1-2023

Fortifying the seams of software systems

Hong Jin KANG
Singapore Management University, hjkang.2018@phdcs.smu.edu.sg

FORTIFYING THE SEAMS OF SOFTWARE SYSTEMS

Hong Jin Kang

SCHOOL OF COMPUTING AND INFORMATION SYSTEMS
SINGAPORE MANAGEMENT UNIVERSITY
2023

Fortifying the Seams of Software Systems

Hong Jin Kang

Submitted to School of Computing and Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

David Lo (Supervisor/Chair)
Professor
Singapore Management University

Lingxiao Jiang
Associate Professor
Singapore Management University

Hady W. Lauw
Associate Professor
Singapore Management University

Ming Li
Professor
Nanjing University

Singapore Management University
2023

I hereby declare that this dissertation is my original work and it has been written by me in its entirety.
I have duly acknowledged all the sources of information which have been used in this dissertation.

This dissertation has also not been submitted for any degree in any university previously.

Hong Jin Kang
5 January 2023

Fortifying the Seams of Software Systems

Hong Jin Kang

A seam in software is a place where two components within a software system meet. There are more seams in software now than ever before as modern software systems rely extensively on third-party software components, e.g., libraries. Due to the increasing complexity of software systems, understanding and improving the reliability of these components and their use is crucial. While the use of software components eases the development process, it also introduces challenges due to the interaction between the components.

This dissertation tackles problems associated with software reliability when using third-party software components. Developers write programs that interact with libraries through their Application Programming Interfaces (API). Both static and dynamic analysis of API-using code require knowledge of the API and its usage constraints. Hence, we develop techniques to learn and model the usage constraints of APIs. Next, we apply the insights gleaned from our studies to support bug-finding techniques using static and dynamic analysis. Then, we look into larger software systems comprising multiple components. We propose techniques for mining rules to monitor the joint behaviors of apps, and for exploiting known library vulnerabilities from a project importing a library. These techniques aim to assist developers to better understand third-party components, and to detect weaknesses in software systems.

The dissertation includes the following contributions:

1. ALP: An approach using active learning to mine GitHub and train an API misuse detector using subgraph patterns. Previous work in this area has relied on learning from examples on GitHub and assuming that frequent usage patterns should be learned. However, frequent examples are not always correct, and correct examples may not be frequent. ALP weakens this assumption by involving a human annotator who labels informative examples. This helps to improve the effectiveness of the detector and reduce the need for large amounts of training data.

2. We have conducted a replication study of techniques for postprocessing the warnings reported by a static analyzer. Static analyzers often produce a large number of false alarms, and many proposed techniques use machine learning to filter out false alarms. However, we have found that the performance of these techniques may have been overoptimistic due to methodological issues in their experiments. As a result, we have proposed a new approach called TrailMarker, which uses few-shot

in-context learning by exploiting a small number of labels on warnings associated with the same traces, including events on the execution paths that invoke library functions. This approach provides a more effective way of filtering out false alarms from static analysis.

3. DICE: A tool that implements our proposed framework, Adversarial Specification Mining, to falsify incorrect rules derived from a collection of execution traces. Uncommon API usage patterns may not be represented in the execution traces, leading to the inference of inaccurate rules. To address this problem, DICE focuses its execution of an API on falsifying each inferred rule. This allows for the creation of a more accurate and refined set of rules, which can be used to infer a behavioral model of the API.

4. SkipFuzz: An active learning approach to learn the input constraints of API. SkipFuzz preprocesses the possible inputs to deep learning libraries. By doing so, it would use inputs that are more likely to provide new information for SkipFuzz to refine its model of the input constraints, e.g. by falsifying a property that SkipFuzz believed had to hold for the input to be valid. This enables SkipFuzz to generate a greater proportion of valid inputs with a high level of diversity. 23 CVE IDs have been assigned to the vulnerabilities found by SkipFuzz.

5. IoTBox: An approach that mines rules for monitoring an Internet-of-Things (IoT) environment. Prior studies use handcrafted safety and security policies to detect these threats from the joint behavior of apps in an environment. These policies may not anticipate all usages of the devices and apps, causing false alarms. Using a formal model of the applications on an IoT environment, IoTBox explores its possible behaviors and encodes them in rules that, after the developer/user validated them, can be used to disallow previously unseen behaviors.

6. TRANSFER: An approach that automatically generates test cases exploiting a vulnerability in a library imported in a project (a client program). TRANSFER implements our proposed framework, Test Mimicry. Given a library test case that demonstrates triggers the vulnerability from the library code, TRANSFER aims to trigger the library vulnerability from the client program. Using search-based test generation, TRANSFER construct goals corresponding to the library's program state when the vulnerability is triggered, which guides the generation of a test case that triggers the library vulnerability.

Table of Contents

Acknowledgements

This Ph.D. dissertation is the culmination of years of hard work. This was a long and challenging period of time. This dissertation was only possible because of the support and generosity of many people.

I thank my advisor, Prof David Lo, for the outstanding mentorship over the past 4.5 years. I am immensely grateful and indebted to David for his patience, encouragement, and support. My research took many twists and turns, and it is only with David's insights and expertise that I managed to find my way to the end of this challenging but rewarding journey.

To Prof Lingxiao Jiang, Prof Hady Lauw, and Prof Ming Li, I extend my thanks for serving on my dissertation committee and for your precious time and effort to review and evaluate my dissertation. The feedback I received from the qualifying exam and the dissertation proposal have shaped the research done in this dissertation.

Additionally, I would also like to thank everyone who gave feedback on this dissertation, including the committee members and Dr Julia Lawall, for their time and effort. This dissertation was significantly improved after incorporating the feedback.

Before starting my PhD, I worked as a research engineer. This brief period of time was deeply influential on the work done in this dissertation. The work we completed in that time shaped my research tastes and honed my research skills. I thank my collaborators who worked on this project with me: Prof David Lo, Prof Lingxiao Jiang, Dr Julia Lawall, Dr Gilles Muller, and Dr Ferdian Thung.

During my PhD, I had the fortune for working with many talented collaborators. I learned many lessons from these collaborations. Without the mentorship, expertise, and support generously given by my collaborators, I would not have been able to complete the work done in my PhD. In alphabetical order, I thank Abhishek Sharma, Andrew E. Santosa, Asankhaya Sharma, Bach Le, Bowen Xu, Chaiyong Ragkhitwetsagul, Corina S. Pasareanu, Ferdian Thung, Gilles Muller, Huy Tu, Imam Nur Bani Yusuf, Jieke Shi, Julia Lawall, Khai Loong Aw, Lingxiao Jiang, Lucas Serrano, Muham-

mad Hilmi Asyrofi, Ming Li, Ming Yi Ang, Pattarakrit Rattanukul, Ratnadira Widyasari, Rahul Yedida, Sheng Qin Sim, Stefanus Haryono, Thanh Le-Cong, Tegawendé F. Bissyandé, Thong Hoang (James), Tim Menzies, Truong Giang Nguyen, Xueqi Yang, Yunbo Lyu, Zhipeng Zhao, Zhou Yang.

I thank both the current members and alumni of our research group. The group was a great place for sharing ideas and I am thankful for the interesting discussions, helpful suggestions, and many more.

I am thankful to the Software Engineering research community, especially the reviewers who impacted and improved my research. My submissions tend to receive insightful reviews that provide constructive feedback for improving the work. My work was built on many inspiring studies and I performed my research while standing on the shoulders of giants.

I would also like to express my gratitude to Singapore Management University (SMU). SMU was a great environment for my research and I would like to thank everyone who provided the resources and support needed for this dissertation. In particular, I thank everyone in the graduate office and research administration, including Chui Ngoh, Caroline, Yar Ling, Chew Hong, Pei Huan, for the support and the immense patience that they must have had for all of my interactions with them.

I thank my family, especially my parents and parents-in-law. I have received tremendous support, companionship, and understanding from my family in the past four years. I would also like to express my thanks to my friends for their support, great memories, and interesting discussions.

Finally, I would like to thank my wife, Wei Ling, for having gone through so much with me. Her love, patience, and support was a driving force for me during the long hours of research. I could not have completed this dissertation without her unwavering encouragement and understanding. I am infinitely grateful.

Chapter 1

Introduction

1.1 Motivation

Software systems are now pervasive in virtually every aspect of our lives. Software reliability is more critical than ever. However, modern software systems are becoming increasingly complex as the use of third-party systems have become ubiquitous. Libraries help to bring down the cost of software development by providing functionalities that can easily be reused, through their APIs (Application Programming Interfaces). The number of libraries on package repositories, such as *PyPI* and *npm*, is rapidly increasing [52, 442]. An average project is now believed to include over 110 libraries.[1] While libraries increase the ease of developing software, they also drive the increasing complexity of software systems, introducing new challenges and risks.

Challenges. The inner workings and implementations of libraries are abstracted through their APIs. This has several implications which pose challenges for software reliability. Firstly, developers may only have a surface-level understanding of the libraries and third-party systems before interacting with them. Secondly, as APIs may be opaque, tools that we have relied on to fortify software systems may not be able to effectively navigate their way around and into the APIs. Thirdly, developers may be unaware of bugs and security weaknesses that arise from the use of libraries with security issues and the composition of smaller software components into a larger system.

To correctly use a library, developers interact with the library through its API. When invoking a library's API, mistakes made by the developers may lead to subtle but severe consequences. While APIs may hide the low-level details of a library, developers have to understand the APIs' intended usages

[1] `https://www.contrastsecurity.com/security-influencers/2021-state-of-open-source-security-report-findings`

and constraints that they may impose. Failing to do so may introduce bugs and defects, which could lead to consequences such as data loss and security vulnerabilities. Close to 10% of all software bugs were found to be caused by API misuses [58]. As such, tools that help developers to understand APIs and detect developer mistakes in their use are important.

Static analyzers and fuzzers are often used to improve software reliability. However, these techniques suffer from limitations. Static analyzers produce a large number of false alarms [215] while fuzzers cannot always generate a large proportion of meaningful, valid inputs [308]. The use of libraries present challenges for static analyzers as they may not be able to correctly understand idioms and domain knowledge of the developers. As such, static analyzers overapproximate possible execution paths, including paths that cannot be taken in reality. Without knowledge of the input constraints of a given function from an API, fuzzers cannot effectively generate structurally and semantically-valid inputs. As such, fuzzers may fail to adequately test the core logic of a library and are stuck testing code for input validation.

Beneath the problems that surface at the API-level, the composition of reusable software components may lead to unexpected behaviors. Even when each component works as expected, their interactions may lead to dangerous behaviors. For example, in an Internet-of-Things environment, programs are triggered given certain conditions, may actuate sensors in the environment and may, in turn, trigger other programs. The joint behaviors may produce unintended results. Furthermore, when reusable software components contain bugs and vulnerabilities, these weaknesses could propagate to larger software systems that used the components as building blocks. Due to this risk, it is critical for developers to better understand possible weaknesses when building systems that comprise these components. Approximately 40% of vulnerabilities in software projects are introduced by the inclusion of a dependency [37]. While developers currently deploy monitoring systems that alert them about vulnerable libraries, these systems are known to produce a large number of false alarms [437]. To support developers in judiciously building larger systems, we need tools that provide more precise warnings.

Thesis. A seam in software is a place where two parts of the software meet [148]. This dissertation takes the perspective that understanding the interaction between components where their seams meet is essential to improving the reliability of modern software systems. The work in this dissertation introduces techniques to prevent bugs in the interaction between software systems. The chapters share a common theme where a useful abstraction or model is identified for capturing salient information about the library/-software system. The abstraction then empowers automated techniques for fortifying software systems.

