.com/> (checked on Dec 14, 2017)

6.4. Limitations

A crucial step of any evaluation of an API-misuse detector is the verification whether a particular finding correctly identifies an actual API misuse. While MUBENCH eases reviews by providing a uniform representation of findings and pre-filtering potential hits based on their location, it cannot fully automate this step. Full automation is indeed impossible for Experiment P, because any finding might identify a previously unknown misuse, such that automated verification would require an approach that can determine for any given usage whether it is a misuse or not. If such an approach existed, it would itself be the perfect misuse detector. For Experiment RUB and Experiment R we provide MUBENCH with specifications of the known misuses and use the location information from this specification to pre-filter potential hits. It may be possible to improve this filtering using additional data. However, since all detectors use different representations of API usage, patterns, and violations, there is no general way to decide for any misuse detector whether one of its findings identifies a particular misuse, especially if both the misuse and the detector’s output format are unknown in advance. Therefore, we can only reduce the manual review effort, but not entirely eliminate it.

Considering that benchmarking experiments require the validation of detector findings by human reviewers, their assessment might be subjective. To mitigate this problem, we propose that at least two reviewers independently review each finding and discuss any disagreements. Furthermore, with the MUBENCH review site, we provide a tool to easily publish review results, including reviewer comments justifying decisions, which makes assessments at least traceable, if not strictly reproducible.

7. An Empirical Study of API-Misuse Detectors

With MUBENCH, our benchmark for static API-misuse detectors, in place, we want to empirically evaluate and compare the misuse detectors JADET, GROUMINER, TIKANGA, and DMMC in the experiments described in Chapter 5. We run all experiments on a *MacBook Pro* with an *Intel Xeon @ 3.00GHz* and *16GB of RAM*. The full results are available on our artifact page.¹

7.1. Experiment P: Precision of the Detectors

Table 7.1 shows our precision results, based on reviewing the top-20 findings per detector on each of our five sample projects. The second column shows the total number of findings in the detectors' top-20, 230 in total across all detectors. Note that all detectors report less than 20 findings for some projects. The third column shows the confirmed misuses after resolving disagreements, and the fourth column shows the precision with respect to the reviewed findings. The fifth column shows the KAPPA score for the manual reviews, and the remaining columns show the frequencies of root causes for false positives. We find that the precision of all detectors is extremely low. TIKANGA shows the best precision of only 11.4%. JADET and DMMC follow immediately behind, with a precision of 10.3% and 9.9%, respectively. GROUMINER reports only false positives in its top-20 findings.

Obs.1: All detectors have extremely low precision (below 12%). On average, they report less than 1.5 actual misuses in their top-20 findings.

The Kappa scores indicate high reviewer agreement, which shows that all detectors produced mostly clear false positives. The score is a little lower for TIKANGA, because it reported one confirmed misuse twice, which one of the reviewers first accepted as an actual hit while the other did not. The score is also lower for DMMC, because we initially disagreed on several violations it identifies in `Iterator` usages that do not check `hasNext()`, but the underlying collection's size.

¹Artifact Page: A Systematic Evaluation of Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mustudy/>)

7. An Empirical Study of API-Misuse Detectors

Table 7.1.: Experiment P: Precision of the Detectors on the Top-20 Findings on Five Projects and Root Causes for False Positives.

Detector	Findings in Top-20	Confirmed Misuses	Precision	Kappa Score	Frequencies of Root Causes for False Positives						
					Uncommon	Analysis	Alternative	Cross-method	Dependent	Bug	Multiplicity
JADET	39	4	10.3%	0.97	21	3	8	0	1	0	2
GROUMINER	66	0	0.0%	0.97	25	22	8	7	2	1	1
TIKANGA	44	5	11.4%	0.93	18	7	7	0	7	0	0
DMMC	81	8	9.9%	0.91	9	19	18	19	4	4	0
Total	230	17		0.94	73	51	41	26	14	5	3

True Positives

Out of the 230 reported findings we reviewed, we confirm 17 true misuses. DMMC reports 8 misuses of the `Iterator` API where `hasNext()` is not checked. JADET reports 4 misuses that access a collection without previously checking its size. Also for collections, TIKANGA reports 4 misuses with a missing `hasNext()` and 1 misuse with a missing size check. One misuse is reported by both TIKANGA and JADET and another by both TIKANGA and DMMC. Additionally, JADET reports one misuse twice. This leaves a total of 14 unique misuses in the detectors’ top-20 findings, all different from the known misuses in MUBENCH. Interestingly, *all* these misuses are missing value/state conditions, for which the detectors report only missing calls to methods that would be used in the respective missing checks (i.e., misuses we accept as hits in our lenient review process).

Obs.2: All 14 confirmed misuses in Experiment P are missing value/state-condition checks before accessing the elements of a collection, either directly or through an iterator.

False Positives

To identify opportunities to improve the precision of misuse detectors, we systematically investigate the root causes for the false positives they report. In the following, we discuss these root causes summarized across all detectors, in order of their absolute frequency. We label these root causes (**FPN**).

Root Cause FP1: Uncommon-but-correct Usage. Particular usages may violate the patterns that detectors learn from frequent usages, without violating actual API usage constraints. Detectors cannot differentiate such infrequent usage from invalid usage. For example, DMMC and JADET learn that the methods `getKey()` and `getValue()`

of `MapEntry` usually appear together in code. They both report violations if a call to either of these method is missing, or, in case of JADET, if the calls appear in a different order. However, there is no requirement by the API to always call both getter methods, let alone in a specific order. Across the reported violations we analyzed, the detectors falsely report 42 missing method calls in cases where one out of a number of getter methods is missing or invoked in a different order. Another example is that JADET and TIKANGA learn that methods such as `List.add()` and `Map.put()` are usually invoked in loops and report five missing loops for respective invocations outside a loop, which are perfectly fine according to the API. Approaches such as multi-level patterns [SBAS15] or ALATTIN’s alternative patterns [TX09a] may help to mitigate this problem. Also note that the four detectors in our experiments all use absolute frequency thresholds, while some of the detectors from our survey in Chapter 4 also used relative thresholds. Future work should investigate how these two alternatives compare.

Obs.3: Particular usages may be uncommon without violating API constraints. Neglecting this causes 73 (34.3%) of the detectors’ false positives in their top-20 findings. This calls for research on detecting patterns without setting a hard threshold on occurrence frequencies. Meanwhile, relaxing requirements on the co-occurrence of getter methods might reduce false positives significantly.

Root Cause FP2: Imprecise Analysis. The detectors use static analysis to determine the facts that belong to a particular usage. Imprecisions of these analyses lead to false positives. For example, the detectors mistakenly report five missing elements in code that uses multiple aliases for the same object and another 17 in code with nested control statements, where they fail to capture all calls belonging to a usage. GROUMINER reports two missing method calls, because it cannot resolve the receiver type for chained method calls, such as for `m()` in `o.getX().m()`, which is not generally possible from source code alone. Therefore, GROUMINER fails to match a call between the pattern and the usage. Another example is that the detectors report eight missing method calls due to chained calls on a fluent API, such as `StringBuilder`, where their intra-procedural analyses cannot determine that all calls actually happen on the same object. JADET, GROUMINER, and DMMC together report nine missing calls that happen transitively in a helper method of the same class or through a wrapper object, such as a `BufferedStream`. DMMC reports a missing call that is located in the enclosing method of an anonymous class instance and a missing `close()` call on a parameter that is, by contract, closed by the callers. Moreover, GROUMINER reports four missing conditions that are checked by assertion helper methods. An inter-procedural detection strategy, as proposed by PR-MINER [LZ05], could mitigate this problem.

Obs.4: Imprecisions of the detectors’ static analyses cause 51 (23.9%) of the false positives in their top-20 findings. An inter-procedural detection strategy might be able to eliminate 14 (6.6%) of these false positives.

Root Cause FP3: Alternative Patterns. The detectors often learn a pattern and then report instances of alternative usages as violations. We define *alternative usages* as a different correct way to use an API, either to achieve the same or a different functionality. When multiple alternatives occur frequently enough to induce patterns, the detectors learn *alternative patterns*. For example, JADET, TIKANGA, and DMMC learn that before a call to `next()` there should always be a call to `hasNext()` on an `Iterator`. Consequently, they report 16 violations in usages that only pull the first element and check either `isEmpty()` or `size()` on the underlying collection to ensure this element exists. DMMC reports another violation, because `isEmpty()` is used instead of `size()` before accessing a `List`. Another example is that JADET, TIKANGA, and DMMC learn that collections are filled one element at a time, e.g., by calling `add()`, and report 10 missing methods in usages that populate a collection differently, e.g., through the constructor or using `addAll()`. GROUMINER reports four usages where an alternative control statement is used, e.g., a `for` instead of a `while`.

A special case of this root cause is alternatives to obtain an instance of a type. For example, GROUMINER mistakenly reports two missing constructor calls where the instance is not created through a constructor call as in the pattern, but returned from a method call. JADET and DMMC each report one missing constructor call where an instance is not created, but received as a parameter. While handling alternative patterns is an open problem, some tools such as ALATTIN already propose possible solutions [TX09a].

Obs.5: A violation of one pattern might be an instance of another, alternative pattern. Not considering this causes 41 (19.2%) of the detectors' false positives in their top-20 findings.

Root Cause FP4: Cross-method Usage. Objects that are stored in fields are often used across multiple methods of the field's declaring class. The respective API usages inside the individual methods might then deviate from usage patterns without being actual misuses. Listing 7.1 shows an example of such a case, where two fields of type `Iterator`, `in` and `out`, are used to implement the class `NeighborIterator`. When `in` yields no more elements (Line 12), the call to `next()` in Line 14 happens on `out` without a prior check whether it has more elements. While this appears to be a misuse of the `Iterator` API inside the enclosing method, it is a correct usage inside the enclosing class, since `NeighborIterator` itself implements `Iterator` and, thereby, inherits its usage constraints. Correct usages of `NeighborIterator` need to check its `hasNext()` method (Line 6) before calling its `next()` method (Line 11), which ensures that `out` has more elements when `next()` is called on it. DMMC and GROUMINER report sixteen violations for such usages of fields of a class.

A special case of this root cause is when a class uses part of its own API in its implementation, for example, when a `Collection` calls its own `add()` method in the implementation of its `addAll()` method. DMMC and GROUMINER report four such violations. This is particularly interesting, because these are actually self usages of the

```

1 class NeighborIterator implements Iterator<GraphNode> {
2     private final Iterator<DiGraphEdge> in = ...;
3     private final Iterator<DiGraphEdge> out = ...;
4
5     @Override
6     public boolean hasNext() {
7         return in.hasNext() || out.hasNext();
8     }
9
10    @Override
11    public GraphNode next() {
12        boolean isOut = !in.hasNext();
13        Iterator<DiGraphEdge> curIterator = isOut ? out : in;
14        DiGraphEdge s = curIterator.next();
15        return isOut ? s.getDestination() : s.getSource();
16    }
17
18    ...
19 }

```

Listing 7.1.: Correct Usages of `Iterator` Instances in the CLOSURE Project that Violate Usage Patterns.

API, while the detectors target client usages. Since any codebase likely contains such self usages, detectors should consider this.

Obs.6: The implementation code of a class may contain partial usages of the class' own API or fields. Such usages cause 26 (12.2%) of the detectors' false positives in their top-20 findings.

Root Cause FP5: Dependent Object States. When two objects' states depend upon each other, usages sometimes check the state of one and implicitly draw conclusions about the state of the other. The detectors do not consider such inter-dependencies. For example, when two collections are maintained in parallel, i.e., always have the same size, it is sufficient to check the size of one of them before accessing either. The detectors falsely report 14 missing size checks in such usages. In 10 of these cases, the equal size is ensured by construction of the collections in the same method. In the remaining four cases, it is ensured elsewhere in the same class. We consider this a dangerous practice, because should the dependency between the collections ever change, it is easy to miss some of the code that relies on it. Thus, warning developers might be justified. Nevertheless, we count these cases as false positives, since the current usages are correct.

Obs.7: Semantic dependencies between objects' states may implicitly ensure conditions. Not considering such inter-dependencies causes 14 (6.6%) of the detectors' false positives in their top-20 findings.

Root Cause FP6: Call Multiplicity. The detectors cannot handle methods that may be called arbitrarily often. GROUMINER and JADET both learn a pattern where the

7. An Empirical Study of API-Misuse Detectors

Table 7.2.: Experiment RUB: Recall of the Isolated Detection Strategies and Root Causes for Divergences.

Detector	Potential Hits	Actual Hits	Conceptual Recall Upper Bound	Empirical Recall Upper Bound	Kappa Score	Frequencies of Root Causes for Divergences					
						Representation Matching	Analysis Bug	Lenient Exception Handling			
JADET	19	15	29.7%	23.4%	0.76	4	4	1	0	3	2
GROUMINER	46	31	75.0%	48.4%	0.84	9	4	6	0	8	0
TIKANGA	23	13	29.7%	20.3%	0.84	4	7	2	0	5	2
DMMC	40	15	26.6%	23.4%	0.85	5	0	0	2	5	0
Total	128	74			0.83	22	15	9	2	21	4

`append()` method of `StringBuilder` is called twice and falsely report three missing method calls where it is called only once.

Obs.8: Detectors should distinguish methods that require a specific number of calls, from methods that require one or more calls, and methods that may be called arbitrarily often. Not considering this causes 3 (1.4%) of the detectors' false positives in their top-20 findings.

Root Cause FP7: Bug. A few findings are likely caused by mistakes in the detector implementations. DMMC reports four violations with an empty set of missing methods. These empty sets are produced when none of the potentially missing methods match DMMC's prevalence criteria. DMMC should probably filter such empty-set findings before reporting. GROUMINER reports one missing `if` that actually appears in all respective usages, because its graph mapping does not match the respective `if` node from one of the usages with the corresponding nodes of all the other usages.

7.2. Experiment RUB: Recall Upper Bound of the Detectors

We run all detectors to see which of the 64 known misuses from MUBENCH they can detect when given the respective crafted correct usages for pattern mining. Table 7.2 shows the results per detector. The second and third column show the number of potential hits and the number of actual hits, after resolving disagreements. The fourth and fifth column show the detector's conceptual and empirical upper bounds of recall, respectively. The sixth column shows the KAPPA score for the manual reviews. The remaining columns show the frequencies of root causes for divergences between a detector's conceptual capabilities from Table 4.1 and its actual findings in this experiment. Note

7.2. Experiment RUB: Recall Upper Bound of the Detectors

that two detectors sometimes have the same root cause for their respective divergence on the same misuse.

We find that GROUMINER has by far the best conceptual recall upper bound and also shows the best empirical recall upper bound in Experiment RUB. This suggests that its graph representation is a good choice to capture the differences between correct usages and patterns. However, the gap between GROUMINER’s conceptual and empirical upper bounds of recall is quite noticeable. Actually, Table 7.2 shows that all four detectors fall considerably short of their conceptual recall upper bound in practice.

Generally, we observe two kinds of divergences between the actual findings and the conceptual capabilities: *Unexpected false negatives*, i.e., misuses that a detector should be able to detect, but does not, and *unexpected hits*, i.e., misuses that a detector supposedly cannot detect, but does. We investigate the root causes of each divergence to identify actionable ways to improve misuse detectors.

Obs.9: All detectors’ empirical upper bound of recall is much lower than their conceptual upper bound of recall and their findings frequently diverge from their conceptual capabilities.

The Kappa scores indicate good reviewer agreement, albeit a little lower than in Experiment P. Since we only reviewed potential hits, i.e., findings in the same method as a known misuse, many potential hits were related to the known misuses. Consequently, we had several disagreements on whether a particular potential hit actually identifies a particular misuse. In total, we had 18 such disagreements (JADET: 4; GROUMINER: 6; DMMC: 5; TIKANGA: 3), which led us to formulate the lenient review process described in Section 5.3. We decided in favor of the detectors in eight of these cases. We observe that the Kappa score is a little lower for JADET, compared to the other detectors. Since the absolute number of disagreements is comparable and JADET had relatively few potential hits, i.e., a small number of decisions as a basis for the Kappa score, we attribute the lower score to chance.

Unexpected False Negatives

To identify opportunities to improve the recall of misuse detectors, we systematically investigate the root causes for the false negatives they report. In the following, we discuss these root causes summarized across all detectors, in order of their absolute frequency. We label these root causes (**FNN**).

Root Cause FN1: Imprecise Representation. Current usage representations are not expressive enough to capture all details that are necessary to differentiate between misuses and correct usages. For example, DMMC and GROUMINER encode methods by their name only and, therefore, cannot detect a missing method call, when the usage calls an overloaded version of the respective method. For example, assume that a pattern requires a call to `getBytes(String)`, but the target usage calls `getBytes()` instead. An ideal misuse detector would still report a violation, since the expected method, with the

7. An Empirical Study of API-Misuse Detectors

```
1  writer.write(value);  
  
2  try {  
3    writer.write(value);  
4  } finally {  
5    if (writer != null)  
6      writer.close();  
7  }
```

Listing 7.2.: Not Closing Writer vs. Correctly Closing Writer.

correct parameters, is not called. However, since only the method name is used for comparison in both these detectors, such a violation is not detected. Another example is that, to use a `Cipher` instance for decryption, it must be in decrypt mode. This state condition is ensured by passing the constant `Cipher.DECRYPT` to the `Cipher`'s `init()` method. None of the detectors captures this way of ensuring that the condition holds, because they do not encode method-call arguments in their representations.

Obs.10: Inability to capture details necessary to differentiate misuses from correct usages in the usage representation is responsible for 22 (45.8%) of the unexpected false negatives.

Root Cause FN2: Imprecise Pattern Matching. The detectors fail to relate a pattern and a usage. Typically, detectors relate patterns and usages by their common facts. If there are no or only few common facts, detectors report no violation. For example, JADET's facts are pairs of method calls. In a scenario where `JFrame`'s `setPreferredSize()` method is accidentally called after its `pack()` method, JADET represents the usage with a pair (`pack`, `setPreferredSize`) and the pattern with the reverse pair. Since it compares facts by equality, JADET finds no relation between the pattern and the usage. Without common facts between a usage and a pattern, the detector assumes that these are two completely unrelated pieces of code and does not report a violation. Another example is when the pattern's facts relate to a type, e.g., `List` in `List.size()`, while the usage's facts relate to a super- or sub-type such as `ArrayList.size()` or `Collection.size()`. The detectors cannot relate these facts, since they are unaware of the type hierarchy. Also, TIKANGA misses four misuses, because the target misses more than two formulae of the pattern (TIKANGA's maximum distance for matching). For example, Listing 7.2 shows a misuse that does not close a `Writer` and the corresponding correct usage. In TIKANGA's representation, the difference between the misuse and the correct usage consists of three formulae: (1) that `close()` follows `write()` in case of normal execution, (2) that `close()` follows `write()` if the latter throws an exception, and (3) that `close()` is preceded by a `null` check.

Obs.11: When matching patterns and misuses, detectors should consider the semantics of their representation, e.g., call order and the number of usage facts generated by adding specific usage constructs, as well as code semantics, e.g., subtype relations. Neglecting this is responsible for 15 (31.3%) of the unexpected false negatives.

7.2. Experiment RUB: Recall Upper Bound of the Detectors

```
1 ArrayList markers;  
2 if (layer == Layer.FOREGROUND) {  
3     markers = (ArrayList) this.fgMarkers.get(index);  
4 }  
5 else {  
6     markers = (ArrayList) this.bgMarkers.get(index);  
7 }  
8 // if (markers != null) { // <-- missing in misuse  
9 boolean removed = markers.remove(marker);  
10 // }
```

Listing 7.3.: Example of an Analysis Problem of GROUMINER.

Root Cause FN3: Imprecise Analysis. The detectors rely on static analysis to extract their usage representations. Imprecisions in these analyses may obscure relations between patterns and usages. For example, GROUMINER fails to detect one missing `null` check, because it cannot determine the receiver type for chained calls, such as for `m()` in `o.getX().m()`, which is not generally possible from source code alone. Also, it fails to detect another four missing `null` checks, because it overlooks dataflow dependencies. Listing 7.3 shows such a case. In addition to the `null` check, GROUMINER also misses the dataflow from the `get()` calls to the `remove()` call in the misuse, which makes the pattern and usage differ by multiple facts. GROUMINER, however, only reports a violation if the difference is a single fact. TIKANGA misses a call that occurs in the correct usage in one case and fails to capture the call order between two calls from the correct usage in another case. We assume that the cause is a limitation of its analysis, but could not ultimately verify this, because the tool’s developer is not available to confirm the implementation details.

Obs.12: Imprecision of the analysis, which obscures the relation between patterns and misuses, causes nine (18.8%) of the unexpected false negatives.

Root Cause FN4: Bug. DMMC skips the comparison of a usage and a pattern if the pattern contains fewer calls than the usage, presumably to improve performance. The pattern for `AuthState` from Apache’s `HTTPCLIENT`, for instance, requires three calls, of which the misuse scenario misses one. However, if this misuse has an additional, optional call that is not in the pattern, DMMC skips the comparison since now both the pattern and the target each contain three calls. This causes two unexpected false negatives in our experiment.

Unexpected Hits

We find that the detectors identify some misuses that we did not expect them to find, according to their conceptual capabilities (see Chapter 4). We systematically investigate the root causes for these unexpected hits. In the following, we discuss these root causes. We label them (**UHN**).

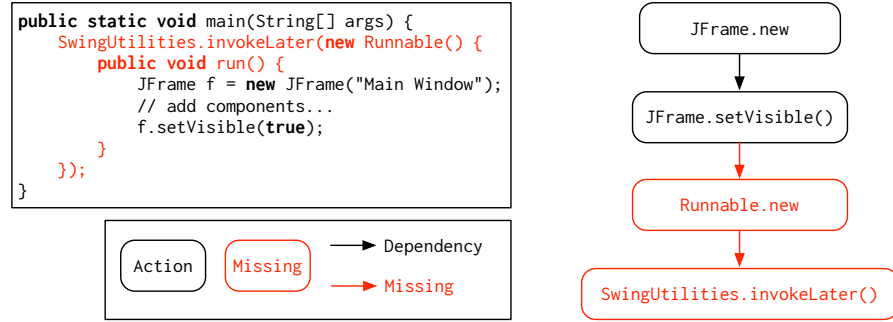


Figure 7.1.: Code and GROUM of a Usage That Instantiates a SWING Component (on the Event Dispatching Thread).

```

1  writer.write(value);
2  writer.close();

1  try {
2      writer.write(value);
3  } finally {
4      writer.close();
5  }
```

Listing 7.4.: Closing `Writer` Without and With Exception Handling.