1.2 Contributions

1.2.1 Static Analysis and APIs

We start by using static analysis to improve the reliability of developer-written programs that use APIs. We propose and investigate a new approach for detecting code that incorrectly use APIs. Next, as static analysis tools tend to produce a large number of false alarms, we look into metohds of identifying and filtering false alarms.

In the first study, this dissertation presents ALP, an approach for detecting API misuses, which are violations of usage constraints of the API. While there have been techniques proposed to detect such misuses, studies have shown that they fail to reliably detect misuses while reporting many false positives. One limitation of prior work is the inability to reliably identify correct patterns of usage. Many approaches conflate a usage pattern's frequency with correct patterns. Due to the variety of alternative usage patterns that may be uncommon but are correct, anomaly detection-based techniques have limited success in identifying misuses. We address these challenges and propose ALP (Actively Learned Patterns), reformulating API misuse detection as a classification problem, where each API usage is identified as a misuse or not. Prior work has shown the promise of representing programs as graphs with complex relationships. ALP mines discriminative subgraphs, capturing relationships between program elements that indicative of correct or misuses of an API. Through limited human supervision, we reduce the reliance on the assumption relating frequency and correctness. The principles of active learning are incorporated to shift human attention away from the most frequent patterns. Instead, ALP samples informative and representative examples while minimizing labeling effort. In our empirical evaluation, ALP substantially outperforms prior approaches on both MUBench [59], an API Misuse benchmark, and a new dataset that we constructed from real-world software projects.

In the second study, we present our research on filtering false alarms of static analyzers. Bug detection approaches based on static analysis, such as FindBugs and Infer, have been adopted by many projects. Still, these approaches are characterized by the large volume of false alarms that are produced. Researchers have proposed techniques to postprocess the warnings, using machine learning techniques on handcrafted features. These features are metrics that are extracted from the characteristics and history of the file, code, and warning. We found that several studies used an experimental procedure that results in data leakage, i.e., the model is trained using information that would not be available at prediction time, and data duplication,

i.e., some testing data is present during training. These issues are subtle but have significant implications. Firstly, the ground-truth labels have leaked into features that measure the proportion of actionable warnings in a given context. Secondly, as the construction of each of the training and testing datasets do not consider the warnings already present in the other, many warnings in the testing dataset appear in the training dataset. Next, we demonstrate limitations in the warning oracle that determines the ground-truth labels. Given warnings reported on a revision of the project, the warning oracle relies on a heuristic comparing these warnings to the warnings reported in a reference revision chronologically in the future of the given revision. We show the choice of reference revision influences the distribution of labels determined by the oracle. Moreover, the heuristic produces labels that do not agree with human oracles. Without data leakage and data duplication, the postprocessing approaches are still effective, but we find that there is still room for improvement as the number of labelled examples required for a machine learning to learn from is still high.

In our analysis, we find that an important aspect of filtering false alarms is the inclusion of the bug triggering path in the postprocessing analysis. For example, some false alarms stem from the static analyzer's inability to account for common code idioms related to commonly used APIs, as well as its inability to handle invocations to third-party libraries. To this end, we propose a new approach, TrailMarker, that exploits in-context learning using large language models to learn from a few labeled examples.

1.2.2 Dynamic Analysis and APIs

Next, we investigate dynamic analysis techniques, which have been used to generate behavioural models of APIs and to find bugs in libraries. In contrast to static analysis techniques, dynamic analysis may fail to cover all behaviors of the program, leading to the inference of an inaccurate model of the program. We propose a method of finding traces of program behaviors that are not represented in a collection of traces. When fuzzing libraries, fuzzers are challenged by the difficulty in generating valid inputs that satisfy the API's input constraints. We propose a method of learning information about the input constraints during fuzzing, which improves the generation of inputs for fuzzing.

In the third study, we present our research on dynamic analysis of APIs. To refine behavioural models of API usages, we use search-based test generation for generating counterexamples to temporal properties, which are used as input to an automata inference algorithm. Our work mines finite-state automata (FSA) from execution traces, collected from the execution of test

cases using the API. To learn accurate specifications, many tests are required. Existing approaches generalize from a limited number of traces or use simple test generation strategies. Unfortunately, these strategies may not exercise uncommon usage patterns of a software system. This may cause incorrect rules to be inferred from the execution traces (e.g. a uncommon pattern may be incorrectly believed to be impossible in the software system). To address this problem, we propose a new approach, adversarial specification mining, and develop a prototype, DICE (Diversity through Counter-Examples). Incorrect rules are caused by missing but possible execution traces, i.e., insufficiently diverse traces. Targeting gaps in the diversity of the test traces, DICE produces new execution traces corresponding to usage patterns that were unrepresented in code executed by the input test suite.

In the fourth study, we extend our insights from the previous chapter to mine API usage constraints for more effective library fuzzing. We focus our attention on deep learning libraries. Deep learning libraries are widely used today, and empirical studies have characterized the properties of inputs expected by the library, as well as the inputs that can trigger bugs and vulnerabilities in them. Building on the domain understanding gained from these empirical studies, we build a fuzzer that can learn input constraints on-the-fly. We propose SkipFuzz, a fuzzer that applies active learning to learn the input constraints to improve fuzzing. Inputs are selected to refine the fuzzer's knowledge of the APIs' input constraints. This enables the fuzzer to produce a greater proportion of valid inputs. This enables quicker enumeration of the inputs that are likely to cause unexpected behaviors. 28 vulnerabilities have been confirmed in TensorFlow, with 23 unique CVE IDs assigned.

1.2.3 Interacting Systems

While the previous studies learn and detect issues in a single program (e.g., a library, or a program using a library), the combination of behaviors of multiple programs may pose challenges to developers. To defend against malicious behaviors from the joint behaviors of multiple programs, we propose a method of mining security policies where individual programs may interact to produce unexpected behaviors. To help developers understand security weaknesses in the libraries included in their programs, we propose a method to generate a test case that provides more information about the exploitability of a library vulnerability from a program.

In the fifth study, we present our research on learning rules about an Internet-of-Things software environment. We report our research on an Internet of Things (IoT) platform where programs may be individually correct,

but threats may emerge from the joint behavior of multiple programs. Rules and policies may be written to allow only expected behaviors or block potentially dangerous behaviors. With these rules, a sandbox or monitoring system can be set up to defend against unexpected behaviors. While prior studies use handcrafted safety and security policies to detect these threats, these policies may not anticipate all legitimate uses of the devices and apps in a smart home, causing false alarms. We propose to mine sandbox rules for securing an IoT environment. After the joint behaviors are analyzed from a bundle of apps and devices, a sandbox can be deployed to disallow previously unseen behaviors. Moreover, the execution of malicious behavior, introduced from software updates or obscured through methods to hinder program analysis, will be blocked.

In the sixth study, we present our research that allows developers to better understand the exploitability of library vulnerabilities. When projects include libraries, library vulnerabilities may propagate and cause these projects to suffer from the same security weaknesses. For example, the Log4Shell vulnerability affected the popular Log4J library and impacted over 8% of the Maven Central repository.[2]. Developers can use Software Composition Analysis tools to warn them of potential library vulnerabilities, but these tools often produce a large number of false alarms, making it difficult for developers to understand and prioritize alerts about library vulnerabilities.

We propose a framework, Test Mimicry, which addresses this problem by constructing a test case that exploits a vulnerability in the library dependencies of a project. It leverages the test cases written by software developers to accompany bugs and vulnerability fixes. Given a test case in a software library that reveals a vulnerability, our approach captures the program state associated with the vulnerability. Then, a test generation tool is guided to construct a test case for the client program that invokes the library and reaches the same program state reached in the library's test case. Our framework is implemented in a tool, TRANSFER, which uses search-based test generation. Based on the library's test case, TRANSFER produces search goals that represent the program state where the vulnerability is triggered. This enables us to construct test cases that demonstrate attacks on a client project, which will provide useful information to developers to understand how the vulnerability could be exploited on their systems.

[2]https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html

1.2.4 Publications.

This dissertation aims to be self-contained. The core of this dissertation reports work that was described in the following first-authored papers:

1. "Active Learning of Discriminative Subgraph Patterns for API Misuse Detection" IEEE Transactions on Software Engineering (TSE 2022). [222]

2. "Detecting False Alarms from Automatic Static Analysis Tools: How Far are We?" IEEE/ACM International Conference on Software Engineering (ICSE 2022) [220]

3. "Adversarial specification mining." ACM Transactions on Software Engineering and Methodology (TOSEM 2021) [223]

4. "IoTBox: Sandbox Mining to Prevent Interaction Threats in IoT Systems." IEEE Conference on Software Testing, Verification and Validation (ICST 2021) [225]

5. "Test Mimicry to Assess the Exploitability of Library Vulnerabilities." ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022) [224]

6. "SkipFuzz: Active Learning-based Input Selection for Fuzzing Deep Learning Libraries" (under submission)

7. "Few-shot In-context Filtering of False Alarms from Static Analyzers" (to be submitted)

Some material from the following papers, whose contributions are not essential to this dissertation but influenced the work reported in the above papers, are included:

1. "Semantic patches for Java Program Transformation (experience report)." European Conference on Object-Oriented Programming (ECOOP 2019) [226]

2. "Assessing the Generalizability of code2vec Token Embeddings." IEEE/ACM International Conference on Automated Software Engineering (ASE 2019) [221]

1.3 Reading Guide

Figure 1.1 illustrates the roadmap of the dissertation. The remainder of this dissertation is organized as follows. Chapter 2 presents an overview of related work on the themes of the dissertation, namely, improving understanding and analysis of APIs, libraries, and the interaction of software systems. Chapter 3 presents ALP, an active learning approach for detecting API misuses. Chapter 4 presents our research on machine learning techniques, including our proposed approach TrailMarker, that filters false alarms from static analyzers. Chapter 5 presents an approach, DICE, that uses a test case generator to obtain more diverse execution traces for learning models of APIs. Chapter 6 presents SkipFuzz, a fuzzer that updates its knowledge about an API's input constraints by learning from the execution outcomes given the inputs generated during fuzzing. Chapter 7 presents IoTBox, a sandbox mining technique to learn rules about the joint behaviors of a system with multiple IoT apps, enabling the construction of a sandbox to defend an environment against malicious attackers. Chapter 8 presents TRANSFER, which constructs a vulnerability-revealing test case for a client program using a vulnerable library. Finally, in Chapter 9, we conclude this dissertation and discuss some potential future research directions.