Root Cause UH1: Lenient Review Process. In all but two cases, the reason for *unexpected hits* is our lenient review process described in Section 5.3. In most cases, the detectors report a missing call that indicates a missing condition check. The only other case is that GROUMINER detects a missing context condition, in a scenario where some SWING code is required to run on the Event Dispatching Thread (EDT). The delegation to the EDT is implemented by wrapping the code in an anonymous instance of `Runnable`, as shown in Figure 7.1. GROUMINER considers the code in `run()` as part of code of the enclosing method. Consequently, it suggests the misuse by reporting a missing instantiation of `Runnable` before the instantiation of the `JFrame`.

Obs.13: Missing method calls may indicate missing condition checks. Detectors that report these missing calls, despite not reporting the exact condition, find violations outside of their conceptual capabilities.

Root Cause UH2: Capturing Exception Handling. In the remaining two cases, JADET and TIKANGA correctly report missing exception handling. For example, Listing 7.4 (left) shows a misuse where `close()` is not called when `write()` throws an exception. A corresponding correct usage is shown on the right. TIKANGA and JADET both represent the correct usage with two facts $\{(\text{write}, \text{close}), (\text{write:EXC}, \text{close})\}$, effectively encoding that `close()` is called after `write()` in normal execution and in case of an exception. In the misuse, they find the second fact missing. This capability of the implementations is not mentioned in the respective publications.

Table 7.3.: Experiment R: Recall of the Detectors on MUBENCH and the New Misuses from Experiment P.

Detector	Potential Hits	Actual Hits	Recall	Kappa Score
JADET	4	3	5.7%	1.00
GROUMINER	4	0	0.0%	1.00
TIKANGA	9	7	13.2%	1.00
DMMC	25	11	20.8%	0.95
Total	42	21		0.97

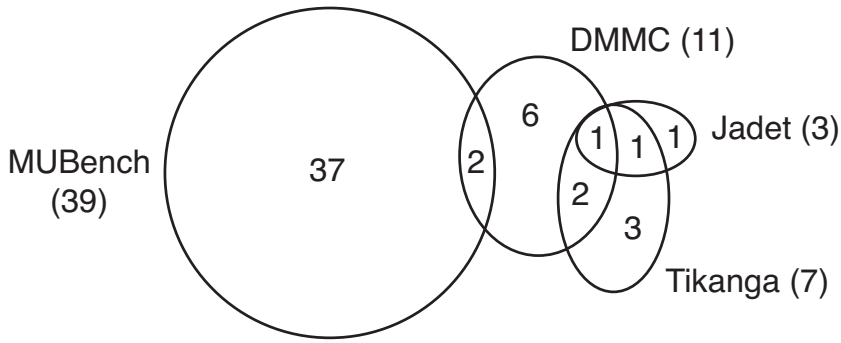


Figure 7.2.: Recall of the Detectors in Experiment R

7.3. Experiment R: Recall of the Detectors

In Experiment R, we run all detectors to assess their recall when using their own pattern mining. To MUBENCH’s 64 misuses we add the 14 new misuses from Experiment P and exclude the 25 hand-crafted examples for which there is no project code to mine patterns from. This leaves us with 53 misuses for Experiment R.

In total, the detectors report 18,384 findings on all projects. Due to the automated filtering for potential hits based on the finding location, we have to review only 62 (0.3%) of these findings to determine the detectors’ recall in Experiment R. This shows that MUBENCH drastically reduces the review effort required to measure recall, which makes our experiment practically feasible at all.

Table 7.3 shows the results of Experiment R and Figure 7.2 visualizes the recall. JADET finds only the three misuses it already identified in Experiment P. GROUMINER does not find any of the misuses. TIKANGA finds the five misuses it already identified in Experiment P, one of the misuses that DMMC identified in Experiment P, and one of the misuses that JADET identified in Experiment P. DMMC finds two misuses from MUBENCH (both missing method calls), the eight misuses it reported in Experiment P, and one misuse both JADET and TIKANGA reported in Experiment P.

7. An Empirical Study of API-Misuse Detectors

DMMC shows by far the best recall in Experiment R. This suggests that its relatively simple detection strategy works well when focusing on missing method calls. However, the recall of all detectors in the real setting offered by Experiment R is low. Analyzing the root causes for their bad performance, we identify two general problems with the design of the detectors and their evaluation setting.

Problem 1: Poor Ranking. While Experiment R shows that the detectors identify more misuses beyond their top-20 findings, they, unfortunately, rank those very low. For example, the two MUBENCH misuses DMMC finds are ranked 309 and 613. This is far beyond the number of findings that we can reasonably expect a user to assess. The four detectors in our experiments all use different ranking strategies, but none of the detectors from our survey in Chapter 4 compared different strategies on the same detector.

Obs.14: Detectors need better ranking strategies to report true positives within their top findings. Furthermore, researchers should compare alternative ranking strategies for single detectors.

Problem 2: Lack of Usage Examples. The huge difference in the detectors’ performance between Experiment RUB and Experiment R suggests that the cause may be a shortage of correct usage examples in the target projects. One possibility is that the number of such examples is smaller than the detectors’ minimal support for pattern mining, in which case we could simply lower these thresholds. However, this would likely also increase the number of false positives as the mined patterns generally become less reliable, which underlines the need to effectively filter false positives (*Obs.1*) and improve ranking (*Obs.14*). Another possibility is that no, or only very few, such examples exist in the projects. This would be a general problem with the evaluation setting of misuse detectors. To solve it, we need additional sources of usage examples to mine patterns from. Gruska *et al.* [GWZ10] demonstrated one possible approach by applying JADET in a multi-project setting with 6,000 projects, but did not measure recall. Other recommender systems for software engineering, such as code-completion engines [PLM15], work cross project, i.e., they learn from a large number projects to provide recommendations for other projects. The misuse detectors CAR-MINER [TX09b] and ALATTIN [TX09a] implement an alternative approach, by specifically searching for usage examples of the APIs used in the target project via a code-search engine. Related to this, other lines of research proposed code-search engines to find usage examples in open source projects [GFX⁺10, MGP⁺11] or on StackOverflow [PBDP⁺14].

Obs.15: All detectors have low recall, likely due to lack of correct usage examples in target projects. Adoption of existing code-search techniques and cross-project mining could mitigate this problem.

The Kappa scores indicate mostly perfect reviewer agreement in Experiment R. This is because the detectors found almost exclusively the misuses that one of them also identified in Experiment P, i.e., the misuses we already agreed on before. The exception

Table 7.4.: JCA Misuses Identified by the Detectors in Experiment RUB

	Total	Identified by				
		JADET	GROUMINER	TIKANGA	DMMC	Any
JCA Misuses	22	2 (9.1%)	2 (9.1%)	1 (4.5%)	1 (4.5%)	3 (13.6%)

is DMMC, where we initially disagreed on one of its 14 potential hits for misuses from the original MUBENCH dataset.

7.4. Vulnerability Detection

We are particularly interested in whether the misuse detectors identify the misuses of the Java Cryptograph Architecture (JCA) APIs, because our study in Section 2.4 suggests that these are prevalent and rarely fixed, possibly because they do not cause spurious behavior. We have 22 JCA misuses in compilable project versions of our ground-truth dataset. Among these we find 15 security vulnerabilities: eight cases specify an insecure algorithm,² five cases rely on a potentially insecure default configuration, and two cases hard-code the cryptographic key. The remaining seven cases have some functional defect that may cause an exception or data loss.

Table 7.4 shows how many of the JCA misuses each of the detectors identifies in Experiment RUB. Each individual detector identifies at most two of the misuses and even together they identify only three of them: A missing reinitialization call, a missing exception handling in case of an invalid cryptographic key being supplied, and a resources leak (missing call to `close()`) that may cause loss of encrypted data. All three misuses cause spurious behavior (exceptions or data loss), rather than security vulnerabilities. This indicates that current misuses detectors are not particularly good at identifying JCA vulnerabilities.

Looking at the concrete misuses, we find that this is actually unsurprising, since all vulnerabilities manifest in illegal parameter values, either in the string parameter passed to `Cipher.getInstance()` (insecure algorithm or algorithm configuration) or in the algorithm’s parameters (e.g., a hard-coded cryptographic key). While we might generally design misuse detectors that consider parameter values, we find that the design of the `Cipher` API poses four particular challenges:

First, the algorithm configuration is passed to `getInstance()` in a single string parameter, which encodes the algorithm name, the algorithm mode, and the algorithm padding to use for encryption, separated by slashes, e.g., `"AES/CBC/PKCS5Padding"`. The second and third part may be omitted, in which case a default configuration is

²In five cases (62.5%) it is the DES algorithm. We hypothesize that a major factor for the prevalence of this particular mistake might be that the official Android developer documentation contains an example of using `Cipher` with DES:

<https://developer.android.com/reference/javax/crypto/Cipher.html> (checked on Jul 18, 2017)

7. An Empirical Study of API-Misuse Detectors

used. This makes it difficult for a misuse detector to learn about safe and unsafe configurations, except through memorizing all respective values. For example, without special knowledge about how to parse configuration strings, detectors cannot abstract that both "DES" and "DES/ECB" are insecure for the same reason, i.e., the insecure DES algorithm.

Second, static API-misuse detectors generally assume that Liskov's Substitution Principle [Lis87] holds, i.e., that all instances of the same static type behave the same. Consequently, they assume that a valid usage of one instance of a type is also a valid usage for any other instance of the same type. This assumption does not hold for the `Cipher` API:

1. The JCA uses the provider pattern, i.e., calls to `Cipher.getInstance()` are delegated to some provider entity, which is responsible for instantiating a respective `Cipher` instance. The concrete provider may change depending on the system or application configuration. While the JCA requires all providers to supply certain algorithms, they may supply additional algorithms. Hence, whether a particular algorithm is available may depend on the concrete provider, which may be unknown statically.
2. Valid algorithm configurations depend on the provider as well. For example, with the BOUNCYCASTLE provider,³ "RSA/None/..." is a valid configuration. Since the RSA algorithm does not use an algorithm mode, the provider expects usages to specify `None` as the mode. The Java default provider, however, expects respective usages to specify `ECB` as the mode. Either provider throws an exception, if the respective other value is passed.
3. How an instance of `Cipher` behaves depends on the algorithm it was created with. For example, while the RSA algorithm expects a `PublicKey` instance for initialization, the AES algorithm expects a `SecretKey` instance and a random initialization vector. This means that part of the string parameter to `getInstance()` effectively determines the behavioral type of the respective `Cipher` instance. Interestingly, both the BOUNCYCASTLE and the Java default provider implement the different algorithms as individual subtypes of `Cipher`. Thus, the runtime type of a `Cipher` instances uniquely identifies its behavioral type.

Third, what can be considered secure usage of the JCA changes over time, since what was considered secure encryption 20 years ago might not be considered secure anymore. This is quite different compared to functional aspects of APIs, where a correct usages remains correct, unless one migrates to another version of the API.

Fourth, it seems likely that we cannot learn about the correct usage of JCA APIs from client code. Egele *et al.* [EBFK13] report that 88% of all Android apps misuse cryptographic APIs. We find a comparably high fraction of GitHub project (73.1%) does the same (see Section 2.4). This suggests that for the JCA API the general assumption that frequently reoccurring usages correspond to correct usages might not hold.

³<http://www.bouncycastle.org> (checked on Nov 02, 2017)

Considering these particular challenges, most of which are specific to the JCA APIs, we deem it unlikely that general API-misuse detectors can be adapted to reliably identify misuses of the JCA APIs that cause security vulnerabilities.

7.5. User Experience

We now report on our experiences as users of our subject misuse detectors. Our observations are based on the experience we gained while reviewing the detectors' findings in our experiments.

DMMC simply reports present and missing method calls, along with the source line number of the first present call. We find this output generally easy to interpret. The line number helps, especially, to locate usages in large methods. GROUMINER reports pattern and usage graphs, which are more difficult to understand. However, we find that the structural properties of the source code that the graph representation captures help with the interpretation. JADET and TIKANGA report the present and missing facts of their respective representations. We find that it is often difficult to relate the facts to each other, especially in the presence of multiple usages of the same API. This might be, in part, due to the textual representation we look at. While none of the detector implementations was intended to present their findings to end users, we still find it interesting to note that the challenge of explaining findings seems to correlate with the distance between the source code and the usage representation.

We also find that Bytecode-based detectors may report findings in code that the compiler introduces. For example, the compiler translates `foreach` loops into `Iterator` usages. TIKANGA reports a missing call in such a usage, i.e., it reports a missing call on `Iterator` in a method where `Iterator` does not appear in the source code. This finding confused us at first. While additional steps could be taken to assist the user in mapping such findings back to the source code, source-based detectors do not face this problem.

Our lenient review process shows that missing method calls frequently indicate missing conditions (*Obs.2* and *Obs.13*). While such findings do not report the entire problem, we found it relatively easy to deduce their meaning. Contrarily, GROUMINER reports only a missing `if` node, when it captures a missing condition. While these findings more explicitly indicate the problem of a missing check, we feel that they are actually harder to act upon, because they give no information about *what* should be checked. This indicates a gap between a detector's capability to identify a violation and its ability to explain this violation to users.

Above all, we believe that the detectors' precision is likely to be the biggest threat to their applicability in practice. As previous studies show, large numbers of false positives are a major barrier in the adoption of code analysis tools [FLL⁺02, BBC⁺10, JS13]. This problem is made worse by the low recall of the detectors, which implies that even if developers would take the time to review all reported warnings, they would still likely miss the vast majority of misuses.

7.6. Call to Action

We find that misuse detectors are practically capable of detecting a considerable part of the misuses in MUBENCH, when provided with the correct usages to compare to (Experiment RUB). However, even though the detectors are also capable of finding some misuses in a realistic setting (Experiment P and Experiment R), they suffer from extremely low precision (*Obs.1*) and recall (*Obs.15*). We identify four root causes for false negatives, seven root causes for false positives, and two general problems with the design of detectors and how they are typically evaluated. This leads us to several observations on how to advance the state of the art in API-misuse detection. Therefore, we call researchers to action:

- We first need a precise definition of API usages, considering usage properties, such as the usage location (*Obs.6*) and call multiplicities (*Obs.8*).
- We need a representation of such usages that captures all code details necessary to distinguish correct usages from misuses (*Obs.10*) and more precise analyses to identify usages in code (*Obs.4* and *Obs.12*).
- We need detectors that retrieve sufficiently many usage examples using project-external sources, such as large project sets or code-search engines (*Obs.15*).
- We need detectors that go beyond the naive assumption that a deviation from the most-frequent usage corresponds to a misuse (*Obs.3*), but consider program semantics, such as type hierarchies (*Obs.11*) and implicit dependencies between objects (*Obs.7*). We hypothesize that probabilistic models might be a way to tackle this problem.
- We need strategies to properly match patterns and usages in the presence of violations (*Obs.9* and *Obs.11*).
- We need strategies to properly handle alternative patterns for the same API (*Obs.5*).
- Finally, we need good ranking strategies, to reduce the cost of reviewing findings (*Obs.14*).

In order to achieve all this, we need repeatable and replicable studies that enable systematic evaluation and analysis of alternative approaches and strategies. We publish MUBENCH,⁴ as a foundation for such work, and call researchers to use and contribute to our automated benchmark, to advance the state of the art in API-misuse detection.

⁴<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

7.7. Threats to Validity

Construct Validity Our study focuses on static misuse detectors. Approaches based on dynamic analyses may perform differently and have unique strengths and weaknesses. To enable dynamic analyses of the project versions in MUBENCH, we would have to ensure that the respective code is executable (which requires a sufficient runtime environment, in addition to compile time dependencies) and to provide example inputs for the execution. It is unclear how to do this such that it results in a fair comparison of both static and dynamic techniques, without resorting to comparing apples to oranges. Therefore, we chose to focus only on static approaches.

Our experiments focus on detectors that detect misuses in Java code. Therefore, the results may not generalize to detectors for other languages. We decided to focus on this subset of detectors, because the majority of approaches we identified in our survey targets Java. To include detectors that target other languages, we would have to either migrate them to Java or build up additional datasets for the respective languages, either of which is outside the scope of this thesis.

We could not obtain working implementations of all detectors that target Java, since the original prototypes of ALATTIN [TX09a] and CAR-MINER [TX09b] are no longer operational. Therefore, we excluded these detectors from our experiments. Since they are the only to obtain evidence about correct usage from a code-search engine, as opposed to from the target project, our results might not generalize to them. To allow future work to include these detectors (and others) in the comparison, we publish MUBENCH⁵ and all review data from this study.⁶

Any detector’s performance is dependent on its configuration. It is possible that the detectors we compared perform better on our benchmark, given a different configuration. Due to the high effort of reviewing the findings of multiple detectors, we could not try different configurations for each detector. To give each detector a fair chance, we used the optimal configurations reported in the respective publications.

Internal Validity Reviewing the detectors’ findings was done by the author of this thesis and two colleagues and was not blind (i.e., we knew the detectors we were reviewing findings for). We could not blind reviews, because each approach has a distinct representation of usages and violations that cannot be anonymized. Moreover, one of the colleagues is among the original authors of GROUMINER. We did our best to review objectively. To avoid bias, at least two reviewers independently looked at every finding. For the findings of GROUMINER, at least one of these reviewers was not involved in the original work. Judging from the comparably bad performance of GROUMINER in Experiment P and 3, we are convinced that we did not favor this detector over the others.

While we did ask the original authors to confirm our assessment of the conceptual capabilities of their tools (see Chapter 4), we did not ask them to confirm the em-

⁵<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

⁶Artifact Page: A Systematic Evaluation of Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mustudy/>)

7. An Empirical Study of API-Misuse Detectors

pirical results of our experiments. We estimate that, including discussions to resolve disagreements, it required each reviewer on average 2 minutes to verify whether a detector identified one of the known misuses in Experiment RUB and 3 and 5 minutes to verify whether a detector’s finding identifies an actual misuse in Experiment P, where we needed to understand the respective code, check documentation, and sometimes also look into transitively called methods. This amounts to 24.8 hours of review effort per reviewer, 4 hours for JADET, 7.2 hours for GROUMINER, 4.7 hours for TIKANGA, and 8.9 hours for DMMC. We decided it is unreasonable to expect the original authors to invest this amount of time in verifying our assessments. We do, however, publish all our review data⁷ to allow them and others to revisit our decisions.

External Validity The study is subject to the limitations of MUBENCH, as discussed in Section 5.6 and Section 6.4.

We publish MUBENCH⁸ and encourage others to extend the benchmark dataset and repeat our experiments, also with other detectors and detector configurations.

⁷See footnote 6.

⁸See footnote 5.

8. Extension and Further Use

We design MUBENCH as an extensible automated benchmark, to facilitate not only our own study of API-misuse detectors presented in Chapter 7, but also future work. As the most important extension points we consider (1) extensions to the benchmarking dataset, to increase our confidence in the generalizability of the benchmarking results, and (2) integrating additional misuse detectors, to move further towards a comprehensive comparison of existing approaches. In this chapter, we discuss examples of respective work by others and ourselves.

8.1. Dataset Extensions

The benchmark dataset forms the basis for all benchmarking experiments. The representativeness of the software projects it contains as well as of the known misuses examples it refers to are crucial to the generalizability of benchmarking results. Though ideally a benchmark dataset is a minimal representative sample, to the best of our knowledge, it is unclear how such a sample may be determined. Therefore, the best way to approximate representativeness is to extend the dataset by additional projects and further examples of API misuses as they occur in real-world projects and as developer face them in their day-to-day work. In this section, we discuss examples of how we extended the benchmarking dataset. Table 8.1 summarizes these extensions. For further technical details on how to extend MUBENCH, we refer to the project website.¹

¹<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

Table 8.1.: MUBENCH Dataset Extensions by Source, with the Number of Projects (#P), the Number of Project Versions (#PV), and the Number of Misuses (#M).

Source	#P	#PV	#M
Runtime-Verification Study (Section 8.1.1)	13	18	77
Misuse-Detector Evaluations (Section 8.1.2)	2	2	5
Java 8 Misuse Repository (Section 8.1.3)	0	0	5
Original Misuse Collection (Section 8.1.4)	9	9	20
Total	24	29	107

8.1.1. Misuses from a Runtime-Verification Study

Runtime verification of API specifications identifies misuses by monitoring executions against formal specifications. The quality of the verification naturally depends on the quality of the specifications. Legunsen *et al.* [LHX⁺16] studied the precision of runtime verification for misuse detection, using a set of 182 manually written specifications [LJMR12] and 17 automatically mined specifications [PG09] for APIs from the JAVA CLASS LIBRARY. Using these specifications, they applied the runtime verifier JAVAMOP [JMLR12] to the 200 most-popular GITHUB projects that use MAVEN for build automation, have at least on test, and have all tests passing with and without JAVAMOP monitoring. Subsequently, they reviewed 852 (13.3%) of the reported violations and submitted bug reports with fixes, i.e., pull requests, for the 114 violations that they found to be potential bugs.

Integrating these true-positive findings into MUBENCH is interesting in two ways: First, it gives us additional examples of known misuses to evaluate the recall of misuse detectors. Second, it enables an empirically assessment of the fraction of misuses identifiable through runtime verification that can also be detected by static API-misuse detectors, allowing for a limited comparison of the recall of both static and dynamic detectors.

Legunsen *et al.*'s artifact page² lists 114 potential misuses with corresponding pull requests. To reduce the manual effort of adding the misuses examples to MUBENCH, we partly automated the process. The artifact page provides a list of the misuses in CSV format. For each misuse, the list names the specification that the misuse violates and the respective pull request URL. From the specification, we infer the API that is misused and whether the misuse causes a crash. From the pull request, we programmatically retrieve additional misuse metadata, via the GITHUB API³. We capture the respective project's name and repository URL, the commit Id of the project version immediately before the fix was accepted, the problem and fix description, the fix changeset, and the file location of the misuse. We also check whether the pull request has been accepted by the project's developers and filter misuses for which this is not the case. This leaves us with 86 misuses with metadata. We store this data according to the schema presented in Section 6.1.