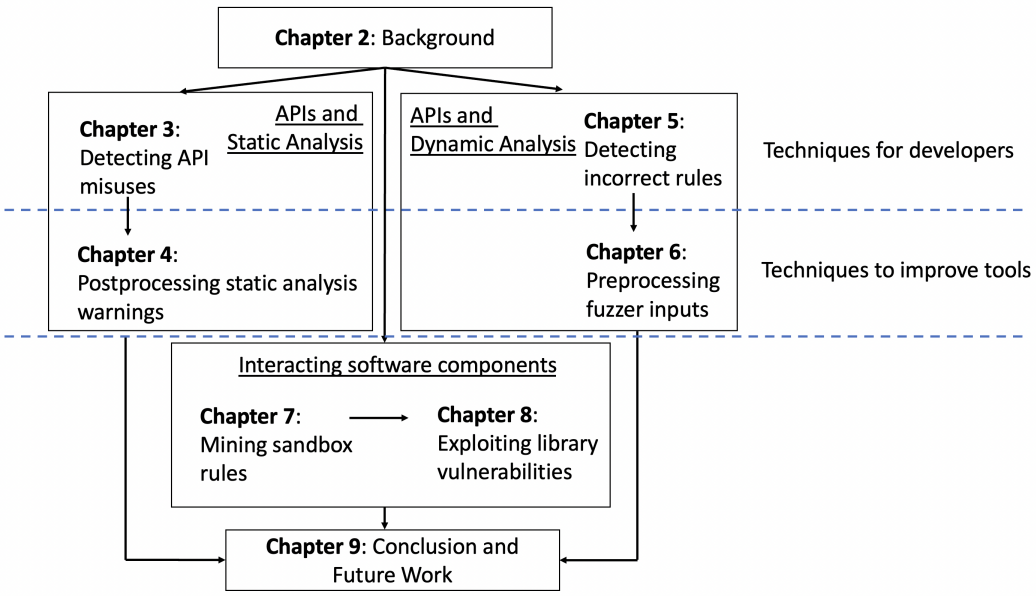


Figure 1.1: Roadmap of this dissertation

Chapter 2

Background

2.1 Third-party Software Components

In modern software development, the use of third-party software components, e.g. libraries and frameworks, are now ubiquitous. While providing benefits, they also introduce new challenges for software development. While developers may understand their own code, they are less likely to understand the code of third-party software components. Developers and their code interact with third-party components through their API, which abstracts away low-level details.

The use of libraries and their APIs presents challenges. These challenges include the need for developers to be aware of and manage breaking changes [419, 88, 292], API deprecations [226, 182, 261, 247], license changes [393], malicious packages [395, 396], as well as bugs and vulnerabilities found in the libraries [156, 332, 203]. Each of these tasks may require substantial effort from developers and may be error-prone. In particular, vulnerabilities in libraries pose challenges as there is growing concern about the opaqueness of the libraries used in a software project. As some libraries, such as Log4J, are widely used, security weaknesses in these libraries have a widespread impact [44, 15]. Libraries may not publicly disclose every vulnerability found in their code [352, 304, 458], and fixes to vulnerabilities may only be released after large delays [204, 329].

APIs can also be challenging to use. While there has been significant effort on improving usability issues [296], a large proportion of bugs are caused by misuses of APIs by developers [58]. APIs impose usage constraints that developers should adhere to. Failing to do so may lead to unexpected behaviors, and may introduce bugs and vulnerabilities [59, 61]. These constraints may not be well-documented [388, 295, 421], making understanding them

challenging for developers. Hence, tools that allow for better understanding of these constraints may be highly valuable.

Many studies have presented methods to improve the reliability of libraries and APIs. Some researchers target developers and propose methods of recommending APIs. Other propose techniques for mining models of API usages [157, 423], synthesizing code using APIs [82], automatically migrating usages of deprecated APIs [305, 226, 182, 247], and finding API misuses [59, 60, 61, 416, 229].

Other research focuses on improving the reliability of the libraries themselves. Some studies propose methods of detecting breaking changes in the libraries [95, 284], detecting malicious packages [395, 396], and finding bugs in the libraries through testing their APIs [213, 206, 447, 415, 132].

Emerging computing platforms, such as an Internet-of-Things (IoT) platform, tend to involve interacting devices and apps [53, 301, 105, 106]. The platform adopts an event-handling paradigm. The apps are bounded to and listens for events from a few physical devices. On receiving events from sensor devices (e.g. a light sensor), the apps may trigger the actuator devices (e.g. a lamp). While providing great convenience, these apps are known to pose a challenge for users to understand [407, 184]. The interaction between multiple apps may lead to unexpected behaviors. Moreover, apps may contain malicious behaviors that are difficult to detect by end users.

2.2 Detecting Bugs

To improve software reliability, both static and dynamic analysis have been used to detect bugs and vulnerabilities. Static analysis tools are able to detect bugs at low cost as they do not execute the program. These tools are well-known to suffer from a large number of false alarms, which hinders their adoption [215, 119, 294]. Dynamic analysis tools execute the programs. While they do not suffer from large numbers of false alarms, they may get stuck in the shallow parts of the programs as entering program states associated with complex constraints can be challenging for fuzzing [308, 421, 325, 326, 70].

Static analyzers, such as Findbugs [75] and Infer [137], report warnings of possible bugs in code. Findbugs includes over 400 bug patterns that match a range of possible bugs, such as null pointer dereferences. They have been known to find a large number of bugs and have been adopted in many different code contexts. Still, many warnings are false alarms, which limits its adoption as developers do not trust its warnings [215, 119]. Many factors, including the use of APIs, may contribute to false alarms. Many studies have proposed methods of improving static analyzers. Several studies [404, 181, 186, 232,

241, 351, 435, 417, 364, 234] postprocess the warnings reported to attempt to filter out false alarms or to prioritize warnings that are more likely to be true alarms.

Dynamic analysis techniques include both test case generation tools and fuzzers. Both techniques usually aim to maximize code coverage of the software under test. Test case generation tools include Randoop [306] and EvoSuite [158]. These tools generate unit tests by creating objects and arguments to invoke the methods in a program. Search-based test generation tools, such as EvoSuite [158], uses evolutionary algorithms to evolve test cases over multiple generations. EvoSuite [158] has been known to achieve a high level of code coverage on large benchmarks of real-world programs [311]. Similar to test case generation tools, fuzzers automatically tests software systems. Fuzzers randomly generate inputs to invoke programs. Since its early successes in crashing UNIX utilities, fuzzers have now been applied to a wide range of software systems, including stateful programs [325] and libraries [213, 206, 447, 415, 132]. As these techniques execute the programs, each execution requires some computational power and time. One challenge faced by dynamic analysis tools is the waste of executions as the majority of generated inputs may be invalid, i.e., they fail shallow checks. For example, when fuzzing deep learning libraries, one study reported that a random input generator produces valid inputs only a fifth of the time [421].

2.3 Mining Rules and Specifications

Mining patterns/features. To understand how an API should be used, many studies have proposed methods for mining patterns from publicly available resources such as GitHub and StackOverflow.

Many studies have proposed approaches for mining API usage patterns [456, 262, 414, 413, 291, 303, 61, 449, 381]. Patterns mined from StackOverflow were shown to contain API misuses [449]. Other approaches focus on mining patterns from repositories on GitHub.

Several studies describe how to use their patterns for detecting API misuses in source code [262, 414, 413, 291, 303, 61, 381]. Many of these studies mine frequent patterns and detect misuses from anomalies from frequent patterns in their choice of source code representation. PR-Miner [262] uses frequent itemset mining over the function calls invoked to identify association rules between functions. JADET [414] constructs finite state automatas based on temporal properties over method calls and extracts patterns from them. It detects anomalies that violate the model and heuristically ranks them. Building on JADET, Tikanga [413] converts JADET's models into

formulas in Computational Tree Logic, then use model checking to identify formulae with enough support in a codebase. DMMC [291] detects missing calls on a receiver type, characterising methods by the invocations on each receiver type. Alattin [381] is an approach that mines alternative patterns to detect missing conditional checks. GrouMiner [303] represents programs as groums, which are graphs that encode method calls, field accesses, control, and data-flow. MUDetect [61] encodes methods as API Usage Graphs, building on top of Groums but encoding more relationships between program elements. The above studies proposed unsupervised techniques to mine patterns solely based on their frequency. Apart from mining patterns from GitHub, MUTAPI [416] applies mutation analysis to discover misuse patterns that are mutants of correct usage patterns.

Apart from static analysis, runtime verification can be used to detect violations of API specifications [214]. Runtime verification depends on specifications or patterns that can be both automatically mined [331] or written by hand. However, Legunsen et al. [255] have shown that both manually written specifications and automatically mined specifications have high false positive rates.

Mining models using Active Learning. In active learning [63, 64, 101], a learner sends queries to an oracle (e.g., a human-in-the-loop, or by execution of a program) who responds with some feedback (e.g., the ground truth label of a given data instance). The feedback is used by the learner to refine its model. Active learning has been employed for inferring models of programs. Some studies learns automaton models using active learning, using Angluin's L* algorithm [63] or the QSM technique [139], to build models of software systems [397, 68, 129]. For example, execution traces can be used to construct a specification of a software system [397]. An oracle, which can be a human user, is involved in answering membership queries, i.e. if a sequence of events (e.g. method calls) should be accepted or rejected by the ground-truth state machine representing the software system. Others have used active automata learning for finding differences between programs [68] and finding security flaws in TLS implementations [129]. Researchers have used active learning for the generation of assertions from test suite [324] as well as for regenerating programs using the models to remove undesired behaviors [389, 365].

In classification tasks, active learning is used to query for labels of a small number of informative data instance when labeling every instance is too costly [361]. Data instances are identified, for example, based on the instances that would cause the model to undergo the greatest change if the labels of data instances were known [363]. In software engineering, active learning has been used for the classification of execution traces [92], and for

tasks related to defect and fault prediction [382, 278].

Mining input constraints. To invoke a function in a library, the inputs supplied has to adhere to the API input constraints. To test a library function, randomly generated inputs can be supplied without having knowledge of the input constraints. However, unlike other programs e.g. UNIX utilities [286, 285] that take sequences of bytes as input, many programs, including libraries, accept inputs with both syntactic and semantic pre-conditions [308, 76]. Therefore, the vast majority of random inputs will be invalid and fail to test the core logic of the program.

Some studies have proposed to extract constraints from documentation [87, 170, 377, 421], which enables test generators to be more effective. Specifications generated by JDoctor [87] can be integrated with Randoop [306], while DocTer [421] extracts input constraints for fuzzing deep learning libraries.

Other research infer types [191, 320, 330, 279], which are useful for programs written in dynamically typed languages. In some languages, type annotations can be added to existing code, allowing flexibility for developers, but is challenging and tedious to perform manually. Studies have proposed automated methods of inferring types, often done so probabilistically using deep learning approaches.

Mining rules for monitoring program behaviors. Traditionally, malicious programs have been executed in sandboxes, which blocks access to security-sensitive resources. Researchers [209, 79] suggest that entirely blocking/allowing access to a certain resource may be too coarse-grained, and have proposed to identify more granular conditions of accessing each resource. To do so, techniques have been proposed to automate the mining of rules to be enforced in a sandbox. Jamrozik et al. [209] mines associations between GUI elements and sensitive API access. This allows the identification of rules permitting access to sensitive resources, such as the camera, only if the user is performing specific actions on the app. If the sandbox detects previously unseen sensitive behaviors (e.g. reading a file), the user is alerted. This allows the user to assesses the situation and determine if the new behavior is permitted or stopped.

Existing techniques [252, 209, 398, 79] rely on test generation to explore the possible behaviors for generating rules. These rules can be mined with test case generation using the *test complement exclusion* [438] method. As test case generation do not observe all behaviors, it is possible that behaviors that have not been observed will occur in future. Test complement exclusion uses this as a guarantee by using the sandbox to allow only behaviors seen as the rules were mined. If malicious behavior wasn't observed before, then no malicious behavior can execute.

Existing techniques differ in the context considered to determine if a

given resource should be accessible. If the rule allowing a resource access is too coarse-grained, then it may fail to detect malicious behaviors. If it is too granular, then it may block benign behaviors with inconsequential differences. Wan et al. [398] considers system calls that are called. In Jamrozik et al's work [209], the execution context is the last interacted GUI element before an API call. Le et al's work [252] considers the execution context to be the sequence of other API calls before a given API call.

Mining specifications and behavioural models. Many specification mining algorithms have been proposed. To infer models based on Finite State Automata (FSA), specification mining algorithms abstract over states, and determine if two set of execution traces result in the same state. A classic algorithm that infers a Finite State Automaton from traces is the k-tails algorithm [62]. The k-tails algorithm [62] first uses the input execution traces to build a Prefix Tree Acceptor (PTA). A PTA is a tree-like deterministic finite automaton (DFA) where states are grouped and merged based on the prefix that they share. This automaton is consistent with the input traces and will accept all of them. Next, the algorithm merges states that have the same sequences of invocations in the next k steps.

Building on the k-tails algorithm, Lo et al. [275] propose to mine temporal rules that hold over the input traces and prevent any merge that will result in a violation of the rules. GK-tail [277] mines extended FSA where transitions are labelled not only with method calls, but includes parameter values. They introduce multiple merging criteria, including criteria that do not require exact matches of the transitions, and allow for more general conditions of the parameter values. Krka et al. [242] introduce multiple algorithms in their work, including SEKT, which extends k-tails by adding another condition for equivalence: States are merged only if they correspond to the same abstract state, which are defined by the invariants extracted by Daikon. Le et al. [252] propose a deep-learning based approach, Deep Specification Miner (DSM). After training a Recurrent Neural Network (RNN) on the execution traces, DSM characterizes the states by feature vectors from the RNN.

While variants of the k-tails algorithm combine states based on their prefixes and an equivalence criteria, other approaches have been proposed to determine the states in a model. The CONTRACTOR [128] and CONTRACTOR++ [242], which uses program invariants [146] to characterize model states. ADABU [127] and Tautoko [126] identify inspector methods for each class. Inspector methods are heuristically identified based on their return type (not void), a lack of parameters, and the lack of side-effects. States are characterized by the return values, which are abstracted over to prevent a large number of states, of these inspector methods.

Chapter 3

(Static Analysis + API) Active Learning of Discriminative Subgraph Patterns for API Misuse Detection

3.1 Overview

Developers frequently use Application Programming Interfaces (APIs) incorrectly by violating usage constraints that these APIs may impose [144, 295, 59]. Known as API misuses, the violation of these constraints is a frequent cause of bugs, resulting in software crashes and vulnerabilities.

There have been many techniques proposed to detect misuses of APIs [344]; both static and dynamic analysis have been employed. However, recent studies have shown the ineffectiveness of existing techniques. Manually written specifications and API misuse detectors using static analysis have low precision and recall [255, 60]. From a survey of existing API misuses detectors by Amann et al. [60], most existing detectors mine frequent patterns from existing code. These detectors then look for code that deviates from these patterns, assuming that these deviations are API misuses. Along with recent studies [61, 416, 381], Amann et al. [60] suggested that this naive assumption is the cause of the numerous false positives of existing detectors. There may be uncommon patterns of usage that do not conform to the mined patterns, but do not lead to bugs. Furthermore, there are classes of APIs where their prevalent use are incorrect, such as Java Cryptographic APIs [144, 295, 243, 165]. For these APIs, their correct usage may look like deviants of the most frequent patterns. Due to these reasons, the frequency

of a usage pattern is not a reliable signal of its correctness.

We propose our approach, ALP (Actively Learned Patterns), for detecting Java misuses. While still using frequency information, ALP does not rely on it as the only signal of correctness, and incorporates some human supervision from an experienced user of each API to identify *discriminative* subgraph features, i.e. subgraphs that occur more frequently in graphs of one label than the other. These subgraph patterns act as indicators of either correctness or misuse. Based on the discriminative subgraphs present in each usage example, they are classified with a machine learning classifier.

However, it is impossible to label every usage pattern of an API. Furthermore, similar to findings from prior work [381], we find that API usage is highly imbalanced, with most APIs exhibiting distributions where correct usage examples outnumber misuse examples. Misuses are usually the minority class in this classification-based formulation of the API misuse detection problem. While they are the minority, misuses may lead to bugs and vulnerabilities. As a result, failing to detect them may have severe consequences. A naive sampling of examples to label is likely to pick usage examples containing frequent patterns, while omitting less common but informative patterns that may be discriminative of the minority class. Therefore, key to our approach is our careful use of usage examples from GitHub. ALP uses active learning to identify informative examples for human annotation. This enables ALP to sample a small but informative number of examples for labeling.

Next, to reduce misclassification, ALP withholds judgement on examples it cannot classify with confidence, such as uncommon usage patterns it has not seen, and leaves them for further human inspection. Notice that this is in contrast with existing misuse detectors, which considers usage that deviates from previously seen examples as misuses. Using a novelty detector, ALP withholds judgement on usage instances that it is uncertain of. During practical use, it is plausible for ALP to encounter usage instances of the API that do not resemble any training example. Given the prevalence of correct alternative usage patterns [381], we suggest that the optimal behavior of an API misuse detection tool is to signal that it does not know how to classify such usage instances. By withholding judgement, ALP prevents wasted developer effort investigating false positives.

3.2 Background

We present some background information of ALP. Through examples, we motivate the features of ALP through the challenges of detecting misuses that are tackled by ALP. Afterwards, we describe the API Usage Graph [61],

a graph representation of source code shown to be promising for the API misuse detection problem. We build ALP on top of the API Usage Graph.

3.2.1 Motivating Examples

In this section, we show simplified examples from GitHub projects that show the challenges we attempt to address. Existing misuse detectors usually conflate frequent usage patterns and correct usage patterns, then look for API usage instances that deviate from these patterns. The two primary challenges that we tackle in this work are that there is no known way to reliably and automatically evaluate **the correctness of a mined pattern (Challenge 1)** and **the correctness of a usage example that deviates from known usage patterns (Challenge 2)**.

```
Cipher cipher = Cipher.getInstance("DES");
cipher.init(1, secretKey);
byte[] textBytes = text.getBytes(charset);
byte[] bytes = cipher.doFinal(textBytes);
```

Figure 3.1: A usage pattern involving Cipher. This uses the "DES" algorithm, known to be insecure [144].

The distribution of API usages is highly skewed, and most frequency-based pattern mining approaches assume that the most common patterns are more likely to be correct. However, this is not the case for some APIs, such as cryptographic APIs. An example of an incorrect API usage is shown in Figure 3.1. Hence, the assumption that a highly frequent usage pattern is a correct usage is not a valid assumption, and we address this by including some human supervision. Still, even with a user of the APIs labeling a sample of usage examples, it is challenging to train an effective classifier. Due to the imbalanced distribution of API usage patterns, if one tries to learn usage patterns through a random sampling of examples from GitHub, then it is possible that the sample does not contain a single example using an uncommon pattern. In the case of **Cipher**, one may only encounter examples similar to the above, and miss out the uncommon but correct usage examples.

The second challenge is related to the high rate of false positives. To detect misuses, existing tools look for deviations from mined patterns. Simply reporting any deviation as a misuse will produce numerous false positives, and existing tools attempt to mitigate this problem through various heuristics although the use of heuristics alone are insufficient to solve the problem.

For an example API, we look at `java.util.Map`. Included in Java's standard library, this data structure is ubiquitous in Java projects. Yet, when we inspect the evaluation results of existing detectors on API misuse detection benchmark, MUBench [59, 60], we find that none of the existing misuse detectors are able to detect misuses of `Map`. We hypothesize that this is caused by the variety of usage contexts it appears in. Based on an empirical analysis by sampling API usage instances from GitHub, the idiomatic usage pattern is a null-check performed on the return value of the *get(key)* method call, but there are less common patterns that may not be idiomatic, but are safe and correct.

```
// 1. A pattern that is common and correct
map1 = new HashMap<>();
Integer val1 = map1.get("data");
if (val1 != null) {
  ...
}
// 2. A pattern that is uncommon but correct.
// If reported as a misuse, it would be a false positive
map2 = new ConcurrentHashMap<>();
for (String key : map2.keySet()) {
  int val2 = map2.get(key);
  ...
}

// 3. A misuse of Map.
// If not reported as a misuse,
// it would be a false negative.
// codeMap.get() may return null
((Integer)this.codeMap.get(
  cause.getClass())).intValue()
```

Figure 3.2: Three usages of Map. The first is common and correct, while the second is uncommon but correct. The third is a simplification of a misuse from MUBench.

In the second example shown in Figure 3.2, we see an uncommon usage of a `ConcurrentHashMap`. While it implements the Map interface, `ConcurrentHashMap` does not allow for null values in it. Hence, invoking *get* on keys of the map (by iterating over *keySet*) never returns null. However, based on the frequent pattern of a null-check, this usage instance may be considered a misuse as it deviates from the pattern. Reporting it results in a false positive. Many tools use a ranking-based approach by computing various metrics, such as *rareness* [303], to rank their output with the aim of ranking false positives below true positives.

The third usage is a misuse, in which a method is invoked on the return value of *get* without checking for null, but is not detected by existing tech-

niques. In particular, while it is a deviation from the null-check pattern, MUDetectXP does not report such usage instances as it heuristically omits fields from its findings. This restriction was required to prevent numerous false positives related to the usage of fields. While this heuristic succeeds in reducing the number of false positives, it may cause MUDetectXP to miss misuses.

The poor empirical performance [60] of these tools suggest the need for a different approach to reduce false positives. We suggest that, despite some success in using heuristics to prevent the reporting of false positives, the fundamental problem of distinguishing real usage constraints from spurious usage patterns remains unsolved.

We also address a third challenge, which is that **API usage patterns can be complex (Challenge 3)**. Some studies model method calls and their order of invocation, data-flow, and control-flow [414, 413], other studies has suggested the need for modelling multiple objects [331], static methods and constants [454], self-usages [61] (the usage of an API within its own implementation), inheritance [454], argument values [144, 455], and synchronization [266, 61]. Consequently, the representation of a usage pattern has to be sufficiently rich to capture this complexity.

```
1  class CopyOnWriteMap<K,V> implements ConcurrentMap<K,V> {
2    ...
3    public V putIfAbsent(K k, V v) {
4      synchronized(this) {
5        if (!containsKey(k))
6          return put(k, v);
7        else
8          return get(k);
9      }
10   }
11   ...
12   public V get(K k) {
13     this.internalMap.get(k);
14   }
15 }
```

Figure 3.3: Example usage of Map in a CopyOnWriteMap, which transitively implements the Map interface

In Figure 3.3, we show an example of the complex relationships between program elements related to usage of an API. We see uses of `Map` that have relationships with program elements beyond control-flow. The class, CopyOnWriteMap, is a sub-type of `Map` through the `ConcurrentMap` interface. The *get(key)* on line 8 is a self-usage, invoking *get(key)* which is overridden by the class (line 12). The object, itself a `Map`, is used to synchronize access to the other method calls on line 4.