Afterwards, we manually reviewed each misuse, to validate the data and to add the method location, which we could not automatically determine. We also hand-craft respective examples of correct usage, following the same process as in Section 5.2. This manual review revealed that some of the misuses are duplicates. JAVAMOP analyzes both the target project's own code as well as the code of its dependencies. If it identifies a misuse in a dependency and that dependency is shared by multiple target projects, the misuse is reported once for each target project. This happens, for example, for misuses identified in the TESTNG project. We exclude all 15 respective duplicates. At the same time, we found that in two cases the pull request fixes additional instances of a misuse, which JAVAMOP did not identify. We include all 6 respective misuses as separate

² <http://fs1.cs.illinois.edu/spec-eval/> (checked on Dec 15, 2017)

³ <https://developer.github.com/v3/> (checked on Mar 17, 2018)

misuses. This leaves us with a total of 77 confirmed misuse instances for inclusion in MUBENCH.

Finally, to include the misuse in MUBENCH, we need compile instructions for the respective project versions. Since all projects use MAVEN for their build configuration, we assumed the respective default paths for source and classes files and used the command `mvn compile` for compilation. This was sufficient to compile all-but-two project versions; one compilation failed due to an underspecified dependency version and one due to a file encoding issue. Both problems were easily fixed, which gave us compilable project versions for all confirmed misuses.

Overall, this process resulted in a dataset extension of 77 misuses from 18 versions of 13 projects. One of these projects, namely JODATIME, was already part of the benchmark dataset. It took the author of this thesis and a student assistant less than one week to build this dataset extension, which more than doubled the dataset, in terms of the number of known misuses. We are grateful to Legunsen *et al.* for publishing their dataset, which enabled this extension of MUBENCH.

8.1.2. Misuses from Previous Evaluations of Misuse Detectors

For the cross-comparison of misuse detectors, it is interesting to extend MUBENCH by misuses that detectors identified in previous evaluations, like we added the confirmed misuses from Experiment P for Experiment R (see Section 7.3). Unfortunately, the publications we identified in our survey in Chapter 4 do not report lists of the respective true-positive findings. However, we noticed that Pradel *et al.* [PJAG12] verified whether their detector PJAG12 identifies the true-positive findings of the misuse detectors GROUMINER [NNP⁺09b] and TIKANGA [WZ11]. Therefore, we contacted Michael Pradel about the respective misuse data, which he promptly provided.

Overall, GROUMINER and TIKANGA identified 19 true positives in eight projects. We excluded three projects, because the source code of the respective project versions is no longer available. We exclude two more projects, because we can neither compile them following the process described in Section 5.2 nor obtain binaries for the respective project versions. We exclude another project, because we cannot locate the single misuse it contains. This leaves us with two projects containing five true-positive findings for inclusion in MUBENCH. We added the respective data manually.

We observe that this attempt to extend MUBENCH was of little success mainly due to the unavailability of original evaluation data. In our eyes, this stresses the importance of building and maintaining public benchmarks like MUBENCH.

8.1.3. Java 8 Misuses

The evolution of programming languages certainly impacts the problem of API misuse. For example, the introduction of the `foreach` loop with Java 5 gave developers a language construct that ensures correct iteration of collections, i.e., correct usage of the `Iterator` API. Similarity, the introduction of the `try-with-resources` statement with Java 7 gave developers a language construct that ensures correct closing of IO streams.

8. Extension and Further Use

Conversely, new language versions often introduce new APIs to the `JAVA CLASS LIBRARY`, which may be misused. For example, Java 8 introduced the `Optional` API as an alternative to using `null` for the representation of absent values. While this potentially reduces the need for `null` checks, developers might, in turn, misuse the `Optional` API, e.g., by omitting checks on `Optional.isPresent()`. Java 8 also introduced the `Stream` API to extend the language by elements of the functional-programming paradigm. Many of the methods of this API, such as `filter()` or `map()`, have lazy-evaluation semantics, i.e., invoking them does not actually perform the respective operation. Instead, the operation is deferred until a subsequent terminal operation, e.g., until iterating over the elements in the stream. Developers might misuses the `Stream` API, e.g., by omitting a terminal operation. Alimenkou *et al.* maintain a repository of such Java 8 misuses,⁴ which they identified during code reviews. In this repository, we find compilable, hand-crafted examples of five Java-8-API misuses with explanations and corresponding examples of correct usage. We add all of them to `MUBENCH`.

8.1.4. Misuse from the Original Misuse Collection

When we created the original `MUBENCH` dataset from the misuse examples we collected in Chapter 2, we had to exclude 50% of project versions due compile errors or missing dependencies that we could not resolve within the time limit we gave us per project version. Excluding such a large fraction of the projects might bias the dataset. Therefore, we invested additional time on compiling project versions. We succeeded for nine project versions, which adds 20 further misuses to the set of misuses available for benchmarking experiments.

8.2. Comparison to Further Detectors

To achieve a broad empirical comparison of state-of-the-art misuse detectors, it is crucial that we integrate further detectors into the benchmark. To make this easy, `MUBENCHPIPE` provides a small Java framework, which simplifies the implementation of `MUBENCH` runners, i.e., adapters for the integration of existing detectors into `MUBENCH` (see Section 6.2). The framework handles the entire communication with `MUBENCHPIPE`, such that runners need only pass the experiment input data to the detector and return a list of its findings. For convenience, we provide the framework as a `MAVEN` dependency. For further technical details on how to integrate detectors into `MUBENCH`, we refer to the project website.⁵ In this section, we discuss examples of integrating detectors and present respective benchmarking results.

⁴<https://github.com/xpinjection/java8-misuses> (checked on Mar 20, 2018)

⁵<https://github.com/stg-tud/MUBench/> (checked on Mar 20, 2018)

Table 8.2.: Precision of FINDBUGS Compared to API-Misuse Detectors.

Detector	Findings in Top-20	Confirmed Misuses	Precision	Kappa Score
JADET	39	4	10.3%	0.97
GROUMINER	66	0	0.0%	0.97
TIKANGA	44	5	11.4%	0.93
DMMC	81	8	9.9%	0.91
FINDBUGS	82	43	52.4%	0.83

8.2.1. A Study of Using FindBugs for Misuse Detection

FINDBUGS is a popular⁶ Open Source static code analyzer created by Hovemeyer and Pugh [HP04]. The tool detects possible bugs in Java Bytecode, using a set of predefined bug patterns, which are matched to the target code by dedicated static analyses. The patterns are classified into four categories of severity, namely *scariest*, *scary*, *troubling*, and *of concern*, and the tools' findings are ranked according to this discrete scale.

While FINDBUGS does not implement general techniques to identify particular types of misuses, say, missing method calls, it contains bug patterns that identify misuses for specific APIs. For example, one pattern identifies misuses that leave IO streams open (missing call to `close()`) and another pattern identifies misuses that invoke a SWING UI method outside the UI thread (missing context condition). To determine which of FINDBUGS' bug patterns identify API misuses, the author of this thesis and one student independently reviewed all 424 FINDBUGS bug patterns.⁷ Subsequently, they discussed each pattern that they disagreed on, until an agreement was reached. The process led to a list of 164 bug patterns that identify API misuses.

Since FINDBUGS apparently searches for many API misuses, we want to compare its capabilities to state-of-the-art API-misuse detectors. Therefore, a student implemented a MUBENCH runner to integrate FINDBUGS into the benchmark.⁸ Before developing the runner, the student was completely unfamiliar with both MUBENCH's runner framework and FINDBUGS. It took him approximately four days to complete the task. This enables us to compare FINDBUGS' detection capabilities with state-of-the-art misuse detectors, using the setup described in Chapter 5.

Precision of FindBugs

We first apply FINDBUGS to the five target projects for Experiment P, to determine its precision. We assess the top-20 findings of the tool, following the review process described in Section 5.3. Table 8.2 shows that FINDBUGS reaches a precision of 52.4% in this experiment, which is significantly better than the other detectors.

In total, FINDBUGS identifies 43 true positives among its top-20 findings. The largest category of findings (eleven findings) identifies missing or redundant `null` checks. FINDBUGS identifies these based on present `null` checks within a method that do not guard all usages of the checked variable. The second-largest category (ten findings) identifies usages that ignore a signal return value. For example, when creating directories through `File.mkdirs()`, one should check the return value, which indicates whether all directories were successfully created. Further categories of findings identify usages that perform string concatenation in a loop using the `+` operator instead of a `StringBuilder` (five findings), usages of that invoke the constructors of primitive-type object wrapper classes instead of the more-efficient `valueOf()` methods (four findings), usages that compare strings using the `==` operator instead of the `equals()` method (three findings), and usages that suppress all exceptions (three findings).

Conversely, FINDBUGS report 39 false positives among its top-20 findings. Most false positives (33 findings) are caused by limitations of FINDBUGS' static code analysis, e.g., because the tool does not analyze transitively called methods or because it over-approximates statement reachability. Another two false positives are caused by FINDBUGS's bug patterns being too restrictive. For example, FINDBUGS reports all invocations of `System.exit()` that occur outside a program entrypoint (`main()` method). However, one usage invokes `exit()` in a different method that is explicitly documented as the program entrypoint. The last four false positives are code smells, such as redundant checks, which are not API misuses per se.

These results show that a more targeted analysis using specific bug patterns may increase precision. Nevertheless, the FINDBUGS' precision is still far from perfect. Interestingly, most of its false positives are caused by imprecisions of the underlying static analyses, a root cause that we also identified for 23.9% of the false positives reported by the other detectors (see Section 7.1).

Recall of FindBugs

Since FINDBUGS does not learn from usage examples its recall upper bound we measure in Experiment RUB equals its actual recall. The left part of Table 8.3 shows the recall of FINDBUGS and the recall upper bound of the API-misuse detectors. We find that FINDBUGS misses 84.3% of the misuses in MUBENCH, more than any of the misuse detectors. The single reason for all false negatives is that FINDBUGS does not have a bug pattern that identifies the respective misuses. Therefore, the misuse detectors, which

⁶The FINDBUGS website reports over a million downloads: <http://findbugs.sourceforge.net/index.html> (checked on Oct 18, 2017)

⁷<http://findbugs.sourceforge.net:80/bugDescriptions.html> (checked on Jun 08, 2017)

⁸Repository: <https://github.com/MUBench/FindBugs> (checked on Jun 27, 2018)

Table 8.3.: Recall of FINDBUGS Compared to API-Misuse Detectors.

Detector	Experiment RUB			Experiment R		
	Actual Hits	Recall	Upper Bound	Actual Hits	Recall	Kappa Score
JADET	15	23.4%	0.76	3	2.5%	1.00
GROUMINER	31	48.4%	0.84	0	0.0%	1.00
TIKANGA	13	20.3%	0.84	7	5.8%	1.00
DMMC	15	23.4%	0.85	11	9.1%	0.95
FINDBUGS	10	15.7%	0.92	53	43.8%	1.00

can learn patterns for any API from usage examples, have the potential to outperform FINDBUGS in terms of recall.

For Experiment R we add FINDBUGS’ 43 true positives from Experiment P to the dataset. Since FINDBUGS can detect misuses even if there is no example usage to learn from, we also include the 25 hand-crafted misuses examples, which we previously only considered in Experiment RUB. This gives us a total of 121 misuses to measure recall. The right part of Table 8.3 shows the results. FINDBUGS correctly identifies its own 43 true positives and the ten misuses it also identified in Experiment RUB. This leads to a recall of 43.8%. While this recall is much higher than that of the misuse detectors, we note that FINDBUGS’ own findings accumulate to more than a third of the dataset, which likely biases the results in its favor.

JADET, GROUMINER, TIKANGA, and DMMC identify only the same misuses they already identified in our previous experiment (cf. Section 7.3) and miss all of FINDBUGS’ true positives. Conversely, FINDBUGS misses all true positive findings of the misuse detectors. This suggests that misuse detectors and FINDBUGS should not be considered alternatives for the identification of API misuses, but rather as complementary tools with distinct strengths.

8.2.2. Integration of Salento

Murali *et al.* [MCJ17] recently presented the API-misuse detector SALENTO. We introduce this detector in our survey in Chapter 4. Their promising evaluation results motivated us to try integrating SALENTO into MUBENCH. When we contacted the authors in December 2018 they were enthusiastic about the idea. Unfortunately, their original detector implementation supports only Dalvik Bytecode, while MUBENCH provides only Java Bytecode. For this reason, among others, Murali *et al.* started migrating SALENTO to a different analysis framework. Afterwards, they want to integrate SALENTO into MUBENCH for a respective evaluation. At the point of this writing, the migration is still in progress.

9. Related Work

To the best of our knowledge, MUBENCH is the only existing automated benchmark for Java API misuses. There are, however, several existing reference datasets with general software-defect data and code-analysis benchmark datasets, as well as automated pipelines for the evaluation of defect detectors. There are also other studies that presented some empirical comparison of API-misuse detectors. This chapter presents an overview over the respective work.

Benchmark Datasets As our survey of misuse-detection literature (see Section 4.3) shows, most prior work used custom collections of target projects for the evaluation of the respective detectors. The only dataset that was used for the evaluation of multiple dynamic detectors [GS10, NK11, PJAG12, PG12] is the DACAPO benchmark suite [BGH⁺06]. Unfortunately, this suite does not provide the source code for the contained projects, which we need as the input for some static misuses detectors, such as GROUMINER [NNP⁺09b]. While we could obtain corresponding source code through decompilation, this might introduce bias in the evaluation, since decompiled sources likely differ from human written code. Furthermore, the only known misuses in the DACAPO projects are those identified by said dynamic misuse detectors. Using them as the ground truth for comparing the recall of misuse detectors likely biases the benchmark towards the capabilities of these detectors.

The QUALITAS CORPUS [TAD⁺10], another benchmark suites for static code analyses, provides both source code and Bytecode for each contained project version. Reif *et al.* [REHM17] present an approach to assemble minimal representative datasets of projects for the evaluation of static analyses, which might be adapted to define a representative dataset for benchmarking of misuse detectors. However, in both cases, obtaining a ground truth about existing misuses in the respective dataset remains an open challenge.

Researchers presented various datasets with general software-defect data from software projects, identified in projects' issue trackers or version-control systems. While these datasets are valuable for many research directions, such as triaging of bug reports, defect localization, or defect prediction, they identify all kinds of bugs, which is too general for evaluations of misuse detectors. We use four such datasets, namely BUG-CLASSIFY [HJZ13], DEFECTS4J [JJE14], iBUGS [DZ07], and QACRASHFIX [GZW⁺15], to study the prevalence of API misuses compared to other types of bugs (see Section 2.2) and derive many of the misuse examples in our benchmark dataset from the results of this study (see Section 5.2). Moreover, we find that developers struggle with API misuses that may not ever be committed to version-control systems and, consequently, may never be mentioned in issue trackers (see Section 2.3). Considering only misuse examples observable in these sources might miss important cases and bias the evaluation of

9. Related Work

API-misuse detectors. Therefore, we add examples from a developer survey and a study of documented API constraints to our benchmark dataset as well (see Section 5.2).

Automated Benchmarks To the best of our knowledge, there is no existing automated benchmark for any type of Java software defects. There are, however, two such projects for other programming languages:

BUGBENCH [LLQ⁺05] is a defect-detection benchmark of C and C++ programs. Its dataset names 13 memory-related defects, four concurrency bugs, and two logic bugs from 17 programs. It comes with a pipeline that automates an experiment to measure detector accuracy, which is used to measure and compare the accuracy of 13 detectors.

BEBUNCH [CHK⁺09] is a defect-detection benchmark of C programs. Its dataset identifies several defects from the categories of buffer overflows, memory and pointer bugs, integer overflows, and faulty format strings. It comes with a pipeline that automates an experiment to measure detector accuracy and another experiment to measure detectors’ runtime performance and scalability.

While neither of these benchmarks focuses on API misuses, the underlying idea of providing a pipeline that automates as much as possible of the evaluation process, to make it repeatable and comparable, motivated the development of MUBENCHPIPE.

Comparison of Misuse Detectors To the best of our knowledge, the only prior work that directly compared the results of multiple API misuse detectors is a study by Pradel *et al.* [PJAG12]. They evaluate their detector PJAG12 on the same target projects as were used in the evaluations of GROUMINER and TIKANGA. They show that PJAG12 successfully identifies six misuses that GROUMINER misses, but misses four misuses that GROUMINER identifies, while there is not a single misuse identified by both detectors. In comparison to TIKANGA, PJAG12 identifies nine additional misuses, but misses two, while three misuses are identified by both detectors. As the main reason for their additional findings they identify that PJAG12 is able to capture calls that are illegal in a particular state of an object, i.e., calls that are missing a precondition, which both other detector cannot, by design. Furthermore, they find that all three detectors miss some misuses, because they did not mine the respective specifications. To address this issue, they suggest to mine larger sets of programs or to generate additional mining input, e.g., through automated test generation to obtain more execution traces for dynamic detectors. We confirm these findings on the basis of a larger dataset and including two further detectors, namely JADET and DMMC, and provide additional insights on the specific problems of static misuse detectors. Additionally, with MUBENCH, we provide an automated pipeline to conduct similar comparisons with less manual effort, including additional detectors and using a larger dataset.

Legunsen *et al.* [LHX⁺16] present a study of the effectiveness of both manually written and mined specifications for misuse detection. While this work does not provide an explicit comparison of different detectors, it provides general insights on the practical capabilities of misuse detectors based on runtime verification. They find that, while runtime verification identifies actual misuses, it also produces large number of false positives,

namely 82.8% for the manually written specifications and 97.9% for the mined specification. They report that only a small number of all specifications effectively identify true positives. These results suggest that the problems of dynamic API-misuse detectors might be quite similar to those of static detectors. We integrated the true positives from Legunsen’s study into MUBENCH (see Section 8.1.1), to enlarge our benchmark dataset and to enable a more direct comparison between static and dynamic misuse detectors in future work.

Part III.

MuDetect

The Next Step in Static API-Misuse Detection

Despite the amount of previous work, API misuse remains a problem in practice [LHX⁺16, ABF⁺16]. Therefore, in Chapter 7 of this thesis, we empirically evaluated and compared existing API-misuse detectors using our automated benchmark MUBENCH (Chapter 5). In this study, we identified several major problems, that affect precision and recall:

- Detectors fail to distinguish correct usages from misuses, due to not capturing sufficient code details in their representations.
- Detectors ignore alternative usage patterns and semantically correct deviations, e.g., using a subtype versus a supertype, naively assuming that all deviations from usage patterns are misuses.
- Detectors often fail to relate misuses with patterns, because the difference between the two exceeds an assumed threshold.
- Detectors have poor ranking strategies that rank many false-positive findings higher than true-positive ones.

Furthermore, the study observed that evaluations mostly apply detectors in a per-project setting, where they mine usage patterns solely from the project in which they detect misuses. A presented hypothesis is that individual projects contain too few usages examples to mine good patterns, which limits the detectors' recall. All these problems need to be properly addressed for API-misuse detectors to be practically useful. In Chapter 10, we discuss these problems in more detail and sketch how we address them with the new API-misuse detector that we introduce in this part of the thesis.

In Chapter 11, we present MUDTECT, a new API-misuse detector that we designed based on the strengths and deficiencies of existing detectors. MUDTECT encodes API usages as API Usage Graphs (AUGs), a new, comprehensive usage representation designed to capture the differences between correct usages and misuses. MUDTECT employs a greedy, frequent-subgraph-mining algorithm to mine patterns and a specialized graph-matching strategy to identify (violating) occurrences of patterns. Both components consider code semantics of API usages to improve the overall detection capabilities. On top, MUDTECT uses an empirically optimized ranking strategy to effectively report true positives among its top-ranked findings.

We investigated and designed our solution via learning from the findings of detectors that we evaluated against MUBENCH in Chapter 7. We then extended MUBENCH by 107 misuses identified in a recent study on runtime verification [LHX⁺16], which more

than doubles its size. We assess and compare the performance of MUDETECT and four other state-of-the-art detectors on this extended benchmark. In Chapter 12, we present the evaluation setup we use to this end. In Chapter 13, we present the respective results, which show that MUDETECT significantly outperforms the other detectors in terms of both precision and recall. In a setting with perfect training data, MUDETECT achieves a recall of 72.5%, which is 20.3% higher than the second-best detector and over 50% higher than the other three detectors. In the typical per-project setting, MUDETECT achieves a recall of 20.9%, which is 10.2% better than the second-best detector, and a precision of 21.9%, which is 13.1% better than the second-best detector. In a cross-project setting, where we use a large quantity of usage examples from 3rd-party projects for pattern mining, MUDETECT achieves a recall of 42.2% and a precision of 34.1%, again significantly outranking its own performance from the per-project setting.

These results show that our systematic design approach, based on the strengths and deficiencies of existing detectors, successfully led us to a significantly better detector, while balancing recall and precision. The most-important decision was to separate pattern mining and violation detection, which allowed us to train MUDETECT on cross-project usage examples. Future work should focus more on providing detectors with high-quality training examples.

In Chapter 14, we discuss the threats to the validity of our results and in Chapter 15, to conclude this part, we provide an overview over related work on API-misuse detection.

10. Motivation

Our goal is to design a new misuse detector that addresses problems of the detectors GROUMINER, JADET, TIKANGA, and DMMC, as identified in Chapter 7. Section 4.2 contains a detailed description of these detectors. We briefly reintroduce them here.

GROUMINER [NNP⁺09b] represents usages as directed acyclic graphs that encode method calls, field accesses, and control structures as nodes and control-flow and data-flow dependencies among them as unlabeled edges. GROUMINER uses sub-graph mining to find patterns and then detects violations of these patterns as missing nodes. It detects missing method calls, as well as missing conditions on the granularity of a missing branching or loop node.

JADET [WZL07] encodes the transitive closure of the call-order relation in each usage as pairs of the form $m() \prec n()$. It uses Formal Concept Analysis [GW97] to identify violations, i.e., rarely missing call order pairs. TIKANGA [WZ11] builds on the same algorithm as JADET, but encodes usages using temporal properties (CTL). Both detectors detect missing calls. However, JADET cannot detect violations of patterns with only two calls, because it works on multiple call pairs, since this would be a usage with a single call, which cannot be encoded using call pairs. TIKANGA can detect such violations.

DMMC [MM13] encodes usages as sets of methods called on the same receiver type. It identifies violations by computing, for every usage, the ratio between the number of equal usages and the number of usages with exactly one additional call. Intuitively, a violation should have only few exactly-similar usages, but many almost-similar usages. DMMC detects misuses with exactly one missing method call.

The empirical evaluation and comparison of these four detectors revealed several problems that result in low recall and precision (Chapter 7). We now give a brief description of these problems and how we mitigate them.