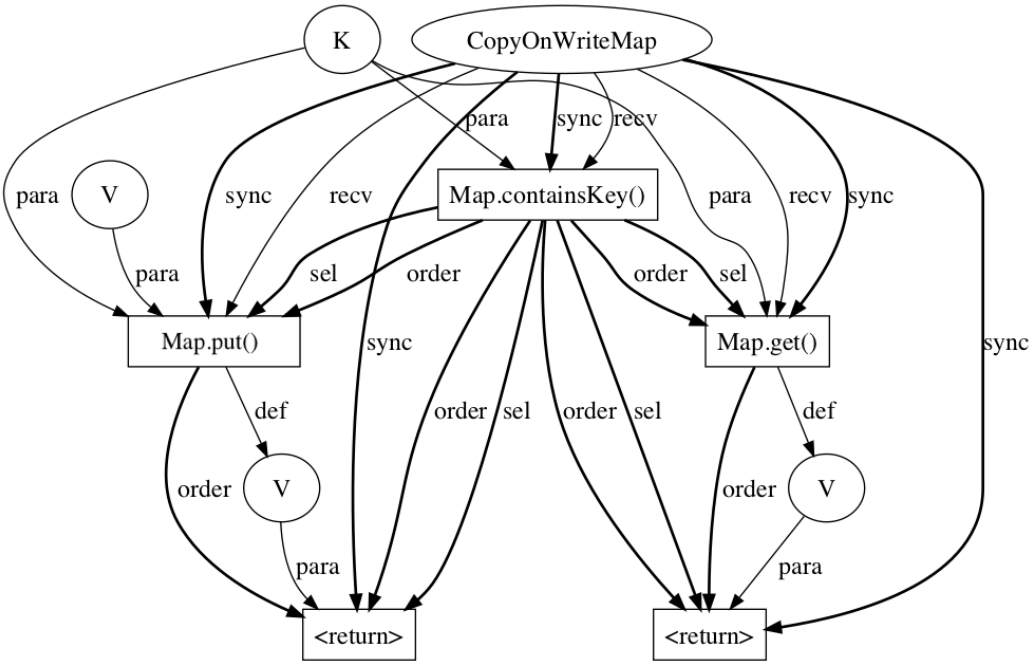


Figure 3.4: Example of an API Usage Graph

3.2.2 API Usage Graph

As prior work [60, 303, 61, 454] have shown the promise of using a graph representation of API usage for detecting API misuses, our work represents programs as graphs. Amann et al. [61] proposed the API Usage Graph. An example is shown in Figure 3.4, constructed based on the *putIfAbsent* method in Figure 3.3. The API Usage Graph is a directed multi-graph with labeled nodes and edges. Nodes represent objects, values, method invocations, constructor calls, field accesses, and conditional checks. Edges represent data and control flow between the program elements of the nodes. Edges have the following labels:

- `recv` (linking method calls that are invoked on a object),
- `param` (linking variables/literals used as arguments to the methods called. While labeled "param", a more accurate label may be "arg" as this links variables/literals used as arguments to method call.),
- `def` (linking actions creating/returning a data),
- `order` (linking actions in order of execution),
- `sel` (for control-flow; representing a control-relationship),

- `sync` (linking actions to objects they are synchronized on),
- `throw` and `handle` (for exceptional flow).

Amann et al. [61] mined subgraphs of the API Usage Graph. Alternatives to using the API Usage Graph are to use patterns mined from other representations of a program. For example, Grouminer [303] considers control and dataflow dependencies, while DMMC [291] encodes the set of methods called on an object. Still, the expressivity of the API Usage Graph allows it to distinguish more types of misuses from correct usage. While previously proposed detectors have modelled some of these relationships, they were not considered in tandem. On the other hand, these relationships are combined in the API Usage Graph [61]. Therefore, patterns mined in these prior studies [303, 291] can also be represented as a subgraph pattern in the API Usage Graphs Frequent subgraphs mined from API usage graphs was shown to be able to capture usage constraints of APIs in previous work [61]. Therefore, we express programs as API Usage Graphs in our work and mine subgraphs from them.

Discriminative Subgraph Mining Many existing approaches focus on mining frequent usage patterns of source code using an API. Frequent subgraphs are discovered by identifying subgraphs occuring more than a user-specified number of times in a collection of graphs. Most of these frequent subgraphs will not be discriminative of API misuses, in other words, they may appear equally frequently in both correct and incorrect API usages. Moreover, there is usually a large number of subgraphs with frequency greater than the user-specified number of times. The large volume of frequent subgraphs may make further processing of the subgraphs unscalable.

One solution to the limitations of frequent pattern mining is to mine discriminative subgraphs. Used for graph classification tasks where graphs have different labels, these subgraphs are both frequent and have discriminative power to distinguish between graphs of different labels. In our study, we mine discriminative subgraphs to distinguish API usage graphs that are misuses from API usage graphs that are correct. In other words, we look only for frequent subgraphs that are indicative of either correct or incorrect API usage. While all discriminative subgraphs are frequent, not all frequent subgraphs are discriminative.

Graphs can be represented by the subgraphs that they contain. To use discriminative subgraphs to perform classification, each usage location is expressed in terms of the presence and absence of these subgraphs. A vector of the same length of the number of discriminative subgraphs, containing 0s and 1s representing absence and presence respectively, is passed into a machine learning classifier. For example, the source code given in Figure 3.3 could

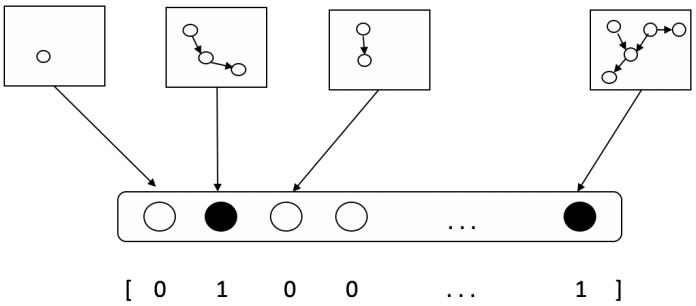


Figure 3.5: Vector representation of a usage example. The i'th element in the vector is 1 if the i'th discriminative subgraph is present in the usage example. Given that the second element represents a check for `containsKey`, and the program in Figure 3.3 contains a checks for `containsKey`, the second element is 1.

be represented as a vector in Figure 3.5. A single usage example may match multiple subgraph features (e.g. both the second and last subgraph features in Figure 3.5 are matched). The other discriminative subgraphs that are not present in the API usage location have a value of 0 in the vector. Representing the usage in this form allows the use of machine learning classifiers which take vectors as input.

The task is of a probabilistic nature; while it is typically the case that *containsKey* followed by *get* is a correct usage, this usage pattern does not guarantee that the usage is correct. Some implementations of `java.util.Map` may allow a `null` value. As such, an approach that directly matches programs to a singular pattern cannot capture this uncertainty while the machine learning classifier-based approach of ALP reflects the nature of this task.

3.2.3 Active Learning

Active learning is a subfield of machine learning that aims to achieve better effectiveness while requiring fewer labelled examples [125, 260, 259, 122, 162]. Many machine learning techniques require many labeled examples, which are examples where their true class has been indicated by a human annotator, to train a model. Active learning techniques can be useful in situations where labels of examples are hard to obtain. Rather than selecting random examples to be labelled, active learning aims to select informative and representative examples, and has been shown to be effective in minimizing the number of labels required to learn an effective classifier. Approaches using

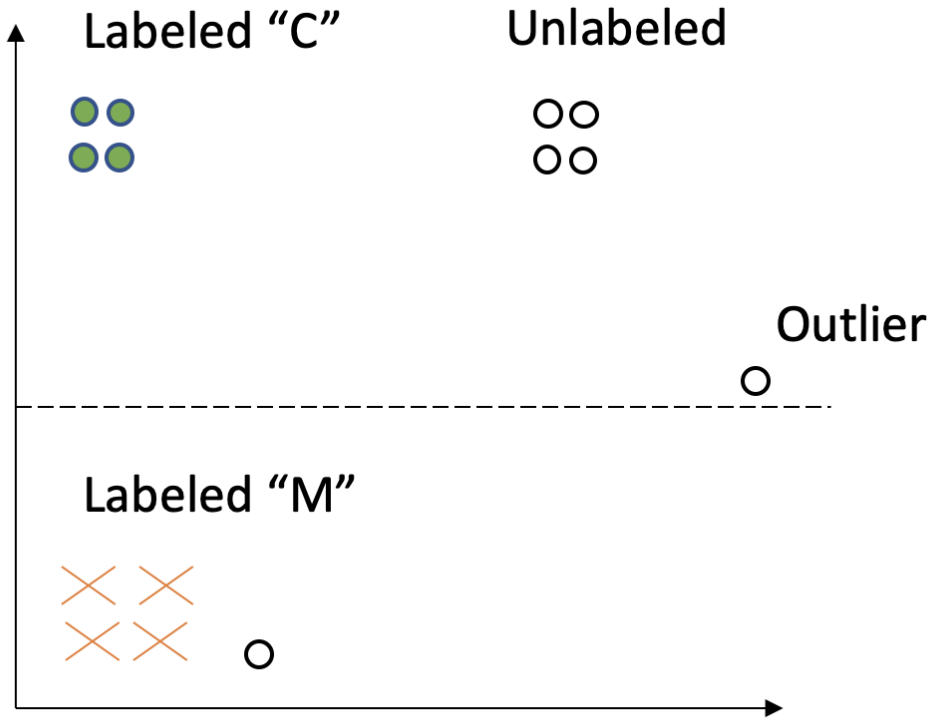


Figure 3.6: Data points projected onto the input feature space. Each point represents one example. The dashed line represented the decision boundary of the model, which learned to distinguish between examples of two different labels. The figure indicates two groups of examples that have been labeled. The unfilled circles indicate examples that have not been labeled. An active learner may select an example from Unlabeled for labeling.

active learning ask *queries* that are answered by an *oracle* (usually human annotators). These queries are unlabelled examples; their true classes are not known yet as they have not been labeled by the human annotator. In the context of graph classification, an active learning technique poses *query graphs*, unlabeled graphs, to the human annotator.

Many frameworks for querying unlabelled examples have been proposed [125, 260, 259, 122, 162]. Many of these strategies direct the human annotator's attention at *informative* examples, such as examples where the model produces the most uncertain predictions [125, 260, 259], examples that would cause the model to change the most [362], or examples that will reduce the variance of a model [122]. In Figure 3.6, points with similar features as the

already labeled examples (and are close to them in the input space) are less likely to be selected, since they likely share the same label as the labeled examples. An example from the **Unlabeled** region is likely to be selected as it is far from the already labeled examples. Another direction of research measures the *representativeness* of unlabelled examples, addressing limitations of prior techniques that tend to select outliers for labeling (e.g. selecting the outlier in Figure 3.6 may not give us information about the other points in the **Unlabeled** cluster). These techniques, e.g. [362, 162], favour examples that are in dense regions of the input space, or are most similar to other unlabelled examples (e.g. examples from **Unlabeled** in Figure 3.6) . In short, active learning tries to minimize the total number of labels required by selecting query examples that are informative and representative. An informative example is one that should be dissimilar to examples that are already labelled, while a representative example is one that is similar to other unlabelled examples.

In ALP, we leverage active learning to iteratively identify examples from GitHub to be labelled by the human annotator. We describe in Section 3.3.6 how we determine unlabelled usage examples that are dissimilar from already labelled examples, while ensuring that they are similar to other unlabelled usage examples.

To motivate this, we use Figure 3.1 as an example. If ALP has already obtained enough examples to mine a feature indicating that *getInstance("DES")* is an incorrect use, ALP will focus more on examples that do not contain *getInstance("DES")* (hence, dissimilar to existing labeled examples). However, of the different usage examples, it will be helpful to focus on examples with usage patterns that do not appear by coincidence and are not outliers (in other words, examples which are similar to other unlabelled examples). These heuristics would help ALP to locate examples that use other types of algorithm (e.g. *"AES"*).

3.2.4 Learning with Rejection and Novelty Detection

Researchers have proposed a framework for classification with a reject option [117, 193, 124]. This framework consists of two components: a traditional classifier and a rejection function. The goal is a machine learning model that knows what it does not know. It is suggested that this framework is useful for scenarios where incorrect predictions can be costly, such as medical diagnosis [193]. Typically, approaches that incorporate learning with rejection estimate the confidence of a prediction. If less confident about a prediction, the model withholds its prediction.

Rejecting a classification and task of novelty detection are closely re-

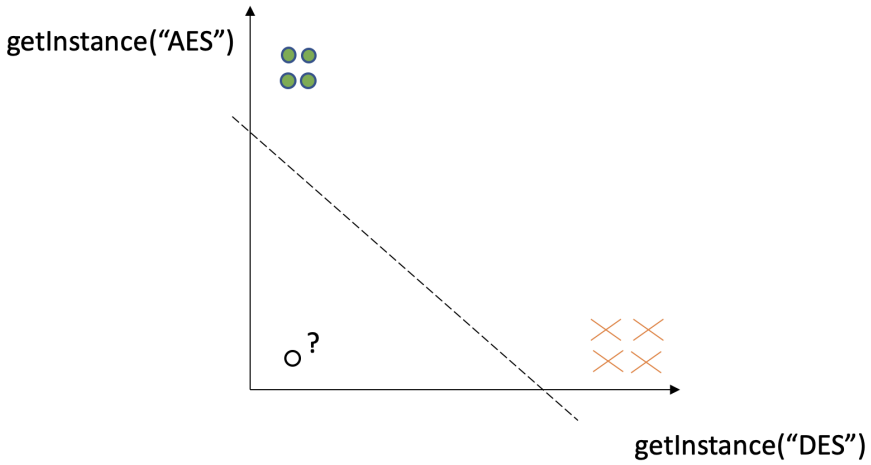


Figure 3.7: Examples projected onto the input feature space. Given that two features related to “DES” and “AES” have been identified, a novelty detector detects if a test instance (shown with a question mark, “?”) uses an algorithm it has not seen (e.g. “RSA”). This allows ALP to make the right choice to reject the classification as it has no way to make the right prediction.

lated [378]; a classification should be rejected if the object under classification is an outlier [378], and novelty detection is the task of identifying that some test example does not resemble the examples used during training. While modern machine learning techniques learn from large amounts of data, it is still possible that a large number of examples that exist in the real world are not reflected within the dataset. Solutions to this problem learn a model of “normal” examples seen during training, and use the model to detect “abnormal” examples during testing. One-class classification can be used to approach this problem [166, 321], in which a model learns to differentiate the “normal” class seen during training from all other examples [230].

In ALP, we use a classifier with a reject option, in which we use an off-the-shelf novelty detector (later described in Section 3.3.7) as a measure of the confidence of a prediction. If the novelty detector determines that the example is novel, then it rejects the classification and does not make a prediction. The novelty detector compares a test instance to its neighbours to determine how isolated the instance is; the more isolated and further it is from its neighborhood, the more novel it is.

In the example of `Cipher`, if ALP has only seen examples that allowed it to mine some features for a small set of algorithms (e.g. `getInstance("DES")`

and `getInstance("AES")`), it will not be able to correctly judge the correctness of a previously unseen algorithm. When faced with a test instance that use a new algorithm, e.g. `getInstance("RSA")`, as shown in Figure 3.7 as the question mark ("?"), the novelty detector allows ALP to detect that it is a an out-of-distribution instance as it does not resemble any of the training examples, all of which have one of the two subgraph features.

3.3 The ALP Approach

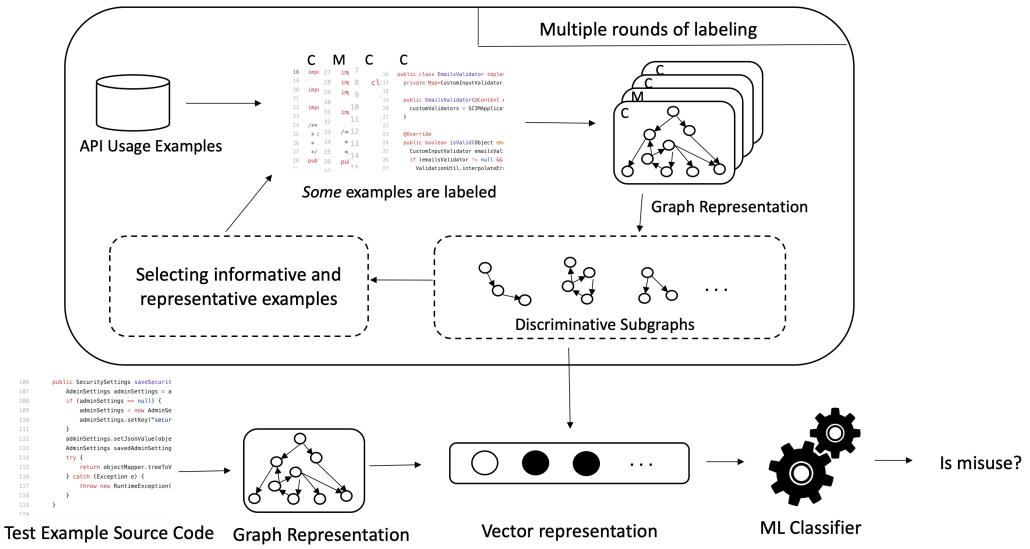


Figure 3.8: High-level overview of ALP. ALP's objective is to classify API usages. A user labels a small sample of usage examples, categorizing them as misuses or correct uses. Discriminative subgraphs, which are indicative of either correct uses or misuses, are identified from these labeled examples. ALP may identify more samples of informative and representative code examples for labeling. Based on the discriminative subgraphs, a given API usage can be represented as a vector, and then input to a Machine Learning classifier to determine if it is a misuse.

3.3.1 High-level Overview

In ALP, the API misuse detection task is reformulated as a graph classification task. In this formulation of the problem, usage sites of an API (methods

using an API) can be classified as either a misuse or a correct usage. Summarized in Figure 3.8, ALP relies on a human annotator to label a small set of selected examples as correct and misuse examples. Next, it performs discriminative subgraph mining while picking more examples for the human annotator to label based on principles of Active Learning. Finally, a machine learning classifier is trained together with a novelty detector.

We represent each method with a usage of the API as a graph, G. ALP extends the API Usage Graph to represent source code (Section 3.3.3). Examples of API usage are mined from GitHub (Section 3.3.4) and these examples are transformed into graphs. In order to train and use a machine learning classifier, each graph is represented as a feature vector, v. To this end, we identify and use discriminative subgraphs features (Section 3.3.5).

The difficulty of addressing **Challenge 1** using a supervised approach is the need to minimize human effort while labeling examples that are less common but informative. It is not possible for a human to label every possible usage pattern. ALP directs human attention to examples that differ from already labeled examples to maximise the diversity of labeled examples, addressing **Challenges 1 and 2**. This is enabled by an active-learning inspired approach (Section 3.3.6), in which ALP iteratively queries the human annotator for the labels of a small but informative batch of examples.

Finally, we perform classification using a machine learning classifier (Section 3.3.7). We utilize reject option in classification [117, 193] for handling usage instances with uncertain labels. This helps addresses **Challenge 2** during testing time, detecting if the usage instance is too abnormal from training usage examples. When faced with an abnormal example, ALP does not consider it a misuse, but defers judgement. The classification module outputs one of three labels, Rejection (signalling uncertainty), Misuse, or Correct.

ALP represents programs using the API Usage Graphs (see Section 3.2.2). This enables ALP to mine subgraph features that are complex, consisting of multiple types of information (e.g. both constraints on parameter values and control-flow can be included within a single pattern). To tackle **Challenge 3**, we address several limitations of the API Usage Graph by extending it to include richer information about the program. In this work, we refer to our extension as the EAUG (Extended API Usage Graph).

To sum up, ALP consists of the following components:

- **Extensions to the API Usage Graph (Extended API Usage Graph)**. This allows for the subgraph mining process to mine features that are key to avoid false positives.

- **A wrapper over GitHub's search API**. This is used to obtain usage

examples.

- **A loop over the human annotating the examples, the discriminative subgraph miner, and the selector of examples to label.** This is the only component requiring (potentially multiple iterations of) human input.

- **The graph classifier.** Based on the features mined by the discriminative subgraph miner, the classifier is tuned over the training dataset. It outputs one of three labels (**C**orrect, **M**isuse, and **U**nknown) for an unlabelled instance during testing time.

3.3.2 Workflow of ALP

ALP is intended for use by an experienced user of a given API. We assume that these users are knowledgeable about the use of the API, and can accurately label the correctness of their usages with relatively low effort. ALP enables a user to identify potential problems without writing a specification by hand. Some studies have shown that manually written specifications may introduce many false positives as they fail to account for all usage patterns used by API clients [255]. Other studies also suggest that developers face many challenges writing specifications [108, 358].

In this work, the first author studied the APIs carefully before labeling the data. We envision that our work may be useful for an API developer to locate all projects and locations where the API may be used incorrectly, such as if a vulnerability related to the API has been reported, or if the API developer plans to introduce a breaking change that may affect clients that have used the API in a way not expected by the API developer [18] (e.g. due to API workarounds [246]). There is often a knowledge gap between API developers and API users [246, 343], and it is possible that users may use an API in ways that violate undocumented usage constraints (i.e., misuses) . For such cases, a modification to the API by the developer may break the client projects [290, 155]. Considering the latter usage scenario (developer introduces breaking changes), the following illustrates the benefit of ALP:

Without ALP, the API developer will have to search for all usages of the API and manually inspect them to determine if it is a misuse. The API developer may try to filter the usages, but ultimately will find it difficult to perform a search that checks if a given constraint holds and to enumerate over all the valid usage scenarios. The developer may also miss out uncommon usage patterns. This leads to loss of time and increases the cost of maintenance when the API developer wishes to introduce breaking changes.

With ALP, the API developer only has to label a small number of usage examples, and ALP can classify the locations that are likely to be misuses, acting as a filter for usages that the developer has to inspect. Moreover, once ALP has been trained, it can be used again without any cost. The developer only needs to inspect the few locations that ALP reports misuses in, and can quickly reach out to the developers of the client projects. As a result, the developer can save time and effort while detecting potential problems.

The workflow of using ALP is given as follows:

- First, the type of the API is given as input, then ALP randomly selects a small number of examples for the user to label.
- Next, the first batch of inputs is input to ALP, which may query the user for the labels to another batch of examples. This step is repeated until the stopping criteria (e.g. a maximum of 5% of the dataset are labelled, or if ALP has found enough discriminative subgraphs for 95% of the dataset) is met. After this step, no further data is labeled.
- ALP uses the labeled examples as input to construct a machine learning model of the API. Both the graph classifier and novelty detector are trained in this step.
- After ALP has a trained model of the API, it can accept test instances as input. Given a file with source code using an API, ALP converts each method using the API into a vector. For each instance, the model produces one of three labels, **C**orrect, **M**isuses, or **U**nknown.

Within this study, we limit ourselves to look for misuses in MUBench and AU500. To evaluate the generalizibility of the approach, we do not label usage examples from projects that are present in MUBench and the AU500. The usages in MUBench and AU500 are used only as test instances.