Problem P1: Imprecise Representation On average, 45.8% of the false negatives were due to the inability of the detectors' underlying representations to capture details necessary for differentiating misuses from correct usages (see Section 7.2). For example, DMMC and GROUMINER encode methods by their names only and, hence, cannot detect a missing method call when an overloaded version of the method is called (e.g., the pattern requires `String.getBytes(String)`, while the misuse calls `String.getBytes()`). MUDetect's representation tracks method-call arguments. Additionally, for the first time, we provide a representation that combines tracking of control flow, exceptional flow, order of method calls, synchronization, and data flow. Such features have typically been captured in isolation by previous detectors, but never all combined in one representation.

Problem P2: Imprecise Pattern Matching On average, 31.3% of false negatives were due to detectors not matching patterns to misuses (see Section 7.2). For example, to identify that two methods are called in the wrong order, say `b();a()` instead of `a();b()`, a detector needs to both capture the call order in its representation and match the pattern and misuse despite the different order. Similarly, a detector needs to consider sub-typing information to match a `Collections.size()` call found in a pattern to an `ArrayList.size()` call found in a usage. Another issue is that some detectors use a distance threshold to filter their findings, which may filter true positives, e.g., if a misuse contains additional, optional method calls. *MuDETECT matches calls even if their order differs, considers type-hierarchy information, and does not employ a distance threshold, but rather enforces a ranking strategy.*

Problem P3: Uncommon-but-correct Usage On average, 34.3% of the false positives were uncommon-but-correct usages (see Section 7.1). It is generally difficult, if not impossible, to automatically and precisely distinguish uncommon usages from misuses. However, we observed that many of the false positives for uncommon usages involved methods without side effects (*pure methods*), such as getters. These violation typically report that a pure method call is missing or that there is a wrong call order of several pure methods. For example, `MapEntry.getKey()` and `MapEntry.getValue()` often occur together, but there is neither a requirement that both are called in the same usage nor that these call occur in a particular order. Since invocations of pure methods cannot be required, unless their return value is actually needed, *MuDETECT removes calls to pure methods from patterns, unless their return value is used in the pattern.*

Problem P4: Alternative Patterns On average, 19% of false positives were usages that deviated from some individual pattern, but conformed to another pattern (see Section 7.1). Other previous work reports that alternative-pattern instances may even accumulate to 28% of the false positives [TX09a]. Therefore, *MuDETECT filters alternative-pattern instances.* In addition, *MuDETECT abstracts over some usage variants to reduce the number of alternative patterns.*

Problem P5: Self- and Cross-method Usages On average, 12.2% of false positives were due to self- and cross-method usages (see Section 7.1). In a *self usage*, a class uses part of its own API in its implementation, e.g., `Collection.addAll()` calls `Collection.add()`. Constraints that client usages must adhere to, e.g., guarding calls by checks, may not apply to self usages. In a *cross-method usage* an object in fields is used across methods, e.g., initialized in the constructor and used in other methods. From the perspective of an individual method, we have only a partial view on the entire usage scattered across methods. For both types of usages, an intra-procedural analysis potentially detects partial usages, i.e., violations, that are unlikely to be actual misuses. Therefore, *MuDETECT ignores usages on the API's own implementation.*

Problem P6: Call Multiplicity On average, 1.4% of the false positives were due to the inability of detectors to distinguish methods that must be called exactly n times from methods that must be called at least once and methods that may be called arbitrarily often (see Section 7.1). In its usage representation, MUDetect *distinguishes whether a method is not called, called once, or called multiple times consecutively*.

Problem P7: Poor Ranking Often, the detectors correctly identified misuses, but ranked them extremely low (see Section 7.3). Since developers are unlikely to go through hundreds of findings, an effective ranking mechanism that pushes true positives to the top is essential. *We empirically investigate several ranking factors from the literature through MUDetect and compose a ranking strategy that effectively prefers true positives.*

Problem P8: Lack of Usage Examples A possible cause of the detectors' low recall is a lack of correct usages in the target projects that they train on (see Section 7.3). To validate this hypothesis, *we evaluate MUDetect in both a per-project setting, which is the norm in the literature, and a cross-project setting that provides more usage examples for training.*

11. A New Detector

We design a new API-misuse detector, MUDETECT, to address the problems summarized in Chapter 10 as follows:

1. We design API-Usage Graphs (AUGs), a new representation of API usages that simultaneously captures many properties of API usages that can distinguish misuses from correct usages.
2. We design a new pattern-mining algorithm based on the idea of frequent-subgraph mining that exploits domain knowledge about API usages to efficiently identify usage patterns.
3. We design a new violation-detection algorithm, which uses domain knowledge to efficiently identify API-usage violations.
4. We design a ranking strategy based on factors from the literature that effectively ranks true positives before false positives.

We subsequently introduce MUDETECT’s four components, one at a time.

11.1. API-Usage Graphs

Graph-based representations are well-suited to simultaneously encode different aspects of API usages, however, existing graph representations do not capture all details necessary to identify API misuses (see Section 7.2). To address the problem of insufficient details in the usage representation (*P1* (Imprecise Representation)), we propose API-Usage Graphs (AUGs), a new representation of API usages. An AUG is a directed, acyclic, connected multigraph with labeled nodes and edges. Nodes represent data entities, such as variables, and actions, such as method calls; edges represent control and data flow between entities and actions represented by nodes. In the following, we discuss all AUG elements in detail using the AUG in Figure 11.1 for illustration.

11.1.1. Usage Actions

We use *action nodes* to represent method calls, operators, and instructions in API usages (boxed shapes in Figure 11.1). We encode method calls, because they are the primary components of APIs. We treat field accesses like method calls, because we regard them as calls to respective getter or setter methods. For method calls, we use labels of the form $T.M()$, where M is the method’s name and T is the simple name of its declaring

11. A New Detector

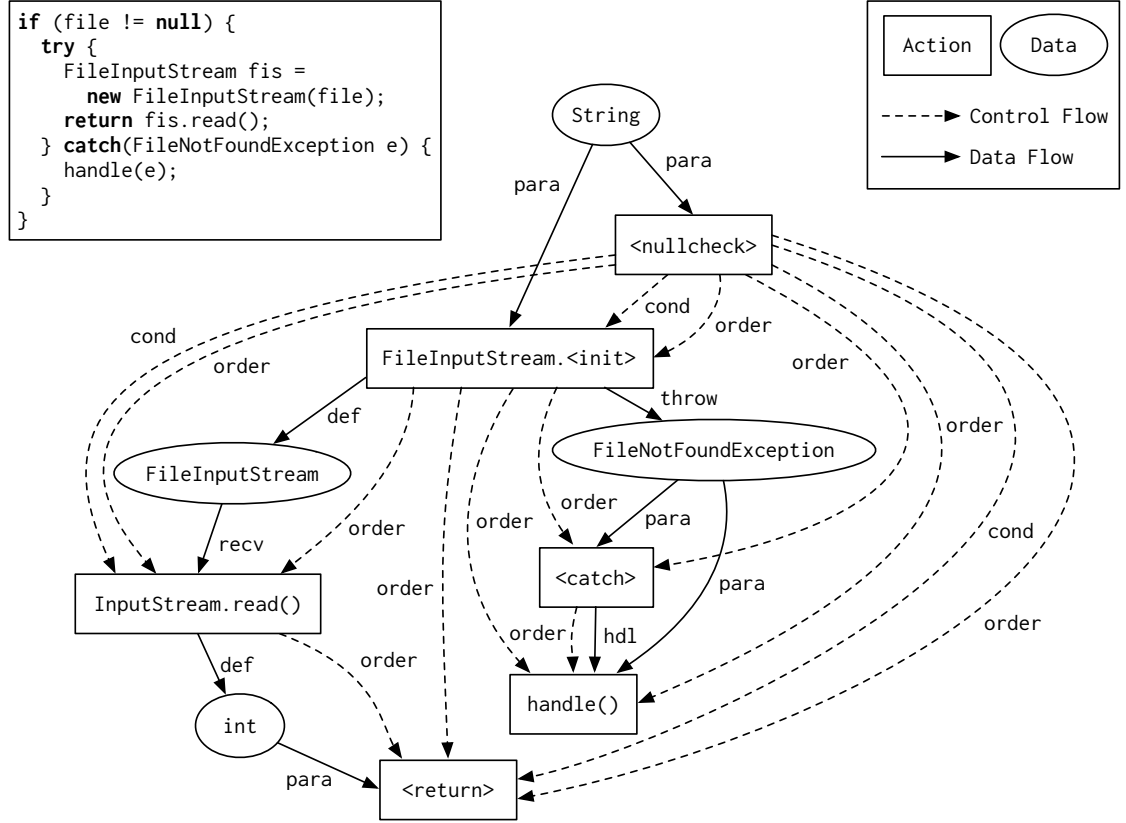


Figure 11.1.: A Code Fragment and its API-Usage Graph.

type. For constructor calls, we use labels of the form $T.<init>$. Using the declaring type abstracts over different static receiver types ($P2$ (Imprecise Pattern Matching)), since, for example, all calls to `size()` on a `List`, `LinkedList`, or `ArrayList` are labeled `Collection.size()`.

We encode equality and relational operators to capture conditions such as `var != null` or `list.size() > 0`. To abstract over alternative ways to express the same condition, e.g., `l.size() != 0` and `!(l.size() == 0)`, we use the label $<r>$ for all equality and relational operators and drop the unary negation operator. To further abstract over alternative ways to compose conditions, e.g., `a && b` and `!(!a || !b)`, we drop the conditional operators `&&` and `||`. With this abstraction level, we focus on detecting the absence or presence of conditions in API usages, rather than subtle logical mistakes in the conditions. We explicitly capture `null` checks, e.g., the `null` check on `file` in Figure 11.1, by action nodes with the dedicated label $<nullcheck>$ to distinguish this special kind of condition from comparisons with values or other literals. We drop arithmetic and bitwise operators, because their usage is mostly related to program logic rather than API usage constraints. We encode unconditional control instructions, such as `return`, `throw`, and `catch`, by action nodes with dedicated labels, e.g., the $<catch>$

and `<return>` nodes in Figure 11.1.

11.1.2. Data Entities

We use *data nodes* to represent objects, values, and literals that appear in API usages (circular shapes in Figure 11.1). We encode data entities as nodes to make data dependencies between actions, such as multiple method calls on the same object, explicit. This ensures that we have a connected subgraph for each usage of an individual object, regardless of the order of method calls, which improves our ability to match usages (*P2* (Imprecise Pattern Matching)). It also enables us to distinguish overloaded versions of methods by their parameter entities (*P1* (Imprecise Representation)).

We uniformly create data nodes for local variables and fields, as well as for values and objects that are not explicitly assigned but directly used, e.g., in a method-call chain. For example, Figure 11.1 shows a data node for the `FileInputStream` assigned to the local variable `fis`, for the `FileNotFoundException` captured as `e`, as well as for the `int` value that is returned from `read()` and directly passed on as the snippet's return value.

Since certain types, such as `List`, `ArrayList`, and `LinkedList`, appear almost interchangeably in API usages, we decided to label all data nodes by the label `<Object>`. This allows us to abstract over different static types (*P2* (Imprecise Pattern Matching)), while checking the data- and control-flow that the data entities take part in. Note that Figure 11.1 shows the simple type names for better readability.

We use the label `<Object>` also for all data nodes created from literals, to unify over equivalent alternatives. For example, both `l.size() > 0` and `l.size() >= 1` represent the same condition using different literals and both `b.setEnabled(true)` and `b.setEnabled(b)` are instances of the same usage, where one uses a literal and the other a value.

11.1.3. Control Flow and Data Flow

We use edges to represent control flow and data flow. We distinguish nine types of edges and label them with their type. Figure 11.1 shows seven of these edge types, labeled with acronyms for brevity.

- A *receiver* edge connects a data node to a method call that is invoked on the respective object.
- A *parameter* edge connects a data node to an action that takes the respective object or value as a parameter.
- A *definition* edge connects an action that creates or returns a value or object to the respective data node.
- An *order* edge connects, in the order of their execution, two action nodes operating on the same data entity (receiver or parameter). Since we want MUDETECT to discover wrong method-call order, we over-approximate temporal relations between

11. A New Detector

actions by building the transitive closure over *order* edges. To keep AUGs acyclic, we exclude backwards edges from loops.

- A *condition* edge connects an action **a** whose result is considered for a branching to all actions that depend on that branching.
- A *synchronize* edge connects a data node that the program obtains a lock on to all actions executed while holding the lock.
- A *throw* edge connects an action that may throw an exception to a data node representing that exception object. We use the **throws** information, if it is resolvable, to determine which exception may be thrown by an action. We connect exception data nodes to respective **<catch>** nodes with parameter edges.
- A *handle* edge connects a **<catch>** node to all actions in the respective exception handling block.
- A *contain* edge connects a data node representing an anonymous-class instance to the data nodes representing each of its declared methods and a data node representing such a declared method to all actions in its body.

This detailed dependency information helps us to distinguish misuses from correct usages (*P1* (Imprecise Representation)), to relate usages despite notational differences (*P2* (Imprecise Pattern Matching)), and to consider code semantics in both pattern mining and violation detection.

11.1.4. Building API-Usage Graphs

We implemented a transformation from Java source code to AUGs. This transformation takes a single source file or a source-file directory as input. It parses each source file into an abstract syntax tree (AST) and traverses the AST to generate one AUG for each method body in the file, using an intra-procedural analysis. The transformation optionally takes a classpath of the source code's dependencies, such as it would be available in an IDE, for type resolution. If type information is unavailable, it falls back to the partial information available in the source code. For each method in the source code, the transformation proceeds in the following steps.

1. We create action nodes for all the method calls, operators, and instruction. To reduce false positives due to *P5* (Self- and Cross-method Usages), we heuristically exclude self- and cross-method usages from AUGs: We skip method calls on **this** and **super** as well as on field accesses on both these qualifiers when creating action nodes.
2. We create data nodes for all the objects and values and connect actions that produce data to the respective data nodes through *definition* edges and data nodes that are consumed by actions to the respective action nodes through *receiver* and *parameter* edges. For example, in Figure 11.1 the **FileInputStream** receives a

`read()` method call, which produces an `int` as a result, which is then passed as a parameter to the `return`.

3. We create *order* edges between each pair of actions that share a receiver or parameter data node, in the order of their execution. Since our intra-procedural analysis is likely to miss data dependencies and, thereby, order relations, we build the transitive closure over *order* edges. This conservatively over-approximates the order relation.
4. We add *condition* edges from each action whose result is checked in an `if`-condition or loop header to all actions within the controls code block(s). This encodes that the execution of these actions depends on the check, as, for example, the creation and use of the `FileInputStream` happens only if the `file` variable is not `null` in Figure 11.1. For consistency, we translate `foreach` loops into the corresponding `Iterator` usages, such that they, too, have a condition that we may capture with a *condition* edge. This unification abstracts over two alternative usage patterns (*P4* (Alternative Patterns)). If an action's result is used in a condition that guards a `return` or `throw` instruction (or `continue` or `break` within a loop), we also add *condition* edges to all actions after the condition (within the respective loop), because depending on the result these may or may not get executed.
5. We use the *throws* information, if it is resolvable, to determine which exceptions may be thrown and create *throw* edges from the call nodes to the respective exception data nodes. If there is a respective `catch` block capturing this type of exception, we add a *parameter* edge from the exception data node to the respective `<catch>` node. Furthermore, we add *handle* edges from each `<catch>` node to all actions within the respective code block. This happens for example, for the `FileNotFoundException` in Figure 11.1. We translate the `try-with-resources` statement into a `try/finally` statement to abstract over manual and automatic resource closing. This unification abstracts over two alternative usage patterns (*P4* (Alternative Patterns)).
6. If the code synchronizes on some object, we create *synchronize* edges from the respective data node to all actions within the `synchronize` block.
7. If some code appears within a method of an anonymous-class instance, we create a data node to present that method. We add *contain* edges from the data node representing the instance to this method node and from the method node to all action nodes within the method. This is motivated by how GROUMINER detects missing context conditions through its encoding of anonymous inner classes (see the unexpected hits discussed in Section 7.2), but makes the representation more explicit. Figure 11.2 shows an example.

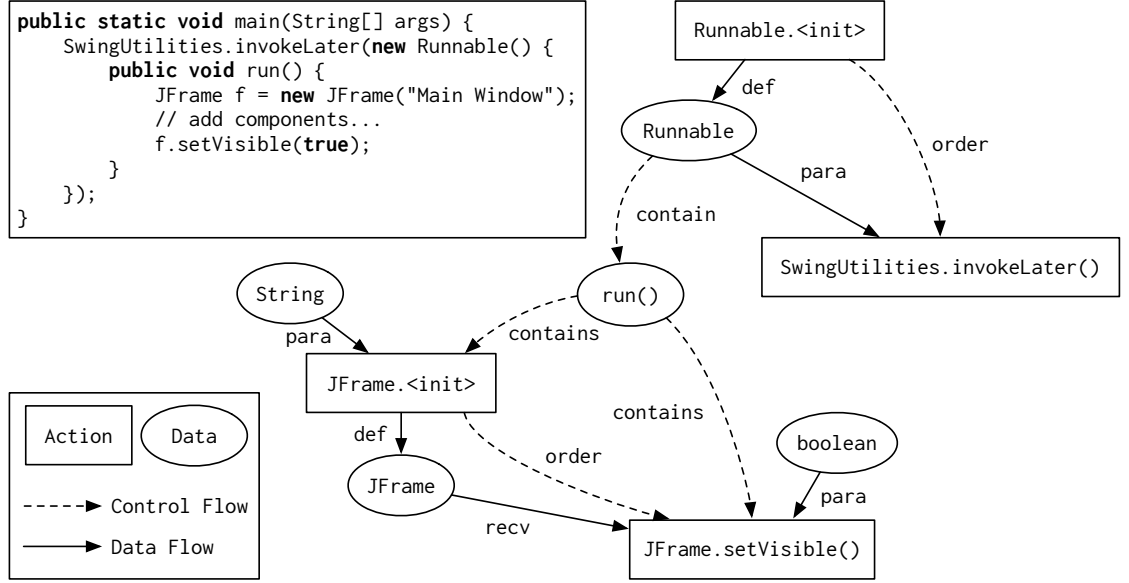


Figure 11.2.: Representation of Anonymous-Class Instances in AUGs.

11.2. Pattern Mining

Algorithm 11.1 shows our pattern-mining algorithm, which takes a set A of AUGs, a frequency measure f , and a frequency threshold σ and produces a set of frequent patterns. A *pattern* is an AUG that reoccurs as a subgraph in A , and a *pattern instance* is one such occurrence. We consider a pattern p as frequent if its frequency $f(p)$ is greater or equal to σ . The algorithm is (1) an instance of a-priori-based frequent-subgraph mining that (2) considers action semantics while (3) greedily exploring pattern alternatives. We subsequently discuss these three key ideas in detail.

11.2.1. Apriori-based Mining

The algorithm follows the general idea of an a-priori-based algorithm for frequent-subgraph mining [RP15], i.e., it mines patterns by starting from all single-method-call patterns (Line 2) and recursively extending them to larger patterns (Line 6). The key idea here is that if a graph occurs frequently, all of its subgraphs also occur frequently. To extend a pattern p of size k , the algorithm generates all suitable extensions of size $k + 1$ for all pattern instances of p (Line 12), by exploring each adjacent node (Line 31). An *adjacent node* to a pattern instance i in an AUG a is a node from a that is not in i and has an edge from or to a node in i . When i is extended by an adjacent node, all edges from a that connect i and the node are added as well. Extending by nodes, as opposed to by edges, enables scalable mining of AUGs, which usually have a large number of edges.

```

1  def mine(A: Set[AUG], f: Pattern → int, σ: int)
2    P0 = {p | p ∈ single_call_node_patterns(A) and f(p) ≥ σ}
3    P = ∅
4
5    for p in P0:
6      extend(p, P, f, σ)
7
8    return P
9
10
11 def extend(p: Pattern, P: Set[Pattern], f: Pattern → int, σ: int)
12   E = {e | i ∈ p and e ∈ generate_extensions(i)}
13   PC = {c | c ∈ isomorphic_clusters(E) and f(c) ≥ σ}
14   UC = PC \ P
15
16   if UC ≠ ∅:
17     c = most_frequent(UC)
18     extend(c, P, σ)
19   else:
20     P = P ∪ {p}
21
22   ip = {i | i ∈ p and ∀c ∈ PC. generate_extensions(i) ∩ c = ∅}
23
24   if f(ip) ≥ σ:
25     P = P ∪ {ip}
26
27
28 def generate_extensions(i: Instance)
29   extensions = ∅
30
31   for n in adjacent_nodes(i):
32     if has_non-order_connection(n, i)
33       and (not is_consecutive_call(n, i))
34       and ((not is_pure_method_call(n))
35         or (is_pure_method_call(n) and has_out_connection(n, i)):
36         or (is_operation_node(n) and has_in_and_out_connection(n, i))
37         or (is_data_node(n) and has_out_connection(n, i))):
38       extensions = extensions ∪ {i ⊕ n}
39
40   return extensions

```

Algorithm 11.1.: MUDETECT's Pattern-Mining Algorithm.

11.2.2. Code Semantics

When extending a pattern instance i , the algorithm distinguishes different types of adjacent nodes. Specifically, the algorithm decides whether an adjacent node is suitable for extending i as follows:

- A node that is only connected by an order edge is unsuitable (Line 32). This prevents MUDETECT from mining patterns with data-independent parts, since we found that the relations between such parts never correspond to API usage constraints.
- A method call $m()$ is also unsuitable, if the extension of i by that call contains two consecutive calls to the same method $m()$ (Line 33). This prevents MUDETECT from reporting violations where the pattern would have, for example, four calls to $m()$ while the target code has only three calls (*P6 (Call Multiplicity)*).
- Otherwise, a non-pure method call is always suitable (Line 34). Recall that a method is *non-pure* when it has a side effect, i.e., when its presence in the usage implies some observable action.
- A pure method call is suitable only if it has an outgoing edge to a node in i (Line 35), i.e., if it defines a data node or controls an action node in i . Since pure methods have no side effects they can impact a usage only through their return value. To avoid the complexity of inter-procedural analysis, the algorithm identifies pure methods heuristically: It considers any method whose name starts with `get` as pure, since getter methods are mostly pure and very prevalent.
- An operator is suitable only if it has at least one incoming and one outgoing edge to i (Line 36), because operators are like pure methods whose result is based solely on their parameters, as opposed to parameters and state.
- A data node is suitable only if it has an outgoing edge to i (Line 37), because a data node does not impact a usage unless it is used in it.