3.3.3 Extended API Usage Graph

To mine discriminative subgraphs, we represent an API usage as a graph, and we propose extensions to the API Usage Graph (AUG). While the AUG can represent many important aspects of an API usage, we find that it considers only information of program elements directly related to the execution of the program. We hypothesize that other elements in the source code, which may not directly influence a program's execution, may serve to inform developers of their purpose and can act as indicators of the different contexts of an API usage. Extending the AUG with these indicators may allow subgraphs that are more discriminative to be mined. In this subsection, we motivate

the choices we make for extending the AUG using concrete examples from GitHub.

```
1  void sendNck(String protocol, PrintWriter printWriter,
2              String result) {
3    printWriter.write(protocol + TOKEN_DIVISION +
4                      result + "\r\n");
5    printWriter.flush();
6  }
7
8  void quit(String nickName, PrintWriter printWriter,
9           BufferedReader bufferReader, Socket socket) {
10   ... // removed irrelevant code
11   printWriter.close();
12   ... // removed irrelevant code
13 }
```

Figure 3.9: Simplified example usage of `PrintWriter`. The `PrintWriter`'s use is spread across multiple methods.

Parameters and fields. The AUG does not distinguish between variables that are instantiated locally in the scope of the current method, the fields of the class, and the parameters of the current method. We suggest that having the ability to distinguish between them will help to further distinguish between different usage contexts, including self-usages [60] where the implementation of an API calls itself. In practice, developers may use fields and parameters differently from local variables. Fields have a wider range of usage compared to local variables; the usage of fields may extend beyond a single method and may hold the result of partial computation, may be used for synchronization, or for caching. Empirically, we found that the use of some APIs are not self-contained within a single call site, e.g. many usages of `PrintWriter` use it as a field in an enclosing class. While the correct use of a `PrintWriter` requires *close* to be invoked at the end of its lifecycle, its correct use may be spread across multiple methods of the enclosing class. Hence, a misuse only occurs if a usage constraint of the enclosing class is, itself, violated. Another frequently observed pattern (e.g. in Figure 3.9) is that a `PrintWriter` is passed as an argument to the method, and it would not be a misuse if the `PrintWriter` wasn't closed as the client of the API that passed in the `PrintWriter` should close it from the outside of the API. On the other hand, had the `PrintWriter` been instantiated as a local variable, then failing to invoke *close* on it will cause a resource leak. Note that while the AUG has an edge type (described above as the `param` edge) to indicate that a particular variable is passed as an argument in a method call, the AUG does not indicate the parameters of the method represented by the AUG.

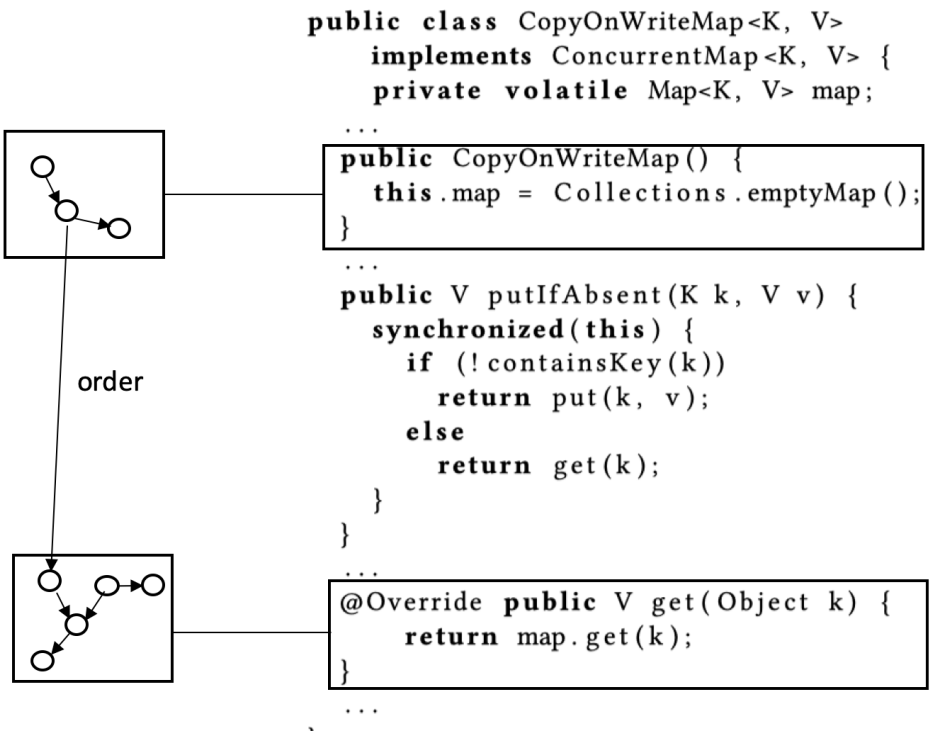


Figure 3.10: When a field is used in a method, the initialization of the field is linked to the method through an `order` edge.

In the EAUG, we indicate fields and parameters differently from local variables, indicating this piece of information in the data node. Similar to fields, method parameters may have implicit usage constraints on them and may follow specific rules and patterns [455, 444].

Constructors and field initializations. We observe that constructors and field initializations provide essential information to identify misuses. For example, some fields are immutable (e.g. the name of a cryptographic algorithm, such as "DES" and "AES", as a string constant, and this information is essential to identifying API misuses related to cryptography), such as the example in Figure 3.11. Once assigned a value, they are never reassigned and this value is critical in deciding if the use of an API is a misuse. To include such information, we find statements from blocks of code containing the initialization of fields (e.g. constructors, field initialization, and initialization blocks), linking them to the graph of methods that use these fields. As the execution of the initializations and constructor must occur before the execution of other methods, they are joined by `order` edges in the graph, where the initializations are ordered before the method invocations. An example is

```
1  // a constant with a choice of algorithm
2  private static String DES_CBC = "DES/CBC/PKCS7Padding";
3
4  byte[] process(
5    byte[] src, byte[] key, byte[] iv) {
6    ...
7    Cipher cipher = Cipher.getInstance(DES_CBC);
8    ...
9  }
```

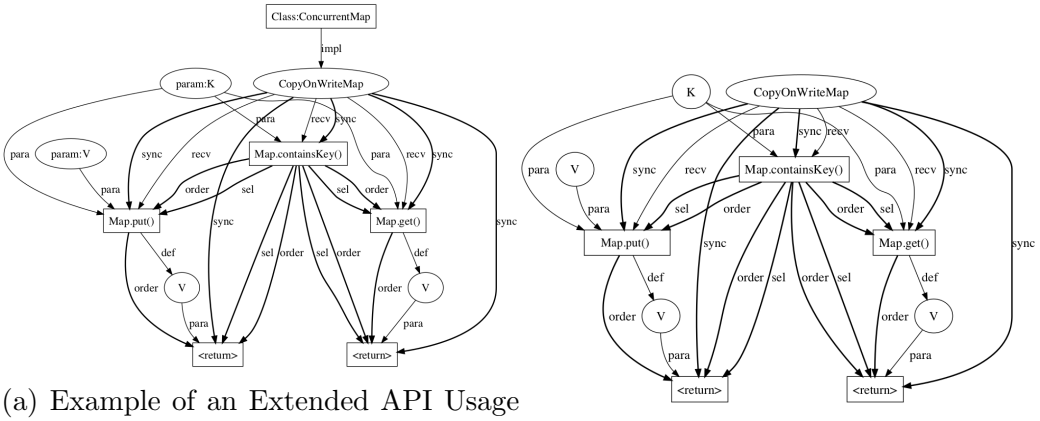
Figure 3.11: Simplified example usage of a `Cipher`. The `Cipher` is initialized using the value of a field.

shown in Figure 3.10.

Subtyping and inheritance. The self-usage of members of an API often imply different usage constraints [61], for example, one source of false positives was related to self-usage, in which a class invokes its own API in its own implementation [61] as API usage within its own implementation may deviate from common usage patterns [61]. For example, a class implementing the `Iterator` interface may internally call its *next()* method without a check of the *hasNext()* method [60]. Usage constraints expected of clients, e.g. guarding method calls or checking a return value, may not be necessary for self-usages. Other studies also found that API clients may extend the API through inheritance [454]. We try to detect this context by including information about subtyping and inheritance in the Extended AUG (EAUG); if a class implements a given interface, a correct usage of the API may resemble other usages that implements the same interface. Thus, the EAUG can model self-usages by generalizing it as a usage contextualized by a particular interface.

We encode such information by including the superclass/interface as a data node linked to the method entry. Concretely, this allows ALP to extract this information as a subgraph of a single node during the discriminative subgraph mining process. In the vector representation of the usage example, it may not be directly useful on its own, but it may provide useful information in conjunction with the other subgraph features for the classifier to make a decision.

Comparison with AUG In Figure 3.12, we provide a visualization of the EAUG of the example shown earlier in Figure 3.3, which is represented as an AUG shown ealier in Figure 3.4. In the EAUG, the interface, `ConcurrentMap`, implemented by the `CopyOnWriteMap` is captured in a node. Moreover, the type of the variable is suffixed with "param:" if it was passed into the method from the calling site as an argument. While this example does not use a field,



(a) Example of an Extended API Usage Graph

(b) Example of an API Usage Graph

Figure 3.12: Example of an Extended API Usage Graph, shown in (a). The corresponding API Usage Graph, previously presented in Figure 3.4, of the same program is shown in (b)

the type of a field would be similarly prefixed with "field:". This allows ALP to distinguish between fields, parameters, and locally instantiated variables.

3.3.4 Mining GitHub

To learn subgraph features, we first require a set of usage examples of a given API. We take advantage of the numerous usage examples of an API on GitHub. The examples from GitHub are later labeled by the human annotator and this data is used as the training dataset.

We mine GitHub for usage examples by adapting a tool [71] that wraps over GitHub's search API. Types are resolved on a best-effort basis by downloading the latest version of the libraries whenever requiring third-party dependencies. If type resolution fails, we discard the usage example.

As the majority of files on GitHub are clones [276], we perform a file-level code clone de-duplication to avoid wasted effort and space storing redundant information. We use a token-based algorithm similar to SourcererCC [356]. Much of SourcererCC focuses on scalability, which we did not need in our work and we used only its code comparison algorithm. Next, methods using the API are identified and a method-level de-duplication is used to further trim the dataset. Hence, only unique methods will be labeled by a human annotator.

To detect code clones, we use a token-based approach [356]. We consider a block of code as a bag of tokens. The similarity measure, $O\left(B_x, B_y\right)$, between two code blocks, $O(B_x, B_y)$, is computed based on the number of

tokens shared by the code blocks, it is given as follows:

$$O\left(B_x, B_y\right) = |B_x \cap B_y|$$

If $O\left(B_x, B_y\right)$ is greater than 0.7, then we consider the code blocks as clones. As earlier described, we perform code clone deduplication at the method-level, and the human annotator does not waste any effort in labeling a code clone of an earlier labeled method.

On average, we find that there are about 7 code clones (explained) for a file containing an API usage. 37% of the files have at least one clone, and the most frequently cloned file had over 1600 code clones. This is in line with previous studies [276, 169], which found a large number of copy-and-pasted code on GitHub.