These decisions based on code semantics contribute to obtaining meaningful patterns, thereby mitigating the problem of flagging uncommon usages as misuses (*P3 (Uncommon-but-correct Usage)*).

11.2.3. Greedy Exploration

To identify $(k+1)$ -patterns in the set of all extensions of the instances of p , the algorithm clusters equivalent extensions, i.e., isomorphic graphs, to pattern candidates, of which it keeps the frequent ones (Line 13). To reduce the complexity of graph isomorphism detection, the algorithm uses a heuristic that combines graph vectorization and hashing [NNP⁺09a]. More specifically, a graph is represented as a vector of features, each of which is extracted from the labels of a sequence of nodes and edges along a path in the

graph. Two graphs are isomorphic if their corresponding feature vectors have the same hash value.

The algorithm then filters out all candidates that it found before (Line 14). If there are no further frequent extensions of p , i.e., p is inextensible, the pattern is added to the set of final patterns P (Line 20). If any unexplored candidate remains (Line 16), the algorithm selects the most-frequent one (Line 17) and recursively searches for larger patterns (Line 18). This greedy strategy avoids the combinatorial explosion problem of exhaustive search with backtracking and makes our mining scale to a large number of large graphs, unlike GROUMINER, which times out on many target projects (see Section 7.1).

In addition to the possible extensions, the algorithm also keeps track of those instances that do not have any frequent extension (Line 22). If these inextensible pattern instances form themselves a frequent pattern, it adds this pattern to the set of final pattern P (Line 25). This follows an observation by Lindig *et al.* [Lin07], who found that an API might have a core pattern and additional alternative patterns that contain this core pattern. Considering both ensures that all these alternative patterns are found (P_4 (Alternative Patterns)).

11.3. Violation Detection

Algorithm 11.2 shows our detection algorithm, which takes a set T of target AUGs, a set P of patterns, and a ranking function r and produces a list of violations. A *violation* is a strict subgraph of a pattern, i.e., an AUG with at least one missing edge. Note that since AUGs are connected graphs, a missing node always implies at least one missing edge. The detection algorithm (1) identifies full occurrences (instances) and partial occurrences (violations) of patterns in a target codebase, (2) eliminates violations that are instances of an alternative pattern, (3) ranks the remaining violations, and (4) filters alternative violations of the same usage. We subsequently discuss these four steps in detail.

11.3.1. Graph Matching

To find violations of the patterns in the targets, the detection algorithm checks each pair of a target and a pattern for common subgraphs (Line 8). To identify the subgraphs, the algorithm follows the general idea of the pattern-growth approach for frequent-subgraph mining [RP15], i.e., it discovers the largest common subgraphs of each pair of a pattern and a target (Line 8), by starting from all common method-call nodes (Line 20) and recursively extending the common subgraph (Line 21), one adjacent edge at a time. This allows us to find even single missing edges, e.g., in case of a wrong order of two method calls.

When searching for possible mappings of a pattern AUG onto a target AUG, the detection algorithm follows a greedy extension strategy. It continuously selects the next-best pattern edge, while exploring all alternative mappings to the target. This avoids the combinatorial explosion problem of an exhaustive search with backtracking. The

11. A New Detector

```

1  def find_violations(T: Set[AUG], P: Set[Pattern],
2      r: (Set[Violation], Set[Instance], Set[Pattern]) → List[Violation]):
3      V = ∅
4      I = ∅
5
6      for target in T:
7          for pattern in P:
8              for overlap in common_subgraphs(target, pattern):
9                  if overlap = pattern:
10                     I = I ∪ {Instance(target, pattern)}
11                  elif overlap ⊂ pattern:
12                     V = V ∪ {Violation(target, pattern, overlap)}
13
14      VA = filter_alternatives(V, I)
15      VR = r(VA, I, P)
16      return filter_alternative_violations(VR)
17
18
19 def common_subgraphs(t: AUG, p: Pattern):
20     S = single_call_node_overlaps(t, p)
21     return {lcs | lcs ∈ extend_subgraph(s, t, p) and s ∈ S}
22
23
24 def extend_subgraph(o: Overlap, t: AUG, p: Pattern)
25     es = next_extension_edge(o, t, p)
26
27     if e ≠ none:
28         return extend_subgraph((o ⊕ n), t, p)
29     else:
30         return i
31
32
33 def next_extension_edge(o: Overlap, t: AUG, p: Pattern):
34     ebest = none,
35     wmin = inf
36
37     for e in adjacent_edges(o, t):
38         w = count_equivalent_edges(e, t) * count_equivalent_edges(e, p)
39         if (0 < w and w < wmin)
40             ebest = e
41             wmin = w
42
43     return ebest

```

Algorithm 11.2.: MUDetect's Detection Algorithm.

algorithm explores all alternatives in the target, as opposed to in the pattern, because targets are usually larger and, therefore, likely contain more alternatives. This results in higher precision. Nevertheless, the algorithm might discover a non-optimal mapping between a pattern and a target, producing a false positive. This may happen whenever there are multiple equivalent candidate edges for extension. Two edges are equivalent if they have the same type, both their source and target nodes have the same label, respectively, and mapping them onto each other is consistent with the current mapping between target and pattern nodes. The node mapping is consistent if every node from the target is mapped to at most one node from the pattern and vice versa. Intuitively, the larger the number of equivalent edges is, the more alternative mappings there are and the more likely it is for the algorithm to select a non-optimal mapping. To decrease the likelihood of this to happen, the algorithm counts the number of equivalent edges in the target and the pattern (Line 38) and gives priority to edges with fewer equivalent alternatives. Mapping these first potentially eliminates equivalent alternatives, since additional nodes get mapped and some alternatives become inconsistent with the node mapping.

11.3.2. Alternative-Pattern Instances

There may be multiple alternative ways to use an API. For example, before fetching an item from a `Set`, we may either check that it is not empty or that its size is larger than zero. If we have patterns for both cases, these overlap, since fetching an item requires the same method calls in both cases. Consequently, an instance of one of the patterns necessarily violates the other pattern (*P4* (Alternative Patterns)), because the instance shares elements with both patterns and either misses the check for emptiness or the check of the size. Following this insight, our detection algorithm separates all common subgraphs of a target and a pattern into (1) pattern instances, i.e., target subgraphs equal to the pattern (Line 9), and (2) violations, i.e., strict subgraphs of the pattern in the target (Line 11). When all targets and patterns have been processed, it uses the set of instances to filter violations that are subgraphs of instances of another pattern (Line 14).

11.3.3. Ranking Violations

After identifying all violations in the target code base, the detection algorithm ranks the findings (Line 15). The concrete ranking strategy is a parameter to our detection algorithm. This allows us to compare how different ranking strategies impacts MUDetect's performance. A detailed discussion of possible ranking strategies follows in Section 11.4.

11.3.4. Alternative Violations

If a usage violates all alternative patterns, the previous steps produce multiple violations for the same usage, one for each alternative pattern. Note that filtering instances of alternative patterns, as described in Section 11.3.2, leaves all these violations in place.

11. A New Detector

To avoid reporting such duplicate violations, we filter out any violation that involves a method call that is already part of another violation at a higher rank (Line 16). In this step, the algorithm might remove true positives, if a false positive involving the same usage elements is ranked higher.

11.4. Ranking

Ranking the detected violations is crucial for MUDETECT’s precision, since it controls how many true positives are ranked among the top findings (*P7* (Poor Ranking)). The ranking may also impact MUDETECT’s recall, since we filter alternative violations based on the ranking order of the findings (see Section 11.3.4), which may eliminate true positives. To design MUDETECT’s ranking strategy, we first survey existing ranking strategies and discuss their individual factors. Then, we compose new ranking strategies from these factors.

Distance Some detectors use a maximal distance between a pattern and a usage to classify it as a violation of the pattern [WZL07, NNP⁺09b, WZ11]. Here, the distance is the number of facts from the pattern that the usage misses. *Facts* might be method calls, order relations between pairs of calls, or nodes and edges, depending on the usage representation. Intuitively, usages that are very different from a pattern P are more likely to be occurrences of an alternative pattern than violations of P . We observe that a strict classification by the number of missing facts may lead to false negatives (see Section 7.2), because some kinds of API misuses lead to multiple missing facts. For example, if a method call is missing, then so are all order relations between this call and others. Therefore, we propose to use the distance between a pattern and a violation as a ranking factor, such that violations that are farther away from the pattern simply get ranked lower.

We compute the distance between a pattern and a usage AUG via the number of nodes and edges n_m that the usage misses from the pattern. We normalize n_m by the total number of elements n_p of the pattern. Since a missing node always implies that all edges connecting to it are also missing, we take the number of missing edges from/to missing nodes n_e out of the equation. This leads to our *violation-overlap measure* $v_o = (n_m - n_e)/(n_p - n_e)$.

Pattern Support Some detectors rank their findings by the support of the violated patterns (p_s) [LZ05, TX09b, TX09a], i.e., by the number of instances of the pattern discovered during pattern mining. Intuitively, p_s expresses the miner’s confidence in the correctness of the pattern and, consequently, in its violations being problematic. Note that this leaves the relative order of multiple violations of the same pattern unspecified.

Confidence Monperrus *et al.* [MM13] rank their findings by the *confidence*, which combines the pattern support (p_s) and the *number of violations of the pattern* (p_v) into $p_s/(p_s + p_v)$. Intuitively, patterns with more violations are more likely to contain usage

properties that are not strictly required and, hence, their violations are more likely to be false positives. Similar to the pattern support, the confidence depends solely on the pattern and, therefore, leaves the relative order of multiple violations of the same pattern unspecified.

Rareness Some detectors [LZ05, NNP⁺09b] rank their findings by their *rareness*, which combines the pattern support (p_s) and the number of times the violation reoccurs, i.e., the *violation support* (v_s), into $(p_s - v_s)/p_s$. Intuitively, a violation that occurs more often is less likely to be problematic.

Defect Indicator Wasylkowski *et al.* [WZL07] rank findings by a *defect indicator*, which combines the pattern support (p_s), the violation support (v_s), and a *pattern uniqueness factor* (p_u), into $p_u \times p_s / v_s$. To compute p_u they count for every API in the pattern the number of violations involving that API and take the inverse of the largest such number. Intuitively, if an API is involved in more violations, any particular violation involving it is less likely to be problematic.

MuDetect’s Ranking Strategy We observe that all ranking strategies from the literature are composed from a small number of ranking factors, namely pattern support (p_s), the uniqueness factor (p_u), number of pattern violations (p_v), violation support (v_s), and the violation overlap (v_o). Each of these factors captures a characteristic of a violation (or the respective pattern) that makes it more or less likely to be an actual problem.

As candidates for our ranking strategy, we consider the strategies from the literature and all combinations of the individual ranking factors by multiplication. For the latter, we use the factors p_s , p_u , and v_o as is, but invert p_v and v_s , such that smaller values imply lower probability of the violation being problematic. We multiply them, such that, if any of the factors is low, the overall ranking weight is low. Since it is unclear which candidate strategy is most useful, we empirically evaluate them. We explain the respective experiment in Section 12.2.4 and its results in Section 13.1.

11.5. Per-project and Cross-project Settings

As Section 11.2 and 11.3 show, we designed MUDETECT with separate pattern mining and detection phases. This allows us to run MUDETECT in two different settings.

Per-project Setting In the *per-project setting*, we configure MUDETECT to use the AUGs from its target project as the input for both pattern mining and violation detection. This enables a fair comparison to existing detectors that combine mining and detection in a single phase—such as the detectors we evaluated in Chapter 7—and, thus, always mine and detect on the same input.

In this setting, we follow existing work [LZ05, NNP⁺09b, WZL07, WZ11] and define the frequency measure $f(p)$ of a pattern as the number of distinct instances of the pattern.

11. A New Detector

Cross-project Setting In a *cross-project setting*, we configure MUDetect to use the AUGs from its target project as the input for violation detection and AUGs from other projects as input for pattern mining. This allows us to provide additional usage examples for mining (*P8* (Lack of Usage Examples)). For easier differentiation in the text, we call this configuration MUDetectXP.

In this setting, we define the frequency measure $f(p)$ of a pattern p as the number of projects from which at least one instance of the pattern originates. The intuition is that a pattern that occurs in more projects is used by more developers (a generally reusable pattern) and, therefore, more likely to be correct than a pattern that occurs only in a single project (a project-specific pattern), even if it occurs frequently within that project.

12. Evaluation Setup

In this section, we present the setup that we use to assess MUDetect’s ability to detect API misuses. Our main goal is to evaluate MUDetect’s precision and recall, especially compared to existing detectors, to understand whether our mitigation strategies are effective in practice. Additionally, we want to compare the various ranking strategies discussed in Section 11.4 in a practical setting.

12.1. Detectors and Dataset

In Chapter 7 we empirically evaluated and compared the four misuse detectors JADet, GROUMINER, TIKANGA, and DMMC. In this part, we compare MUDetect against these four detectors.

As the ground-truth for the previous experiments, we used MUBENCH, a dataset of 84 API misuses that we collected from general-purpose bug datasets, a developer survey, and confirmed findings of existing misuse detectors (Table 12.1, Row 1). For 64 of these misuses, MUBENCH also contains examples of correct usages, which are derived from the fix of the misuse, when available. Since we designed MUDetect using insights from the study presented in Chapter 7, an evaluation only on MUBENCH may suffer from overfitting. Therefore, we extend the benchmark dataset by misuses from four sources:

1. We add misuses identified in a recent study by Legunsen *et al.* [LHX⁺16]. They applied runtime verification of API specifications to 200 open-source projects and submitted 114 pull requests that fix API misuses identified in this process. From this set, we take the 77 misuses for which the pull request was accepted as of August 8, 2017.
2. We add 5 misuses identified in two previous misuse-detector evaluations [NNP⁺09b, WZ11].

Table 12.1.: MUBENCH: Number of Misuses (#MU) and Number of Misuses with Corresponding Correct Usages (#CU).

	Dataset	#MU	#CU
1	Original MUBENCH	84	64
2	Dataset Extension	107	107
3	Extended MUBENCH	191	171

12. Evaluation Setup

3. We add 5 misuses from a public collection of common misuses of APIs from the Java 8 Standard Library.¹
4. We invest more time trying to compile project versions from our original collection of API misuses that we could not compile when we first created the MUBENCH dataset (see Section 5.2). We manage to compile nine additional project versions with 20 misuses.

Section 8.1 discussed all these extension in detail. In total, we obtain 107 new misuse examples from 30 projects for our experiments (Table 12.1, Row 2).² Following the structure of MUBENCH, we hand-craft examples of correct usage for all newly added misuses. We publish the extended dataset with MUBENCH.

Overall, this gives us a benchmark dataset with 191 API misuses from real-world projects (Table 12.1, Row 3). We use this dataset in the subsequently described experiments. For simplicity, we refer to this extended dataset as MUBENCH throughout this part of the thesis.

12.2. Experimental Setup

To evaluate the precision and recall of MUDETECT, we conduct three experiments, namely Experiment P to measure precision, Experiment RUB to determine a recall upper bound, and Experiment R to measure the actual recall. This is the same per-project experimental setup proposed in Chapter 5. In addition, we design two further experiments: Experiment RNK, to compare the various ranking strategies discussed in Section 11.4, and Experiment XP, to evaluate MUDETECTXP in a cross-project setting, since one reason for the bad performance of detectors may be a lack of usage examples in the per-project setting (see Chapter 7). For the experiments, we set the frequency threshold $\sigma = 10$ for MUDETECT and $\sigma = 5$ for MUDETECTXP. For the other detectors, we use the best configurations as reported in the respective publications.

We execute the experiments using MUBENCHPIPE, the public automated benchmarking pipeline that we built on top of MUBENCH (see Chapter 6). MUBENCHPIPE facilitates preparing the target projects from MUBENCH, executing the detectors on them, and collecting result statistics about the detectors' performance we manually reviewed the detectors' findings. In all our experiments, two authors first independently reviewed each detector finding and then discussed any disagreements until a consensus was reached about whether the finding correctly identifies a misuse. We report Cohen's Kappa score as a measure of the reviewers' initial agreement. We now introduce these five experiments in detail.

¹<https://github.com/xpinjection/java8-misuses> (checked on Mar 20, 2018)

²JADET and TIKANGA initially crashed on most of these new projects, because both detectors use the outdated Bytecode toolkit ASM 3.1. To fix this, we migrated them to the most recent version ASM 6.0. To ensure that this change did not hamper with their capabilities, we repeated the experiments from Chapter 7 and successfully verified that the detectors still produce the exact same results.

12.2.1. Experiment P

The goal of Experiment P is to measure the detectors' *precision*. We run the detectors on all projects from MUBENCH, letting them *mine patterns and detect violations* on a per-project basis. Since some detectors report several hundreds of findings, reviewing all findings of all detectors on all projects is practically infeasible. Therefore, we sample ten projects and review the top-20 findings per detector on each of them, as determined by the detectors' individual ranking strategies. In this sample, we include the five projects that we used in the precision experiment in Chapter 5. In addition, we choose another five of the new projects we added to MUBENCH. To this end, we compute the average normalized number of findings (ANNF) across detectors for each project. The NNF for a given detector for a given project, is the number of findings the detector has on that project divided by the maximum number of findings the detector has on any project. Then, we select two projects with the highest ANNF, two projects with the lowest ANNF, and one random project from the mid range. For fairness, we exclude projects where one of the detectors failed or where two or more detectors did not report any findings. We cannot exclude all projects where one of the detectors did not report any findings, because this left us with less than five projects to chose from.

12.2.2. Experiment RUB

The goal of Experiment RUB is to assess the detectors' *recall upper bound*, given perfect training data. This helps us separate conceptual limitations from the effect of insufficient training data. Since we need to provide a correct usage as input training data, we limit this experiment to the 171 misuses in MUBENCH that have corresponding correct usages (see Table 12.1). We run the detectors once for each of these misuses, providing them with enough copies of the corresponding correct usage for pattern mining. This ensures that detectors always find sufficient evidence to mine the pattern required to identify the misuse. We review all potential hits, i.e., all detector findings in the same method as the known misuse.

12.2.3. Experiment R

The goal of Experiment R is to measure the detectors' *recall*. We run the detectors on all projects of MUBENCH, letting them *mine patterns and detect violations* on a per-project basis. Then, we review all potential hits, i.e., all findings in the same method as a known misuse. As the ground truth, we use all 191 known misuses from MUBENCH, plus any previously unknown true positives identified by the detectors in Experiment P. This gives us the recall of the detectors with respect to a large number of misuses and, at the same time, crosschecks which of the detectors' findings are also identified by other detectors.

12.2.4. Experiment RNK

The goal of Experiment RNK is to find *the best ranking strategy* for MUDetect among the candidate strategies discussed in Section 11.4. Ideally, we would repeat both Experiment P and Experiment R for all 34 candidate ranking strategies to determine the best strategy. However, repeating Experiment P would require us to review up to 20 findings for each of the ten target projects per candidate strategy—a total of 6,800 findings—which is practically infeasible. Therefore, we only repeat Experiment R for each of the candidate ranking strategies. This gives us both the recall of the detector, as well as the ranks of all confirmed hits, i.e., findings that identify a known misuse from MUBENCH through reviewing a relatively small number of findings. We use the *number of hits*, the *average rank of all hits*, and the *number of hits in the top-20 findings* as quality measures for the ranking strategies.

12.2.5. Experiment XP

The goal of Experiment XP is to measure MUDetectXP’s *precision* and *recall*. To measure precision, we run MUDetectXP on the ten sample projects from Experiment P review its top-20 findings. To measure recall, we run MUDetectXP on all projects in MUBENCH and review all its potential hits for all known misuses, as in Experiment R. For each target project, we provide the detector with training projects for all APIs with known misuses in the target project. To ensure that the training projects contain examples of the APIs with known misuses in MUBENCH, we collect client projects of the respective APIs using the code-repository mining platform BOA [DNRN13] (full 2015 GitHub dataset). For each API, we query BOA for projects that either declare a field, variable, or parameter, or call a static method of the respective API type. We publish the query template and the result lists.³ From each list, we take the first 50 projects⁴ and randomly sample up to 20 usage examples of the respective API from each project. This gives us a diverse cross-project sample of up to 1,000 usage examples per API.

³Artifact Page: MUDetect: The Next Step in Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/>)

⁴Some projects that the latest BOA dataset (2015) refers to have meanwhile been deleted or renamed. We exclude projects that are unavailable as of February 2018.

13. Evaluation Results

In this chapter, we present the results of our experiments and compare MUDetect’s performance with the detectors JADet, GROUMINER, TIKANGA, and DMMC. All experiments ran on a *MacBook Pro* with an *Intel Xeon @ 3.00GHz* and *32GB of RAM*. The full results are available on.¹

13.1. Experiment RNK: Ranking Violations

We first run Experiment RNK to determine the best ranking strategy for MUDetect. Table 13.1 shows the number of hits in the top-20 findings (@20), the number of hits (#H), and the average hit rank (AHR) for the best and worst ranking strategies we evaluated. We order the ranking strategies by these three metrics in the mentioned order. The full list is available on our artifact page.²

The results show that the ranking has a huge impact on how many misuses MUDetect finds and how well it ranks them in the top findings. We observe that the pattern support (p_s) appears in all of the top-16 and in none of the bottom-10 ranking strategies. Contrarily, the pattern uniqueness (p_u) appears in 11 of the bottom-15 strategies and in none of the top-10. The violation-overlap measure (v_o), the violation support (v_s), and the pattern violations (p_v) appear in different combinations in ranking strategies throughout the field. While this clearly shows that the pattern support is the most important ranking factor, the strategy consisting of only this factor is only the 9th-best strategy. This suggests that detectors should consider other factors as well. The best strategy combines the pattern support (p_s), the support of the violations (v_s), and the violation-overlap measure (v_o) into $p_s/v_s \times v_o$. We use this strategy for both MUDetect and MUDetectXP in all remaining experiments.

13.2. Experiment P: Precision

We measure the detectors’ precision in their top-20 findings in Experiment P Table 13.2 summarizes the results. MUDetect reports 146 violations in the top-20 findings in the ten projects. Among these findings, we find 32 true positives, 19 of which were previously unknown. This results in a precision of 21.9%, which exceeds the precision of the other detectors more than two-fold.

¹Artifact Page: MUDetect: The Next Step in Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/>)

²Artifact Page: MUDetect: The Next Step in Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/>)

13. Evaluation Results

Table 13.1.: Results of Experiment RNK: Alternative Ranking Strategies Ordered by The Number of Hits in the Top-20 (@20), The Total Number of Hits (#H), and The Average Hit Rank (AHR).

#	Strategy	@20	#H	AHR
1.	$p_s/v_s \times v_o$	19	34	91.6
2.	p_s/v_s	17	34	91.8
3.	$p_s/v_s \times v_o \times p_v$	16	34	90.1
4.	Rareness $((p_s - v_s)/p_s)$	16	33	94.3
...				
9.	p_s	14	34	305.5
...				
33.	p_u/v_s	2	18	53.2
34.	v_o	1	26	1187.4

Table 13.2.: Results of Experiment P: Precision of the Detectors in Their Top-20 Findings.

Detector	Confirmed Misuses	Precision	Kappa Score
JADET	8	8.8%	0.64
GROUMINER	4	2.6%	0.49
TIKANGA	7	8.2%	0.52
DMMC	12	7.5%	0.72
MUDETECT	32	21.9%	0.90

While clearly outperforming the precision of the other detectors, considered on its own, MUDETECT’s precision is still low. Hence, we analyze the root cause for each false positive it reports. We label these root causes (**FPN**). A detailed discussion of possible mitigation strategies follows in Section 13.6.

Root Cause FP1: Uncommon-but-correct Usages. 84 (73.7%) of the false positives are uncommon-but-correct usages. MUDETECT does not mine patterns for these usages, since they occur only infrequently in the target projects. It reports them as violations, because they deviate from an alternative pattern that MUDETECT mined.

In eleven cases, for example, the usage checks for the presence of sufficiently many elements in an `Iterator` through either `size()` or `isEmpty()` on the underlying collection. Both are valid alternatives, but occur themselves too infrequent for MUDETECT to learn a pattern. Since it, however, learns a pattern for checking `hasNext()` to ensure sufficiently many elements, it reports missing checks of `hasNext()` in all cases.

In another seven cases, the usage calls `size()` on a collection for another purpose than controlling access to the collection’s elements, e.g., to write the size into a log message. In these cases, MUDETECT reports that the call to `size()` should control an access to

the collection. This shows two problems:

1. The heuristic for identifying pure methods by looking for methods with the prefix `get` is insufficient, e.g., it fails to tag `size()` as pure.
2. The detection algorithm does not recognize that retrieving the size of a collection is perfectly fine for purposes other than guarding access to that collection, i.e., it does not consider that an action requires that its preconditions are ensured, but not vice versa.

These problems also cause nine cases where a loop calls `Iterator.hasNext()` again after calling `next()`, to check whether there will be a subsequent iteration. MUDetect reports a missing call to `next()` after this second call to `hasNext()`. Here, the heuristic for pure methods fails to recognize that `hasNext()` is pure and the detection algorithm does not recognize that `hasNext()` does not need to guard a call to `next()`.

Root Cause FP2: Imprecise Analysis. 18 (15.8%) of the false positives are due to limitations of MUDetect’s static analysis. In seven cases, the usage elements that MUDetect reports missing happen in a transitively called method, which our intra-procedural analysis cannot consider. In another four cases, the usage ensures a precondition through an `assert` statement, which our analysis does not recognize to control the execution of subsequent actions.

Root Cause FP3: Dependent Object States. Five (4.4%) of the false positives are due to implicit dependencies. In two cases, for example, the code ensures that two collections have the same size and then iterates over one, while checking the size of the other. In another case, the usage iterates over the key set of a `Map`, while checking the `Map`’s size. In the two remaining cases, the usages loop over a `List` in a `for` loop with an index i , while accessing the list with an index j that is guaranteed to be smaller than i . All these usages are safe, but MUDetect cannot capture the implicit dependencies between object states and values.

Root Cause FP4: Non-optimal Mapping. Another five (4.4%) of the false positive are due to MUDetect’s detection algorithm choosing a non-optimal mapping between a pattern and a target. In these cases, the target is actually an instance of the pattern, but MUDetect fails to recognize this. This shows that the greedy extension strategy in the detection, which we chose to keep the detection scalable while detecting even single missing edges, comes at the cost of precision.

Root Cause FP5: Cross-method Usages. One false positive is a cross-method usage, which our filtering misses, because the respective object is initialized as a local variable and only later assigned to a field.

Table 13.3.: Results of Experiment RUB: Recall Upper Bound of the Detectors with Respect to the Misuses with Corresponding Correct Usages from MUBENCH.

Detector	Hits	Recall Upper Bound	Kappa Score
JADET	29	16.9%	0.79
GROUMINER	88	51.2%	0.85
TIKANGA	15	8.8%	0.73
DMMC	28	16.3%	0.88
MUDETECT	124	72.5%	0.89

Root Cause FP6: Alternative Patterns. The last false positive is a case where a violation is an instance of a combination of two alternative patterns, which our alternative-pattern filtering cannot detect.

13.3. Experiment RUB: Recall Upper Bound of the Detectors

Table 13.3 summarizes the results of measuring the upper bound to the detector’s recall. MUDETECT identifies 124 of the 171 misuses used in this experiment (72.5%). This upper bound of recall clearly outranks that of the other detectors by 20.3% for GROUMINER and by over 54% for each of the other three detectors. This shows that (1) AUGs are able to better capture the difference between correct usages and misuses, and (2) our detection algorithm succeeds in identifying these differences.

MUDETECT identifies 39 misuses that all other detectors miss. In turn, MUDETECT misses ten misuses that at least one of the other detectors finds. Eight of these cases we miss due to one of the heuristics we introduced to improve precision, such as the filtering of cross-method usages. There are 38 more misuses that all detectors miss.

Next, we analyze the root causes for each of these 47 false negatives of MUDETECT. We label these root causes (**FN**). A detailed discussion of possible mitigation strategies follows in Section 13.6.

Root Cause FN1: Imprecise Representation. In 15 cases, an illegal parameter value (constant or literal) is passed to a method as a parameter. MUDETECT cannot detect this, because AUGs do not capture concrete values. In one other case, a condition is checked with an `if`, but should be checked repeatedly using a loop.

Root Cause FN2: Filtering of Self Usages. Eight cases are due to our removal of self usages (five cases) and cross-method usages (three cases) to counteract their potential to cause false positives.

Root Cause FN3: Imprecise Pattern Matching. In Seven cases, MUDETECT cannot match the respective pattern and the target usages, because they contain only a single,

Table 13.4.: Results of Experiment R: Recall of the Detectors with Respect to The Known Misuses in MUBENCH and the Detectors' Own True-Positive Findings from Experiment P.

Detector	Hits	Recall	Kappa Score
JADET	15	6.7%	0.64
GROUMINER	7	3.1%	1.00
TIKANGA	17	7.6%	0.69
DMMC	24	10.7%	0.91
MUDETECT	47	20.9%	1.00

distinct call each. Our detection algorithm, by design, does not match AUGs that have nothing in common.

Root Cause FN4: Inability to Identify Redundant Usage Elements. Seven cases are misuses where the usage has a redundant element that should be removed. Since all detectors in our experiments are designed to detect missing elements, none can detect these misuses.

Root Cause FN5: Filtering of Pure-Method Calls. In six cases, our mining excludes elements from the pattern when heuristically determining semantically irrelevant nodes (calls to getter methods in all cases). The remaining pattern can no longer be matched to the target usage, because they have nothing in common.

Root Cause FN6: Imprecise Analysis. In one case, our analysis misses data-flow relations, because it assumes single static assignment. In the correct usage `if (v == null) { v = ... } v.m()`, which ensures the initialization of `v` before its use, it regards `v` before and after the assignment as two different objects. Therefore, our pattern mining sees no data flow between the `null` check and the subsequent usage and excludes the check from the pattern. Consequently, MUDETECT cannot identify the missing check in the respective misuse.

Root Cause FN7: Operator Abstraction. In one case, the misuse is an accidentally inverted condition, which we miss because we abstract from concrete operators in representing conditions.

13.4. Experiment R: Recall of the Detectors

Overall, the detectors identified 34 unique previously unknown misuses in Experiment P. We add these misuses to the 191 misuses from MUBENCH for Experiment R. Table 13.4 summarizes the results of measuring the detectors' recall. MUDETECT identifies 47 of the 225 misuses. This results in a recall of 20.9%, which exceeds the recall of the other

13. Evaluation Results

detectors almost two-fold. MUDETECT correctly identifies 13 misuses that none of the other detectors identifies, eleven of which it already identified in Experiment P. The other six previously unknown misuses from Experiment P are also correctly identified by at least one other detector.

In turn, MUDETECT misses 13 misuses that one of the other detectors finds. Six of these are misuses that only DMMC identifies, because the projects contains too few usage examples for any of the other detectors to mine a respective pattern. DMMC’s probabilistic approach is able to identify misuses with very little evidence. In the extreme case (three of the six cases), DMMC finds exactly two usage examples for the respective API: a correct usage and the misuse. Consequently, $p_s = 1$ and $p_v = 1$ and, therefore, confidence = 0.5, which is exactly DMMC’s lower bound on confidence for reporting a misuse. Since MUDETECT requires a pattern support of at least 10, it cannot find these misuses. Another five of these misuses are identified by JADET or TIKANGA or both. In all these cases, the target method contains multiple equal misuses of the same API. JADET and TIKANGA report a single finding identifying the misuse, but since we cannot determine which particular misuse instance it refers to, we conservatively count it as a hit for all instances. MUDETECT, on the other hand, reports its findings at line level, which is why we only count hits when the reported violation line matches the known misuse line, resulting in only 1 hit being counted. For the last two of these misuses, MUDETECT misses the pattern due to the greedy extension strategy that we chose to keep the mining scalable.

While MUDETECT has higher recall than the other detectors, its recall is still low in absolute terms. We find that MUDETECT has on average 227.6 usages examples (median = 105) for APIs whose misuses it identifies, but only 38.6 examples (median = 11) for those it misses. There is a moderate correlation (Pearson’s $r = 0.52$) between the number of examples and detecting a misuse. This supports the hypothesis that the target projects contain too few usage examples for some APIs. Experiment XP investigates this further.

13.5. Experiment XP: Cross-project Misuse Detection

The 225 misuses that we consider for Experiment R are usages of 59 APIs. For five of these, we find no projects with respective usages on GitHub and for another thirteen, we find less than 50 projects (1, 1, 2, 4, 8, 12, 14, 20, 21, 26, 39, 42, and 44). For the remaining 41 APIs, we find 50 or more projects. Our cross-project sampling strategy (see Section 12.2.5) collects on average 239.3 usage examples per API (median = 172), compared to the average of 78.5 examples (median = 25) in the per-project setting.

Table 13.5 shows the precision and recall of MUDETECTXP compared to the performance of the other detectors in Experiment P and Experiment R. MUDETECTXP reports 91 violations in the top-20 findings on the 10 projects from Experiment P. Among these findings, we find 31 true positives, three of which were previously unknown. This results in a precision of 34.1%, which outranks MUDETECT by 12.2%. Moreover, MUDETECTXP identifies 95 of the 225 misuses. This results in a recall of 42.2%, which

Table 13.5.: Results of Experiment XP: Precision and Recall of MUDETECTXP in the Cross-project Setting in Comparison to the Precision and Recall of the Detectors in the Per-project Setting.

Detector	Precision			Recall		
	Confirmed Misuses	Precision	Kappa Score	Hits	Recall	Kappa Score
JADET	8	8.8%	0.64	15	6.7%	0.64
GROUMINER	4	2.6%	0.49	7	3.1%	1.00
TIKANGA	7	8.2%	0.52	17	7.6%	0.69
DMMC	12	7.5%	0.72	24	10.7%	0.91
MUDETECT	32	21.9%	0.90	47	20.9%	1.00
MUDETECTXP	31	34.1%	0.88	95	42.2%	0.93

improves on MUDETECT more than two fold. This clearly shows that additional usage examples are crucial to improving the performance of API-misuse detectors.

MUDETECTXP identifies 65 misuses that MUDETECT misses. For these misuses, MUDETECT has on average only 94.6 usage examples (median = 16), while MUDETECTXP has on average 258.4 examples (median = 216). This suggests that detectors should search for additional usage examples, if the target project itself contains too few. MUDETECT, in turn, identifies 17 misuses that MUDETECTXP misses. Ten of these are usages of APIs declared in the respective target project. Interestingly, the problem is not a lack of examples, as MUDETECTXP has on average 239.3 (median = 172). A possible explanation is that APIs are used differently in the declaring project than in client projects. This suggests that detectors should consider, but distinguish both sources of usage examples. Overall, we observe only a weak correlation (Pearson’s $r = 0.17$) between the number of examples that MUDETECTXP has for pattern mining and the detection of a respective misuse. This suggests that, while the number of available examples is an important factor for the performance of a misuse detector, there are other factors at play as well. We hypothesize that the quality of the examples or their representativeness of the possible usages of an API might be such factors. Future work should investigate this.

13.6. Discussion

Our results show that, with MUDETECT, we successfully took a next step in static API-misuse detection. With a precision of 34% and a recall of 42%, we are closer than ever to practical applicability of a detector. Moreover, our findings suggest that there is still room for improvement. We subsequently discuss the effects of the mitigation strategies we implemented for the problems that motivated our work (see Chapter 10) and the

13. Evaluation Results

remaining problems that we identified in our experiments, and sketch ideas for future work.

The results of Experiment RNK and Experiment P show that our ranking strategy successfully orders true positives in the top findings (*P7* (Poor Ranking)). The results of Experiment XP show that mining patterns from other projects significantly improves both precision and recall (*P8* (Lack of Usage Examples)). Future work should investigate techniques for the *retrieval of high-quality usage examples* to train detectors.

As Experiment RUB shows, we successfully mitigated *P1* (Imprecise Representation) with the design of AUGs. To further reduce the remaining false negatives caused by *FN1* (Imprecise Representation), we plan to investigate the effect of encoding constant names as data-node labels. We abstract such names to avoid mismatches when variables and literals are used interchangeably. However, constants are less likely used interchangeably with variables or literals. Thus, we might be able to capture such cases, without introducing many false positives.

While MUDetect identifies significantly more misuses than the other detectors, this comes at a price. The greedy extension strategy that we chose to keep both our mining and detection scalable causes false negatives in Experiment RUB (*FP4* (Non-optimal Mapping)) and Experiment R (missing pattern). Future work should investigate alternative mining approaches to avoid these problems.

For the cause of another false negative, *FN7* (Operator Abstraction), we do not believe that misuse detectors can mitigate it efficiently. While AUGs could be adapted to capture concrete operators, this would most likely cause many false positives. Given this trade-off, we believe our operator abstraction is the better design decision.

MUDetect identifies wrong call order and abstracts over static types to address *P2* (Imprecise Pattern Matching). However, there are still instances of *FN3* (Imprecise Pattern Matching). They are patterns containing only a single method call. A possible solution could be to identify usages based on the presence of respective data nodes, as opposed to on the presence of at least one call on the API. However, then we would compare target usages to patterns with which they do not share even a single call, which would always result in violations. This would produce many false positives, since the target is likely to implement an alternative usage pattern that the detector is unaware of. Given this trade-off, we believe that not matching these cases is the better design choice.

We failed to mitigate *P3* (Uncommon-but-correct Usage), as *FP1* (Uncommon-but-correct Usages) remains the most-prevalent cause for false positives. While our pure-methods heuristic filters some false positives, it is obviously insufficient to address the problem. To reduce the impact further, we might choose a lower frequency threshold to also learn patterns for the alternative usages that are mistaken for violations. However, this would merely shift the problem to other APIs where one alternative usage occurs slightly more frequently and another less frequently than the threshold. A solution might be a *probabilistic model of API usage* that considers the likelihood of different usages and reports no violation if one usage is only slightly more likely than another or if an API's usages generally vary a lot.

The false negatives caused by *FN4* (Inability to Identify Redundant Usage Elements)

cannot be detected by misuse detectors that search for missing elements. DROIDASIST [NPVN16] uses a *probabilistic approach* that might find superfluous method call, but the approach has never been evaluated.

By excluding self usages and usages on fields from both mining and detection, we reduced the impact of *P5* (Self- and Cross-method Usages). However, this strategy and the removal of pure method calls as a means to reduce the impact of *P3* (Uncommon-but-correct Usage) immediately cause false positives as identified by the two root causes *FN2* (Filtering of Self Usages) and *FN5* (Filtering of Pure-Method Calls). While Experiment P suggests that these measures are effective, apparently, we trade precision for recall. By *capturing inter-procedural usages*, we might make filtering them unnecessary and enable us to identify misuses in them. The CHRONICLER [RGJ07b] detector mines usages from an inter-procedural call graph, which might mitigate the problem. However, it is unclear how to adapt this approach from considering only method calls to all usage elements we encode in AUGs. Furthermore, such an approach duplicates evidence, if methods are called multiple times, which might bias the mining. Future work should investigate the possibilities and effects of such an approach. On a related note, some false negatives and false positives are caused by *FN6* (Imprecise Analysis) and *FP2* (Imprecise Analysis), i.e., by imprecisions of our static analysis. We observe that a large fraction of these cases are inter-procedural usages, where API objects are passed to or returned from methods. The misuse detector PR-MINER [LZ05] analyzes transitive calls to check whether missing method calls appear there and filters respective violations. Such a strategy would reduce the impact of the problem, but we need to also consider callers to be able to generally *capture inter-procedural usages*. Although such cases also appeared in the detector findings we analyzed in Chapter 7, we did not specifically address them with MUDetect. A *more sophisticated, possibly inter-procedural, static analysis* might help reduce the impact of these problems.

The root cause *FP3* (Dependent Object States) also appeared in Chapter 7, as the least prevalent cause for false positives. Since it is very difficult, if not impossible, to capture inter-dependencies of object states using static analysis, we did not address this problem with MUDetect. However, again, a *more sophisticated static analysis* might help reduce the impact of these problems.

13.7. Reviewer Agreement

To mitigate subjectivity in the reviews of detectors' findings in our experiments, the author of this thesis and one of his colleagues separately reviewed all findings. We report Cohen's Kappa score as a metric for the initial reviewer agreement. For our experiments, we observe noticeable differences in the Kappa scores between detectors. We investigate possible causes.

13.7.1. Agreement in Experiment P

Table 13.6 shows the detailed agreement statistics for our reviews in Experiment P, including MUDetectXP. We observe that the agreement for our detector is considerably

13. Evaluation Results

Table 13.6.: Reviewer Agreement in Experiment P, Including MUDETECTXP. Reviewers classified each finding as a true positive (TP) or false positive (FP)

Detector	Agreements			Disagreements			Cohen's Kappa				
	TP	FP	Total	TP/FP	FP/TP	Total	p_0	p_{TP}	p_{FP}	p_e	Score
JADET	5	82	87	3	2	5	0.95	0.01	0.84	0.85	0.64
GROUMINER	2	150	152	2	2	4	0.97	0.00	0.95	0.95	0.49
TIKANGA	6	70	76	0	9	9	0.89	0.01	0.77	0.78	0.52
DMMC	10	144	154	2	5	7	0.96	0.01	0.84	0.85	0.72
MUDETECT	29	112	141	2	3	5	0.97	0.05	0.62	0.66	0.90
MUDETECTXP	30	56	86	0	5	5	0.95	0.13	0.41	0.54	0.88

higher than for the other detectors. The agreement is lowest for GROUMINER and TIKANGA and on a middle ground for JADET and DMMC.

The relative observed agreement between reviewers (p_0) is comparable across all detectors (≈ 0.96), except for TIKANGA, where it is a bit lower (0.89). Among the nine disagreements for TIKANGA we find two duplicated findings, one of which TIKANGA reported twice and the other even three times. The reviewers consistently disagreed on these duplicates. We were no disagreements on duplicated findings for any of the other detectors. This likely explains the difference in the relative observed agreement.

The expected probability for both reviewers to classify a findings as a true positive (p_{TP}) is a bit higher for MUDETECT than for the other detectors, and another bit higher for MUDETECTXP. The expected probability for both reviewers to classify a findings as a false positive (p_{FP}), on the other hand, is considerably lower for our detector, but also varies between the other detectors. We observe that p_{FP} decreases with the precision of the detectors, except for TIKANGA, where it is again lower due to our consistent disagreements. Consequently, the hypothetical probability of chance agreement ($p_e (= p_{TP} + p_{FP})$) also decreases with the precision of the detector, again with the exception of TIKANGA. This phenomenon is to be expected: For a detector with a very low precision, most decisions are for false positives and, thus, the probability of chance agreement is high. The closer the precision gets to 50%, the more balanced become the decisions and the lower is the probability of chance agreement.

Since the relative observed agreement between reviewers (p_0) is high for all detectors, the difference in the hypothetical probability of chance agreement (p_e) has a huge impact on the Kappa score. This explains the high variance in the scores between detectors in Experiment P.

13.7.2. Agreement in Experiment RUB

Table 13.7 shows the detailed agreement statistics for our reviews in Experiment RUB. We observe that the agreement is highest for MUDETECT and DMMC, and decreasing a bit for each of the other detectors in the order GROUMINER, JADET, and TIKANGA. The relative observed agreement between reviewers (p_0) decreases slightly between detectors in the same order. We note that the absolute number of decisions is considerably

Table 13.7.: Reviewer Agreement in Experiment RUB. Reviewers decided for each known misuse whether a detector identifies it (Hit) or not (Miss).

Detector	Agreements			Disagreements			Cohen's Kappa				
	Hit	Miss	Total	Hit/Miss	Miss/Hit	Total	p_0	p_{Hit}	p_{Miss}	p_e	Score
JADET	26	12	38	3	1	4	0.90	0.44	0.11	0.55	0.79
GROUMINER	84	41	125	3	6	9	0.93	0.44	0.12	0.55	0.85
TIKANGA	14	19	33	1	4	5	0.87	0.19	0.32	0.51	0.73
DMMC	28	69	97	4	1	5	0.95	0.09	0.49	0.58	0.88
MUDETECT	118	26	144	1	4	5	0.97	0.65	0.04	0.69	0.89

Table 13.8.: Reviewer Agreement in Experiment R, Including MUDETECTXP. Reviewers decided of reach known misuse whether a detector identifies it (Hit) or not (Miss).

Detector	Agreements			Disagreements			Cohen's Kappa				
	Hit	Miss	Total	Hit/Miss	Miss/Hit	Total	p_0	p_{Hit}	p_{Miss}	p_e	Score
JADET	14	1	15	1	0	1	0.94	0.82	0.01	0.83	0.64
GROUMINER	7	11	18	0	0	0	1.00	0.15	0.37	0.52	1.00
TIKANGA	15	3	18	2	0	2	0.90	0.64	0.04	0.68	0.69
DMMC	24	40	64	0	3	3	0.96	0.14	0.38	0.53	0.91
MUDETECT	44	26	70	0	0	0	1.00	0.40	0.14	0.53	1.00
MUDETECTXP	92	55	147	2	3	5	0.97	0.39	0.14	0.53	0.93

smaller for JADET and TIKANGA than for the other detectors, yet, the absolute number of disagreements is not. Many of the disagreements on these detectors' findings are due to different interpretations of their representation. Conversely, we note that the absolute number of disagreements is twice as high for GROUMINER than for MUDETECT and DMMC. We analyzed the disagreements for GROUMINER, but could not identify any systematic reason for them. We speculate that the simplicity of DMMC usage representation and the reviewer's familiarity with the MUDETECT's representation led to a smaller number of disagreements. We previously discussed the problem of interpreting detector findings in Section 7.5.

13.7.3. Agreement in Experiment R

Table 13.8 shows the detailed agreement statistics for our review in Experiment R, including MUDETECTXP. We observe that the agreement is high for GROUMINER, DMMC, MUDETECT, and MUDETECTXP, but considerably lower for JADET and TIKANGA. For the latter detectors, the absolute number of disagreements is low, as is the absolute number of decisions the reviewers had to make. Due to this small number of decisions, the hypothetical probability of chance agreement (p_e) is very sensitive to single disagreements, as the comparison to the scores for GROUMINER shows. Otherwise, the relative observed agreement between reviewers (p_0) is high across detectors. The

13. Evaluation Results

hypothetical probability of chance agreement (p_e) for GROUMINER, DMMC, MUDTECTXP, and MUDTECTXP is almost equal. This is reflected by the high Kappa scores for these detectors.

14. Threats to Validity

Overfitting We designed and fine-tuned MUDetect based on observations from running detectors on MUBENCH (see Chapter 7). We evaluated MUDetect on the same benchmark, which bears the danger of overfitting. To mitigate this threat, we extend the benchmark to more than twice its original size and evaluate all detectors on this extended benchmark. We publish this extension and our implementation of MUDetect.¹

Internal Validity We did not fine-tune the other detectors, but used the best configurations reported in the respective publications. Therefore, it is possible that they do not show their optimal performance in our experiments.

We reviewed the detectors’ findings ourselves. The detector producing a finding was known, because each detector has a distinct representation of API usages and violations that we could not blind.

We evaluated only our own detector, MUDetectXP, in the cross-project setting. We did not try other detectors in this setting, because they cannot use separate datasets for pattern mining and violation detection. Hence, evaluating them in the same cross-project setting as MUDetectXP, would have required us to modify their implementations, which might hamper with their capabilities. We published the list of example projects we used to train MUDetect in the cross-project experiment² and encourage others to assess the performance of their approach in this setting.

Providing MUDetectXP with only example usages for the APIs with known misuses in MUBENCH potentially biases the results with respect to precision, because it reduces the overall number of patterns and, consequently, might reduce the number of reported violations.

External Validity The study is subject to the limitations of MUBENCH, as discussed in Section 5.6 (limitations of the benchmarking dataset) and Section 6.4 (limitations of the benchmarking pipeline). We extended the MUBENCH dataset to more than twice its original size, to reduce the impact of these limitations.

¹Artifact Page: MUDetect: The Next Step in Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/>)

²Artifact Page: MUDetect: The Next Step in Static API-Misuse Detection (<http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/>)

15. Related Work

API-Misuse Detectors Helping developers identify API misuses has received much attention. Chapter 4 presents a detailed survey of existing approaches in this field. In this part of this thesis, we presented a comparison of our detector, MUDETECT (and MUDETECTXP), to the other detectors that target Java. We exclude the misuse detectors that target other programming languages [LZ05, RGJ07b, Lin07, RGJ07a, AX09] here.

The closest work to ours is Nguyen *et al.*'s [NNP⁺09b], which proposed a graph-based representation of API usages (GROUMs) for their misuse detector GROUMINER. The good conceptual coverage of detectors using graph-based usage representations in Chapter 4 and the relatively high recall of GROUMINER in our experiments from Chapter 7 led us design our graph presentation, AUG. AUGs, in contrast to GROUMs, are directed acyclic multigraphs that capture method calls, field accesses, `null` checks, and data entities as nodes and control/data dependence among them as labeled edges. Control structures are encoded by control edges between the nodes representing the conditions and the nodes representing the controlled actions. This allows to give more precise information about misuses. For example, while GROUMINER can detect a missing `if`, MUDETECT can also tell what should be checked in the `if` condition. Additionally, AUGs encode exceptional, synchronized, and iterative control flow, and distinguish receivers from parameters, to better differentiate between correct usages and misuses. Encoding more information in edges also makes our pattern mining more scalable.

MUDETECT is the only approach that handles all API-usage elements targeted by all of the detectors we identified in our survey in Chapter 4 combined and, thereby, more usage elements than any one of them individually. It conceptually covers all violations covered by any of the detectors, with the exceptions of redundant method calls, which PJAG12 [PJAG12], PG12 [PG12], and DROIDASSIST [NPVN15] may find. In all three cases, this ability comes from patterns modeling object states, using method calls to signal state transitions. It is unclear whether and how these models can be extended to cover further usage elements, such as conditions, exception handling, and iteration. Future work should investigate this, to further increase the coverage of misuse detectors.

We directly motivate the work in this part of the thesis from the problems of the four detectors JADET [WZL07], GROUMINER [NNP⁺09b], DMMC [MBM10], and TIKANGA [WZ11] as identified in our empirical study in Chapter 7. Our direct empirical comparison to these detectors in Chapter 13 shows that MUDETECT successfully mitigates many of the root causes of the respective problems. We cannot empirically compare MUDETECT to the static misuse detectors ALATTIN [TX09b] and CAR-MINER [TX09a], because their implementations are unavailable (see Section 5.1), and DROIDASSIST [NPVN15], because it is implemented for Dalvik Bytecode, while MUBENCH contains general Java projects. We also cannot empirically compare MU-

15. Related Work

DETECT to the dynamic detectors, because MUBENCH does not support their execution. However, in our experiments we included 77 misuses identified by a dynamic specification verifier (see Section 12.1). Our results show that MUDETECT can detect many of these misuses statically.

Mining Patterns from Project-external Usage Examples Gruska *et al.* [GWZ10] evaluated JADET in a multi-project setting, where it simultaneously mined patterns and detected violations in a combined set of all usage from 6,000 projects. This setting is different from the cross-project setting of Experiment XP, where MUDETECTXP mines patterns in examples from a large number of projects to detect misuses in another project. While MUDETECTXP mines patterns with cross-project support, i.e., patterns that re-occur in a certain number of projects, JADET mines any pattern with high support, even with in a single project. Furthermore, while [GWZ10] measures the precision of JADET in the multi-project setting, we measure both the precision and recall of MUDETECTXP in the cross-project setting. This allows us to show—in direct comparison to the per-project setting we use in Experiment P and Experiment R—that learning from cross-project data significantly improves both precision and recall of our detector. To the best of our knowledge, such a comparison has not been presented before.

CAR-MINER [TX09b] and ALATTIN [TX09a] mine target-specific patterns from examples retrieved via a code-search engine. This presents an alternative to the cross-project mining we do for MUDETECTXP. Future work should compare the performance of both approaches.

Part IV.

Conclusion and Outlook

16. Conclusion

In this chapter, we present a brief overview over the findings from this thesis. We start with a review of the results and contributions of this thesis and follow with a closing discussion.

16.1. Summary of Results

This thesis contributes to the area of API-misuse detection. We provide a holistic view on the problem space of API misuse and the state of the art in misuse detection. We provide a conceptual framework, MUC, which we use to qualitatively compare existing misuse detectors, and an automated benchmark, MUBENCH, which we use to empirically compare them. Based on insights from this empirical comparison, we systematically design a new detector, MUDetect, which advances the state of the art. Finally, we present evidence for possible directions towards further improvement, as a stepping stone for future work.

Prevalence of API Misuse In-the-wild From the findings of multiple studies on API-usage directives we learn (1) that up to two thirds of all API-usage elements come with usage directives that, if violated, lead to API misuse, (2) that developer are often unaware of these directives, and (3) that more than half of these directives are incorrectly documented. This shows the immense potential for API misuse in today’s software products. Through analyzing bug reports we show that indeed almost 10% of all bugs are misuses and that misuses often have severe consequences, such as application crashes or data loss. Through a developer survey and a study on STACKOVERFLOW we gain initial evidence that misuses may considerably slow down the development process. And through an analysis of API usages in Open Source projects, we confirm the prevalence of API misuses that cause security vulnerabilities.

Classification of API Misuses From the examples of API misuse we identified, we create the API MISUSE CLASSIFICATION (MUC), as both a taxonomy of API misuses and a framework to assess the conceptual capabilities of API-misuse detectors. As a first result, MUC shows that certain types of API misuses—namely missing value or state conditions, missing `null` checks, and missing method calls—are much more prevalent than others.

Survey of State-of-the-art Misuse Detection We present a systematic literature review of existing work on API-misuse detection and a qualitative comparison of the re-

16. Conclusion

spective detectors according to MuC. Both static and dynamic detectors focus on only a small subset of all API misuses, mostly neglecting usage elements other than method calls. Empirical evaluations focus on the precision of detectors. Since evaluations generally use different datasets and review different samples of detector findings, a direct comparison of evaluation results is very unreliable. It seems that detectors focusing on specific types of API misuses are more precise, but this has never been investigated. Studies that directly compare multiple detectors are practically non-existent.

Automated Benchmark for API-Misuse Detectors Using the examples of API misuse we identified, we build MUBENCH, the first-ever automated empirical benchmark for API-misuse detectors. MUBENCH enables systematic, comparable, and reproducible experiments, measuring both a detectors' precision and recall. It automates large parts of the evaluation process, including the retrieval of project source code and Bytecode, the execution of detectors, and the preparation of their findings for manual review. And it drastically reduces the remaining effort of the necessary manual reviews through pre-filtering of relevant detector findings. Furthermore, MUBENCH is easily extensible by new misuse examples and additional detectors for further experiments.

Performance of State-of-the-Art Misuse Detectors We use MUBENCH for a systematic evaluation and comparison of four state-of-the-art misuse detectors. We find that all these detectors may successfully identify many misuses (up to 48%), but suffer from extremely low precision (below 12%) and recall (below 21%) in a practical setting. Our root-cause analysis of the detectors' false positives and false negatives reveals 13 reasons for their bad performance. From them, we derive possible directions towards more powerful misuse detectors.

Improved Detection of API Misuses Based on the results of the previous empirical study, we develop MUDETECT, a new API-misuse detector that mitigates many of the problems of existing detectors. MUDETECT uses API Usage Graphs (AUGs), a new graph-based representation of API usages. We specifically design AUGs to simultaneously capture many properties of API usages that can distinguish misuses from correct usages. MUDETECT employs a pattern-mining and a violation-detection algorithm that efficiently and effectively identify usage patterns and misuses based on AUGs. Our empirical evaluation shows that MUDETECT outranks existing detectors by factor 2.5 in terms of precision and by factor 2 in terms of recall, in the typical per-project setting. In a cross-project setting, MUDETECT achieves a precision of 34.1% and a recall of 42.2%, thereby outranking the other detectors by factor 3.9 with respect to both metrics.

Possible Directions Towards Further Improvement An analysis of MUDETECT's findings shows that we need to consider usage examples from both within target projects and across different projects, to mine the necessary patterns for effective misuse detection. Furthermore, we find that future work should explore the use of inter-procedural program analyses and the possibilities of probabilistic usage models. A detailed discussion

of these research opportunities follows below.

16.2. Closing Discussion

Reuse of existing software components is an integral ingredient to efficient software development. Software developers use such components through their APIs. Thereby, they must consider the usage constraints that come with these APIs. The work presented in this thesis shows that failing to do so, i.e., misusing the API, is a prevalent cause of severe software defects and that struggling with misuses may considerably slow down the development process.

Researchers have dedicated much work to the automated detection of API misuses. This thesis consolidates over a decade of research for a qualitative and quantitative assessment of the state of the art. We find that existing detectors conceptually cover only a small subset of all types of API misuses. Moreover, we find that while detectors may potentially detect many misuses, their actual precision and recall are extremely low (below 10%). We analyze their false positives and false negatives to determine the respective root causes. Based on our findings, we systematically design a new static API-misuse detector, *MUDETECT*, that improves on the state of the art. *MUDETECT* achieves a precision of 34.1% at a recall of 42.2%.

Nevertheless, it is clear that misuse detection is not yet mature enough for practical applicability. Existing detectors either focus on very specific types of misuses (see Chapter 4) or have false-positive rates way above the 20% that practitioners report as the highest acceptable rate [BBC⁺10]. However, the work presented in this thesis shows that through systematic analysis of the problems of state-of-the-art misuse detectors and a strategic design of new approaches, we can improve on both precision and recall at the same time. Our results also show that there is further potential for improvement, which is why we are convinced that misuse detection can reach a practical level. We present several challenges that future work should address, in order to reach this goal.

First, current detectors employ rather simple code analyses, such as intra-procedural analysis assuming static single assignment or analysis of simple method-call traces. Imprecisions of these analyses cause many of the detectors' false positives. More advanced analysis techniques, such as inter-procedural analyses, points-to analyses, or abstract interpretation, might deliver much more precise information, which may help to improve overall detection quality.

Second, many current detectors employ frequency-based approaches to mine API specifications, i.e., they assume that frequent usages are correct and infrequent usages, consequently, incorrect. Our empirical results show that this assumption is the main reason for false positives across all detectors. We should search for alternative metrics for usage correctness to replace pure frequency and develop techniques to prune specifications, e.g., considering the implementation code of APIs, to avoid reporting violations of meaningless specifications. We should also advance probabilistic API-usage models [NPVN15, MCJ17], as a means to avoid binary decisions about correctness entirely.

Third, our results show that the recall of current detectors is severely limited by the

16. Conclusion

number of available usage examples to learn specifications from. To mitigate this, we should investigate techniques that scale to larger training datasets and approaches for targeted retrieval of usage examples, e.g., through code-search engines [TX09a, TX09b] or from answers on STACKOVERFLOW [GZW⁺15]. We should also investigate approaches ensure the quality of the training examples we use. Moreover, we should combine examples from different sources, to maximize the available training data.

17. Future Work

In this chapter, we present our ideas for future work, with respect to benchmarking API-misuse detectors, as well as advancing the state of the art in API-misuse detection. For some of these ideas we present preliminary results. Others we identify as interesting challenges.

17.1. Benchmarking API-Misuse Detectors

Mining Misuse Examples from Project History Despite the huge manual effort that we invested to create our benchmarking dataset, the number of misuse examples in the current dataset is still relatively small, especially because we had to exclude examples where we could not compile the respective project version that contain them. This bears the danger that the dataset is not representative for API misuses as they appear in-the-wild. To mitigate this threat, future work should explore ways to obtain large numbers of misuse examples with less manual effort. For example, Dallmeier *et al.* [DZ07] present an approach to automatically classify bugs by the changes applied in their fixes. It may be possible to leverage such an approach to automatically identify examples of API misuses in the version-control history of software projects. This would enable us to obtain misuse examples for our benchmarking dataset on a much larger scale.

Defining a Minimal Benchmark The MUBENCH dataset contains all examples of API misuses that we identified in compilable project versions throughout the work presented in this thesis. Thereby, we included multiple instances of some misuses, i.e., multiple usages that violate the same API-usage constraint. While this reflects the distribution of violations as they appear in the wild, we argue that it would be interesting to also define a minimal benchmark dataset with only unique misuses. In order to achieve this, future work should develop a notion of *misuse equivalence*. Note that misuse equivalence cannot simply be defined via the equality of usages, since two misuses might contain different additional elements that do not contribute to the misuse. Furthermore, our experiments for measuring the recall upper bound of detectors have shown that the context in which a misuse appears may impact detectors. The code surrounding a usage may introduce noise, which leads to a detector identifying one instance of a particular misuse, while missing another. We need to consider such factors in the creation of a minimal benchmark that enables a fair comparison of misuse detectors.

Studying the Impact of API Misuse Throughout the Development Process We found initial evidence that developers struggle with API misuses already at development time

17. Future Work

and that these misuses might be different from the misuse examples we identify in issue trackers and version-control systems (see Section 2.3). However, we know little about the actual impact of API misuse at different stages of the development process and the potentially distinctive properties of misuses at any particular stage. To find out more, future work could mine Q&A sites or conduct surveys and field studies to learn more about which problems developers face at different stages. This would enable a systematic comparison to reveal whether there are indeed differences and how these impact research on API-misuse detection.

Reducing Review Effort for Measuring Recall MUBENCH significantly reduces the effort of measuring the recall of misuse detectors. For example, we had to review only 0.3% (62) of all findings of the four detectors in Experiment R (see Section 5.5) due to pre-filtering for potential hits by their code location. However, the overall effort of the necessary manual reviews across all experiments is still high.

It is likely impossible to reduce the number of reviews in Experiment P, where we review the top-20 findings per detector and project, because we lack a-priori knowledge about the findings. Instead, future work could develop techniques that speed up individual reviews. One possibility might be to automatically group similar findings, to allow bulk reviews. Another orthogonal approach might be to automatically provide reviewers with additional information for the reviews, such as documentation for the API(s) in question.

On the other hand, we believe that is possible to further reduce the number of necessary reviews in the experiments that measure recall. Currently, MUBENCHPIPE pre-filters the detectors' findings in these experiments based on their file and method location. Since we are checking for known misuses, we could, for example, additionally consider the name of the misused API to further reduce the number of potential hits to review. Such filtering, however, needs to be carefully designed, in order to guarantee that all true positives are retained.

Benchmarking Further API-Misuse Detectors Our empirical assessment of the state of the art in API-misuse detection is limited to four static misuse detectors that we could obtain implementations of (see Chapter 7). Our survey in Chapter 4 identifies many more detectors, some of which realize quite distinctive ideas, such as obtaining usage examples for specification mining via a code-search engine [TX09a, TX09b] or learning correct usage in probabilistic models [NPVN15, MCJ17]. We believe it is important to incorporate such approaches into the benchmark, in order to get a more complete picture of the state of the art and identify the strength and weaknesses of the different approaches. As the work presented in this thesis shows, such systematic assessment can reveal opportunities for significant improvements.

For some detectors, where previous implementations are unavailable, non-functional, or target different programming languages, we need to reimplement the respective concepts. For other detectors, we need to advance MUBENCH. For example, to integrate dynamic detectors, we must ensure that the target project versions are executable, in

addition to being compilable, and that respective test harnesses are available to execute them. For yet other detectors, we might approach integration from either direction. For example, to integrate detectors that work on Dalvik Bytecode, we might either migrate them to read general Java Bytecode or provide the project versions in MUBENCH also as Dalvik Bytecode, which is generally feasible, because we can compile Java Bytecode to Dalvik Bytecode.

17.2. Advancing API-Misuse Detection

Leveraging Advanced Static-Analysis Techniques Static misuse detectors analyze program code to extract their usage representations. Our experiments show that imprecisions and limitations in these static analyses can introduce noise that causes false positives and false negatives (see Section 7.1, Section 7.2, Section 13.2, and Section 13.3). Nevertheless, current static detectors employ only simple intra-procedural analyses. Future work should investigate whether advanced analysis techniques, such as inter-procedural analyses or abstract interpretation, can deliver more precise information that helps to improve overall detection quality. We obtained some preliminary insights by using the static-analysis framework OPAL [EH14] to extract AUGs from Java Bytecode:

For example, using a basic inter-procedural data-flow analysis allows us to detect for fluent APIs, such as `StringBuilder`, that all its methods return the receiver object. This allows us to represent the receiver and the return value by the same data node, such that we generate the same AUG regardless of whether a usage is written as `sb.append().toString()` or as `sb.append();sb.toString()`. Such unification over semantically equivalent usages effectively eliminates false positives.

Another promising idea is to inline code from private helper methods into their callers, replacing the calls to the private methods themselves. This follows two observations:

1. Private methods are themselves not part of an API. Calls to them should, therefore, not be considered when learning API-usage patterns or detecting misuses. Since our algorithms do not distinguish calls based on method visibility, removing calls to private methods from AUGs effectively removes noise.
2. Private methods often encapsulate parts of usages on their parameters. Consequently, these usages are distributed across these private methods and their callers. Inlining the private methods into the callers joins the partial usages together, increasing the chance that we capture the complete usage, which may eliminate false positives.

A particularly interesting phenomenon that inlining causes is the *duplication of evidence*. When we inline a private method into multiple callers, the AUG that we previously generated for the private method itself now appears as a subgraph in each of the AUGs we generate for the callers. We note that this is similar to how call traces analyzed by dynamic misuse detectors contain the subtrace generated by the execution of a private method once for every call to that method. We observed several partially opposing effects of such duplication:

17. Future Work

- From the perspective of our pattern mining algorithm, the support of the inlined subgraph increases, which may lead to additional or larger patterns. This may reduce false positives, if we mine more alternative patterns. It may also cause false negatives, if the private method contains a violation, which now becomes frequent enough to be mined into a pattern.
- From the perspective of our detection algorithm there are now fewer, larger target AUGs to check. This may reduce false positives, if what previously appeared as two usages with missing elements in two AUGs now appears as one pattern instance in a single AUG. It may also duplicate both true positives and false positives, if a usage with missing elements from the private method now appears in all AUGs we generate its callers.

These examples show that leveraging advanced static-analysis techniques may benefit misuse detection. However, future work needs to systematically investigate the effects and counter-effects of individual techniques, to see whether we can balance them out in favor of the overall detector performance.

Extending Usage Models of Dynamic Detectors Current dynamic detectors work solely on method-call traces. While this allows a precise analysis of which methods are invoked on an API and in which order, it neglects data flow between calls and control dependencies among calls. Consequently, respective detectors cannot detect violations involving usage elements such as conditions or exception handling. Future work should investigate whether and how the concept of dynamic detection can be extended in this regard.

Calculating the Support of Specifications The most common way to determine the support of a specification is to count the number of usages that adhere to the specification, either in the code (statically) or in the execution traces (dynamically). We call this the *occurrence support* of a specification. It follows the intuition that a specification that holds more frequently is more likely to generalize.

A possible alternative is the *method support*, which counts the number of methods that contain at least one usage following the specification. If we interpret methods as code units implementing a particular task, the method support can be interpreted as the number of tasks using a certain API according to the specification. The method support is smaller than or equal to the occurrence support, as it ignores additional usages following the same specification within the implementation of the same task. We hypothesize that this might reduce evidence for violations, as multiple usages within the same task are likely written under the same (mis)conception of the APIs constraints.

Another alternative is the *project support*, which counts the number of projects that contain at least one usage following the specification. We used this definition for MU-DETECTXP in Experiment XP (see Section 12.2.5). If we interpret projects as the work of different development teams, the project support can be interpreted as a measure of how many teams believe this specification to be correct. We hypothesize that this might

be a better indicator for correctness than occurrence or method support, as different teams are more unlikely to share the same misconceptions.

Future work should investigate and compare the impact of these (and possible other) ways to calculate the support of specifications on the quality of misuse detectors. Moreover, future work may investigate ways to combine different support metrics.

Determining Thresholds on the Support of Specifications The detectors in our experiments all use *absolute minimum-support thresholds* to mine specifications. This follows the intuition that a specification that holds at least a certain number of times, is likely to be correct. However, with an increasing number of training examples, e.g., when mining from larger projects, it becomes more likely that misuses also appear more frequently than a given absolute threshold.

A possible alternative are *relative minimum-support thresholds*. For example, we may determine a global threshold relative to the total number of training examples. This follows the intuition that a correct specification should hold more often in a larger training dataset. We may also determine individual thresholds relative to the total number of training examples for the API(s) involved in a specification, as proposed by Ramanathan *et al.* [RGJ07a, RGJ07b], Thummalapenta *et al.* [TX09a, TX09b], and Acharya *et al.* [AX09]. Intuitively, this leads to mining specifications that are more likely to capture general usage constraints of the respective APIs, but is less likely to mine specifications for alternative usage patterns.

Finally, we may consider several different thresholds at once. For example, Thummalapenta *et al.* [TX09a] first mine specifications with a high absolute threshold. Then they collect the training examples that do not conform to any mined specification and repeat mining on them, using a lower absolute support threshold. They report that considering such lower-support specifications may reduce false positives by up to 28%. Similarly, Saied *et al.* [SBAS15] first mine specifications with a high absolute threshold and then successively mine extensions to these specifications using ever lower thresholds. This follows the observation that there is often a strict core specification that all usages of an API must adhere to, and several alternative extensions to it.

To the best of our knowledge, no previous work systematically compared alternative ways to determine support thresholds. Future work should investigate how they impact the quality of API-misuse detection.

Replacing Support as a Metric for Correct Usage Most current detectors mine API specifications based on the support for that specification in a given training dataset. Our empirical results show that support is not a good indicator for correctness and low support is an even worse indicator for incorrectness. Future work could develop techniques to prune specifications, e.g., considering the implementation code of APIs, to avoid reporting violations of meaningless specifications. Future work should also investigate ways to avoid frequency thresholds entirely by using probabilistic API-usage models, e.g., based on the ideas of Nguyen *et al.* [NPVN15] or Murali *et al.* [MCJ17], or usage models capturing tpestates, e.g., similar to Pradel *et al.* [PJAG12, PG12].

Mining Usage Examples from Further Sources Our results show that the recall of current detectors is severely limited by the number of available usage examples to learn specifications from (see Section 13.5). To mitigate this, future work should investigate techniques that scale to larger training datasets and approaches for targeted retrieval of usage examples, e.g., through code-search engines, as proposed by Thummalapenta *et al.* [TX09a, TX09b], or from answers on STACKOVERFLOW, as proposed by Gao *et al.* [GZW⁺15]. Future work might also leverage changesets of fixes, to learn about both correct and incorrect usage. And future work should develop approaches that combine data from multiple sources, because we find that different sources may provide mutual exclusive information (see Section 13.5) and because it is likely that for new APIs there is insufficient information available through any single source.

Mining High-Quality Usage Examples In our work, we show that simply providing larger quantities of usage examples for the APIs under analysis serves to significantly improve misuse detection. We hypothesize that we might be able to further improve the detection, by ensuring that the provided examples have high quality. While measuring code quality remains generally an open question, using simple indicators like project maturity, code churn, or number of tests might already improve detection results. Future work should investigate respective possibilities.

Presenting Detector Findings to Developers We conducted a preliminary user study [Wei16] in which we showed developers different mockups of how a misuse detector might present its findings in an IDE. This study revealed a major challenge for the practical applicability of misuse detectors, namely the need to explain and justify the detector’s findings. For example, when shown a misuse of an API that the participants were unfamiliar with, they mostly replied that they could not ultimately judge the detector’s finding—even if it appeared to be sensible—without studying the respective API’s documentation. And when shown a misuse and a corresponding fix, participants often argued that the fix might be unnecessary, depending on the context of the usage (which we did not provide in the study). Future work should develop techniques to generate explanations and justifications of misuse detectors’ findings to gain the trust of developers in using this kind of tool. Furthermore, future work should investigate techniques to present misuse-detector findings in a way that is intuitive to developers.

Contributed Implementations and Data

In the course of the projects presented in this thesis, research prototypes have been implemented and data has been sourced or collected. We provide these implementations and datasets to enable other researchers to validate our work and to build new research upon them. We believe this to be good scientific practice and encourage other researchers to do the same.

MuBench

The dataset of API-misuse examples that we collected throughout this thesis is publicly available as part of our automated benchmark. The dataset can be accessed from:

`https://github.com/stg-tud/MUBench/tree/master/data`

It is presented in a folder structure that follows the data scheme presented in Section 6.1. The metadata about projects, project versions, and misuses is stored in YAML files, following the YAML 1.2 format (3rd Edition).¹ In the root folder resides a file named `dataset.yml`, which specifies several sub-datasets for each of the different sources that we obtained misuse examples from and for each of the experiments presented throughout this thesis.

MuBenchPipe

The benchmarking pipeline presented in Part II consists of a command-line application for conducting experiments, written in PYTHON 3.5, and a review website, written in PHP 7. We provide the entire source code here:

`https://github.com/stg-tud/MUBench`

We provide a DOCKER container that allows running the pipeline independent of the host platform. We use COMPOSER for the dependency configuration of the review site. The README files of the GITHUB repository present detailed instructions on how to setup and use both components of MUBENCHPIPE.

MuDetect

We provide the implementation of MUDETECT presented in Chapter 11 here:

¹ <http://www.yaml.org/spec/1.2/spec.html> (checked on Dec 14, 2017)

<https://github.com/stg-tud/MUDetect>

The detector is fully integrated into MUBENCH to allow execution in the context of the respective experiments. The MUDetect project uses APACHE MAVEN for its build configuration, to allow system- and IDE-independent builds. The detector can be built using the `mvn package` command, which creates standalone bundles called `MuDetect.jar` and `MuDetectXP.jar` for the respective variants of our detector in the `target` folder. The README files of the GITHUB repository present further details on the organization of the project and its code.

Study Artifacts

We provide the questionnaire and all responses to our developer survey on API misuse and the full list of STACKOVERFLOW threads we reviewed in our study on the prevalence of problems leading to a `ConcurrentModificationException` (see Section 2.3) on respective artifact pages:

<http://www.st.informatik.tu-darmstadt.de/artifacts/api-misuse-survey/>

<http://www.st.informatik.tu-darmstadt.de/artifacts/stackoverflow-cme/>

We provide the review results and additional data artifacts for both the study presented in Chapter 7 and the evaluation presented in Chapter 12 on the respective artifact pages:

<http://www.st.informatik.tu-darmstadt.de/artifacts/mustudy/>

<http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/>

The full experiment data may be downloaded from the review sites in CSV format or viewed online.

Bibliography

- [ABF⁺16] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for. The impact of information sources on code security. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2016.
- [ANN⁺16] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th Working Conference on Mining Software Repositories*, MSR '16. ACM Press, 2016.
- [ANN⁺18a] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. MUDetect: The next step in static API-misuse detection. In *Under Review*, 2018.
- [ANN⁺18b] Sven Amann, Hoan A. Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A systematic evaluation of static API-misuse detectors. *IEEE Transactions on Software Engineering*, 2018.
- [AW05] Martín Abadi and Bogdan Warinschi. *Password-based Encryption Analyzed*, pages 664–676. Springer-Verlag GmbH, 2005.
- [AX09] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, FASE '09*, pages 370–384. Springer-Verlag GmbH, 2009.
- [BBC⁺10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- [BBMZ16] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, volume 1. IEEE Computer Society Press, 2016.

Bibliography

- [BGH⁺06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *ACM SIGPLAN Notices*, 41(10):169–190, 2006.
- [BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09*, pages 213–222. ACM Press, 2009.
- [BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 255–264. ACM Press, 1997.
- [Boe99] B Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.
- [BRT12] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *Lecture Notes in Computer Science*, pages 312–329. Springer-Verlag GmbH, 2012.
- [BW08] Raymond P L Buse and Westley R Weimer. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'08. ACM Press, 2008.
- [CBC⁺92] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.
- [CHK⁺09] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the International Workshop on Defects in Large Software Systems*, DEFECTS '09, pages 16–20. ACM Press, 2009.
- [DH09] Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 320–330. IEEE Computer Society Press, 2009.

- [DKM⁺10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96. ACM Press, 2010.
- [DLMK10] Tuan-Anh Doan, David Lo, Shahar Maoz, and Siau-Cheng Khoo. LM: A miner for scenario-based specifications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2 of *ICSE '10*, pages 319–320. ACM Press, 2010.
- [DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 422–431. IEEE Computer Society Press, 2013.
- [DZ07] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering*, ASE '07, pages 433–436. ACM Press, 2007.
- [EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the Conference on Computer & Communications Security*, CCS'13, pages 73–84. ACM Press, 2013.
- [ECH⁺01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72. ACM Press, 2001.
- [EH14] Michael Eichberg and Ben Hermann. A software product line for static analyses: The opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6. ACM Press, 2014.
- [EW98] K. El Emam and I. Wiczorek. The repeatability of code defect classifications. In *Proceedings 9th International Symposium on Software Reliability Engineering*, pages 322–333. IEEE Computer Society Press, 1998.
- [FHM⁺12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61. ACM Press, 2012.
- [FLL⁺02] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, Raymie Stata, Mark Lillibridge, Greg Nelson, James B Saxe,

- and Raymie Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, 2002.
- [GFX⁺10] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 475–484. ACM Press, 2010.
- [GIJ⁺12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49. ACM Press, 2012.
- [GS67] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*, volume 46. Aldine de Gruyter, 1967.
- [GS10] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24. ACM Press, 2010.
- [GW97] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., 1st edition, 1997.
- [GWZ10] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISTA '10, pages 119–129. ACM Press, 2010.
- [GZW⁺15] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Proceedings of the 30th ACM/IEEE International Conference on Automated Software Engineering*, ASE '15, pages 307–318. IEEE Computer Society, 2015.
- [HDG⁺11] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. *Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings*, chapter On the Extent and Nature of Software Reuse in Open Source Java Projects, pages 207–222. Springer-Verlag GmbH, 2011.
- [HJZ13] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 392–401. IEEE Computer Society Press, 2013.

- [HKNP06] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. *PRISM: A Tool for Automatic Verification of Probabilistic Systems*, pages 441–444. Springer-Verlag GmbH, 2006.
- [HM05] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125. IEEE Computer Society Press, 2005.
- [HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.
- [HWM05] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE ’13*, pages 237–240. ACM Press, 2005.
- [IEE10] IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.
- [JA09] Ciera Jaspan and Jonathan Aldrich. *Checking Framework Interactions with Relationships*, pages 27–51. Springer-Verlag GmbH, 2009.
- [JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA ’14*, pages 437–440. ACM Press, 2014.
- [JMLR12] Dongyun Jin, Patrick O’Neil Meredith, Choonghwan Lee, and Grigore Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering, ICSE ’12*, pages 1427–1430. IEEE Computer Society Press, 2012.
- [JS13] Brittany Johnson and Yoonki Song. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering, ICSE ’13*. IEEE Computer Society Press, 2013.
- [KSA⁺18] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP’18*, 2018.
- [LCWZ14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems, APSys ’14*, pages 7:1–7:7. ACM Press, 2014.

Bibliography

- [LHX⁺16] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE '16*, pages 602–613. ACM Press, 2016.
- [Lin07] Christian Lindig. Mining patterns and violations using concept analysis. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2007.
- [Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, 1987.
- [LJMR12] Choonghwan Lee, Dongyun Jin, Patrick O’Neil Meredith, and Grigore Roşu. Towards categorizing and formalizing the JDK API. Technical report, University of Illinois at Urbana-Champaign, 2012.
- [LKL08] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.
- [LLQ⁺05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
- [LZ05] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '13*, pages 306–315. ACM Press, 2005.
- [LZL⁺14] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O’Neil Meredith, Traian Florin Şerbănuţă, and Grigore Rosu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Runtime Verification*, pages 285–300. Springer-Verlag GmbH, 2014.
- [MBM10] Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting missing method calls in object-oriented software. In *Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP '10*, pages 2–25. Springer-Verlag GmbH, 2010.
- [MCJ17] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding API usage errors. In *Proceedings of the 11th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '17*, pages 151–162. ACM Press, 2017.

- [METM12] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.
- [MGP⁺11] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11*, pages 111–120. ACM Press, 2011.
- [MM13] Martin Monperrus and Mira Mezini. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–25, 2013.
- [NK11] Anh Cuong Nguyen and Siau-Cheng Khoo. Extracting significant specifications from mining through mutation testing. In *Formal Methods and Software Engineering*, pages 472–488. Springer-Verlag GmbH, 2011.
- [NKMB16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. ”Jumping through hoops”: Why do developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering, ICSE’16*. ACM Press, 2016.
- [NNP⁺09a] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, FASE ’09*, pages 440–455. Springer-Verlag, 2009.
- [NNP⁺09b] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE ’09*, pages 383–392. ACM Press, 2009.
- [NNP⁺12] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.
- [NNW⁺10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA’10*, pages 302–321. ACM Press, 2010.
- [NPVN15] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Recommending API usages for mobile apps with Hidden Markov Model. In *Proceedings*

- of the 30th ACM/IEEE International Conference on Automated Software Engineering, ASE '15, pages 795–800. IEEE Computer Society Press, 2015.
- [NPVN16] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Learning API usages from bytecode : A statistical approach. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16. ACM Press, 2016.
- [PADP⁺12] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *Proceedings of the 20th IEEE International Conference on Program Comprehension*, ICPC '12, pages 63–72. IEEE Computer Society Press, 2012.
- [PBDP⁺14] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 102–111. ACM Press, 2014.
- [PBG10] Michael Pradel, Philipp Bichsel, and Thomas R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10. IEEE Computer Society Press, 2010.
- [PG09] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382. IEEE Computer Society Press, 2009.
- [PG12] Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 288–298. IEEE Computer Society Press, 2012.
- [PJAG12] Michael Pradel, Ciera Jaspán, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 925–935. IEEE Computer Society Press, 2012.
- [PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84. IEEE Computer Society Press, 2007.
- [PLM15] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–31, 2015.

- [RBK⁺13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.
- [REHM17] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. Hermes: Assessment and creation of effective test corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 43–48. ACM Press, 2017.
- [RGJ07a] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE ’07, pages 240–250. IEEE Computer Society Press, 2007.
- [RGJ07b] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, pages 123–134. ACM Press, 2007.
- [RJ86] L. R. Rabiner and B. H. Juang. An introduction to Hidden Markov Models. *IEEE ASSp Magazine*, 3(1):4–16, 1986.
- [RMWZ14] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer-Verlag GmbH, 2014.
- [RP15] T Ramraj and R Prabhakar. Frequent subgraph mining algorithms – a survey. *Procedia Computer Science*, 47:197–204, 2015.
- [RWZ10] Martin P Robillard, Robert J Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.
- [SBAS15] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level API usage patterns. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER ’15, pages 23–32. IEEE Computer Society Press, 2015.
- [SDG⁺16] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE ’16, pages 21–30. ACM Press, 2016.
- [SE13] Widura Schwittek and Stefan Eicker. A study on third party component reuse in Java enterprise open source software. In *Proceedings of the 16th International ACM SIGSOFT Symposium on Component-based Software Engineering*, pages 75–80. ACM Press, 2013.

Bibliography

- [SHA15] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. Searching the state space: A qualitative study of API protocol usability. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension, ICPC '15*, pages 82–93. IEEE Computer Society Press, 2015.
- [TAD⁺10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proceedings of the Asia Pacific Software Engineering Conference, APSEC '10*, pages 336–345. IEEE Computer Society Press, 2010.
- [TR16] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 392–403. ACM Press, 2016.
- [TX09a] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 283–294. IEEE Computer Society Press, 2009.
- [TX09b] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 496–506. IEEE Computer Society Press, 2009.
- [UR15] Gias Uddin and Martin P Robillard. How API documentation fails. *IEEE Software*, 32(4):68–75, 2015.
- [Wei16] Simon Weiler. Integrating an API-misuse detector into Eclipse. Master’s thesis, Technische Universität Darmstadt, 2016.
- [WLW⁺11] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Iterative mining of resource-releasing specifications. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11*, pages 233–242. IEEE Computer Society Press, 2011.
- [WPVS17] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatically generating natural language descriptions for object-related statement sequences. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER '17*, pages 205–216. IEEE Computer Society Press, 2017.
- [WZ11] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

- [WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 6th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '07, pages 35–44. ACM Press, 2007.
- [ZGC⁺17] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*, ICSE '17, pages 27–37. IEEE Computer Society Press, 2017.
- [ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, PROMISE '07. IEEE Computer Society Press, 2007.
- [ZS13] Hao Zhong and Zhendong Su. Detecting API documentation errors. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages & Applications*, volume 48, pages 803–816. ACM Press, 2013.
- [ZS15] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923. IEEE Computer Society Press, 2015.

Appendix

A. Six Basic Rules for Safe Usage of the Java Cryptographic Architecture APIs

Egele *et al.* [EBFK13] report that 88% (10,327 of 11,748) Android apps on the Google Play marketplace violate at least one of six basic rules for safe usage of the Java Cryptographic Architecture (JCA) APIs. These six basic rules are:

- R1** Do not use ECB mode for encryption – the Electronic Codebook (ECB) mode is considered insecure, since with it identical plaintext is converted into identical cipher text, thus, preserving data patterns.¹
- R2** Do not use a non-random initialization vector (IV) for CBC encryption – using a non-random IV, i.e., hard coding the IV in the program, nullifies the random noise that the IV is intended to introduce and allows attackers to remove the remaining constant noise by using the IV that can be extracted from the code.²³
- R3** Do not use constant encryption keys – using a constant, i.e., hard-coded, key allows attackers to extract that key from the code and undo any encryption.
- R4** Do not use constant salts for PBE – using constant salts for password-based encryption, i.e., using the same salt for every password, effectively nullifies the random noise that the salt is intended to introduce, making it easier for attackers to guess passwords [AW05, BRT12].
- R5** Do not use fewer than 1,000 iterations for PBE – using fewer iterations effectively makes cryptographic keys generated in password-based encryption less random [AW05, BRT12].
- R6** Do not use static seeds to seed `SecureRandom` – using static seeds leads to pseudo random, i.e., reproducible, random number sequences, which nullifies the effect of the randomness.

¹ <http://cseweb.ucsd.edu/~mihir/cse207/classnotes.html> (checked on Dec 26, 2017)

² <http://www.openssl.org/~bodo/tls-cbc.txt> (checked on Feb 16, 2018)

³See footnote 1.

B. BOA API Usage

We used the following script to search for GitHub projects using a particular API via the BOA repository-mining infrastructure [DNRN13] in Section 12.2.5. Before sending the query, we replaced `TARGET_TYPE` by the API's fully qualified type name, e.g., `java.util.List`, `PACKAGE_STAR_NAME` by the respective type's package name with a wildcard import, e.g., `java.util.*`, and `SIMPLE_TYPE` by the type's simple type name, e.g., `List`.

```
1  p: Project = input;
2  out: output set of string;
3
4  files: set of string;
5  revision: Revision;
6  file: ChangedFile;
7  imports_package: bool;
8
9  visit(p, visitor {
10     before r: Revision -> revision = r;
11     before f: ChangedFile -> {
12         if (contains(files, f.name) || match("test", lowercase(f.name))) stop;
13         file = f;
14     }
15     after f: ChangedFile -> add(files, f.name);
16     before astRoot: ASTRoot -> {
17         imports := astRoot.imports;
18         imports_package = false;
19         foreach (i: int; def(imports[i])) {
20             if (imports[i] == "TARGET_TYPE") {
21                 out << p.name;
22                 stop;
23             } else if (imports[i] == "PACKAGE_STAR_TYPE") {
24                 imports_package = true;
25                 break;
26             }
27         }
28     }
29     before variable: Variable -> {
30         if ((imports_package && (variable.variable_type.name == "SIMPLE_TYPE_NAME")) ||
31             (variable.variable_type.name == "TARGET_TYPE")) {
32             out << p.name;
33             stop;
34         }
35     }
36 });
```


C. BOA Cipher Usages

We used the following script to search for GITHUB projects using the **Cipher** API via the BOA repository-mining infrastructure [DNRN13] in Section 2.4. The script ignores any files that have the keyword **test** somewhere in their path or filename, since we consider vulnerabilities in test code as non-critical. The script misses **Cipher** usages that use fully qualified type references, instead of imports. However, this is not common practice in Java, thus, considering this is unlikely to increase the number of projects we identify significantly.

```
1  p: Project = input;
2  out: output set[string] of string;
3
4  files: set of string;
5  revision: Revision;
6  file: ChangedFile;
7
8  visit(p, visitor {
9    before r: Revision -> revision = r;
10   before f: ChangedFile -> {
11     if (contains(files, f.name) || match("test", lowercase(f.name))) stop;
12     file = f;
13   }
14   after f: ChangedFile -> add(files, f.name);
15   before astRoot: ASTRoot -> {
16     imports: = astRoot.imports;
17     foreach (i: int; def(imports[i])) {
18       if (imports[i] == "javax.crypto.Cipher" || imports[i] == "javax.crypto.*") {
19         out[p.name] << p.project_url + "/blob/" + revision.id + "/" + file.name;
20         stop;
21       }
22     }
23   }
24 });
```