3.3.5 Discriminative subgraph mining

Unlike typical machine learning applications, graphs cannot be immediately encoded in a vector space, which is required to pass the graph as input to a machine learning classifier. One method of encoding a graph in a vector space is to run a subgraph mining algorithm and identify the best subgraph features. The graph is represented as a vector indicating if these subgraphs are contained in it. Then, we mine subgraphs that are *discriminative*, i.e. occurring more frequently in graphs of one label than the other. Rather than using frequent subgraphs, we posit that discriminative subgraphs are more likely to better represent usage constraints.

In Figure 3.13, we illustrate the discriminative subgraph mining process using the running example. Given some examples that have been labeled, ALP mines subgraphs from them. Among the examples containing a null-check following a *get* method call, there are significantly more examples that are labeled by the human annotator as correct usages than incorrect usages. Thus, the subgraph shown in Figure 3.14, common to the three examples in Figure 3.13, is identified as a discriminative subgraph feature.

In this study, discriminative subgraphs are identified by two criteria. As ALP enumerates through frequent subgraphs, a test of statistical significance is performed to filter out insignificant subgraphs. At the end of the frequent subgraph mining process, a second round of filtering is done using the CORK criterion [379], which we use to remove subgraph features that do not contribute to improving classification.

Enumerating frequent subgraphs Enumerating frequent subgraphs allows us to skip the consideration of subgraphs with support below a threshold, *min_sup*. However, identifying frequent subgraphs is computationally

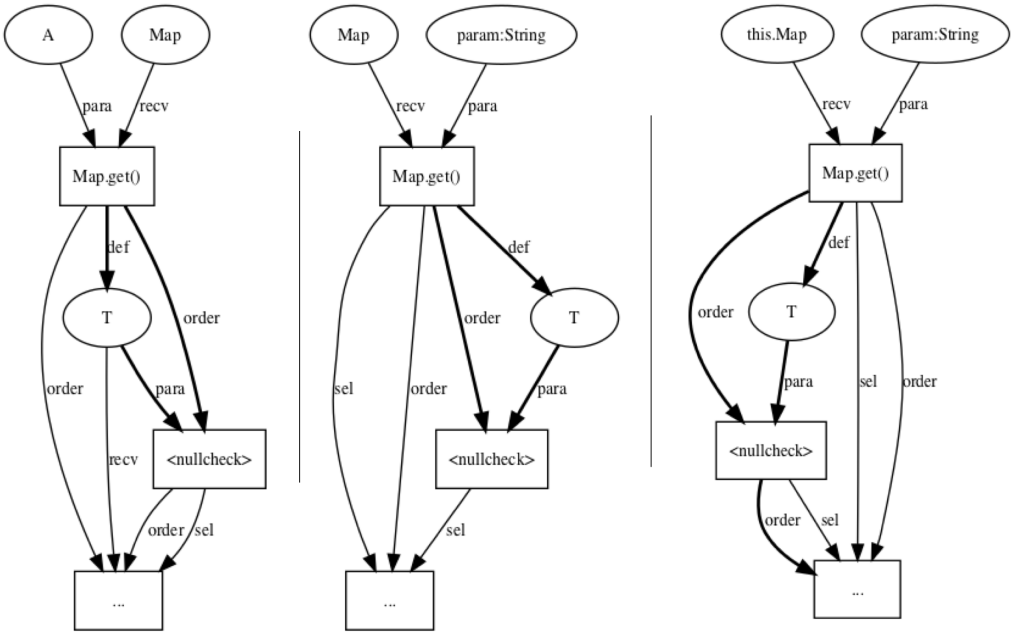
```
T bs = relation.get(a);

if (bs == null) {
    ...
```

```
T data = cache.get(key);
if (data == null)
{
        ...
```

```
T value = PropMap.get(key);
if (value == null) {
    ...
```

(a) Fragments of three examples using `java.util.Map`



(b) Parts of the three examples represented as EAUGs

Figure 3.13: Simplified examples of three usages of `java.util.Map`, shown in (a), and parts of their EAUG representation, shown in (b)

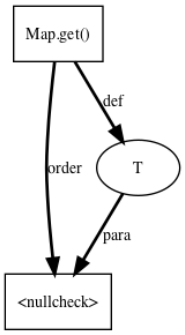


Figure 3.14: Example discriminative subgraph mined from the examples in Figure 3.13 after labeling. This discriminative subgraph represents a null-check following `get`

costly due to the cost of checking for subgraph isomorphism, which determines if a graph contains a particular subgraph. This check is known to be NP-complete. To enumerate frequent subgraphs quickly, we leverage the frequent subgraph mining algorithm, gSpan [427], which is well-understood and efficient.

gSpan takes a collection of graphs and *min_sup* as input, identifying subgraphs with frequency above *min_sup* as output. To efficiently enumerate the frequent subgraphs, gSpan maps each subgraph to a canonical representation, a minimum DFS code. Through a depth-first search, gSpan enumerates subgraphs in their DFS code order. The order of subgraphs visited can be viewed as a traversal of a DFS code tree. When a subgraph with a non-minimum DFS code is reached, it is directly pruned from the code tree. Using this strategy, gSpan visits subgraphs in the canonical search space, without the computationally expensive test for subgraph isomorphism. In this work, we set *min_sup* to a small value, 3. In practice, the threshold required for a subgraph to be discriminative predominantly depends on the Chi-Square test of independence if the choice of the *min_sup* parameter is small.

Testing for significance A frequent subgraph may occur commonly among the usage examples of an API, but have no implication on the correctness of the usage. For example, a frequent subgraph of `java.util.Map` may capture the common use of *System.out.println* (as shown in Figure 3.15), which is unrelated to using `java.util.Map`, while another frequent subgraph captures the usage pattern of a null-check following *get*, a true usage constraint (previously shown in Figure 3.14). Our objective is to filter away the subgraph shown in Figure 3.15, but keep the subgraph shown in Figure 3.14. Therefore, when enumerating the frequent subgraphs, we perform a second

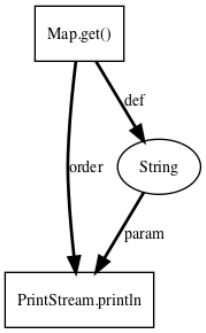


Figure 3.15: Simplified example of a frequent subgraph that is *not* discriminative, representing usages printing to standard output. ALP enumerates such subgraphs, but they fail to pass the checks used to identify discriminative subgraphs.

check that the frequent subgraph has some discriminative power. In the case of `java.util.Map`, the subgraph that captures the null-check would be discriminative, while the subgraph that encodes printing to *System.out* would be filtered out.

Next, we try to find subgraphs that appear more frequently among one label (C or M). To determine if a subgraph appears significantly more in graphs of one label than the other, we compute the support of each subgraph. We count four quantities. They are C_H, the support (number of 'hits') of the subgraph among examples labeled C, examples of correct usage, C_M, the number of graphs labeled C that does not contain the subgraph ('misses'), and M_H and M_M, similarly defined for the examples labeled M, the examples of misuses. Then, we use a Chi-Square test of independence to determine if the difference in the number of occurrences of a subgraph in the two set C and M is more often than expected by chance, using a significance level of 0.05 as a threshold. The Chi-Square statistic measures the statistical significance of a pattern, and is calculated as follows:

$$\chi^2 = \sum_{i=\{C,M\}} \sum_{j=\{H,M\}} \frac{\left(o_{i_j} - e_{i_j}\right)^2}{e_{i_j}} \tag{3.1}$$

o_{i_j} is the support observed of C_H, C_M, M_H, and M_M. e_{i_j} is the expected support, given the null hypothesis that the subgraph is not discriminative of either label. Under the null hypothesis, the subgraph is expected to appear a similar number of times in graphs of both labels. The Chi-Square statistic is, therefore, a measure that the difference in the number of observations did not occur by chance. For the null hypothesis to be rejected, the subgraph should

appear significantly more frequently in graphs of one label. The larger χ^2 is, the more probable there is a relationship between the subgraph and the label. Subgraphs that are not discriminative with respect to the two label are filtered out.

CORK scoring criterion Finally, we use the CORK scoring criterion [379]. While the subgraphs remaining after the previous step are discriminative, the subgraphs may not be independent and may frequently co-occur with one another. The CORK criterion allows us to address this by discarding subgraphs that do not contribute to better classification. CORK counts the number of *correspondences*, which are pairs of misuse and correct examples that cannot be disambiguated from each other given the selected set of features. A high correspondence indicates that the selected features lack discriminative power and more features should be selected. CORK is defined by the following equation, where v is the example represented as a feature vector, and f is a feature in the set of selected features, F. Given a pair of labeled examples i and j, i and j correspond with each other if they have different labels but their feature vectors, v, constructed based on the currently selected features, are identical.

$$\begin{aligned} correspondence(i,j) \Leftrightarrow \left(v^{(i)} \in C\right) \wedge \left(v^{(j)} \in M\right) \\ \wedge \forall f \in F \left(v_f^{(i)} = v_f^{(j)}\right) \end{aligned} \tag{3.2}$$

Initially, F is empty. We iterate through the significant subgraphs in the decreasing order of their coverage of the unlabeled dataset, considering one subgraph at a time. A subgraph feature is added to F only if it improves the CORK score (i.e. only if it contributes to disambiguating at least one pair of misuse/correct example). After this step, we have obtained a set of discriminative subgraphs which will be used to construct the feature vector of a usage site.

Branch-and-prune Mining subgraphs is known to be computationally intensive. For mining subgraphs in a shorter time, researchers have proposed various methods to speed up the search. Such approaches may involve pruning the search space. For example, gSpan employs a heuristic to prune branches during the traversal of the code tree. Branches corresponding to supergraphs are pruned if a subgraph has frequency below the minimum support, *min_sup*, as any supergraph has a lower frequency than its subgraph. Similar to existing subgraph mining algorithms [379], ALP's extends the branch-and-prune approach in the canonical search space to speed up subgraph mining. As our focus is to identify subgraphs that occur more significantly for one label, we compute the upper-bound of significance that a

supergraph can have in this particular branch. For any subgraph, the best case is that one of C_H or M_H is maximized while the other is 0. Within the code tree, as we traverse from a subgraph to its supergraph, a supergraph feature is most informative when one of C_H or M_H reaches 0 while the other (M_H or C_H) is maintained at its current value. As such, we compute the best p-value (as given by the Chi-Square test) that a supergraph can achieve on a traversal of a particular branch. If this score is insignificant, then we prune this branch. This allows us to speed up the subgraph mining process. In this study, we only consider subgraph patterns with size up to 6 edges to keep running time reasonable. The size of subgraphs considered influences the running time. There is an exponential increase in running time for every increase in the number of edges we consider. Six was, therefore, selected as this was the maximum size of subgraph that we could mine within one hour from a dataset of over 2000 API usage examples. The size of the subgraph affects the specificity of the features we mine; if we set the size to a greater value, then it could mine larger patterns. However, the larger the pattern is, the more likely it is specific to a few usage locations, while smaller patterns are more likely to be found in more usage locations. Empirically, all the mined subgraphs in our experiments had fewer than 6 edges.

3.3.6 Selection of examples to label

ALP involves a human-in-the-loop, using multiple rounds of labeling. It initially queries for labels of several dozens randomly sampled unique methods. For the subsequent rounds of labelling, ALP applies principles of active learning [240, 147, 199] to the selection of queries for labeling. Different from typical scenarios that active learning is applied to [147], the choice of selected examples to label and the identification of discriminative subgraphs features are closely coupled. For a subgraph to be identified as a discriminative subgraph, it has to occur among the labeled usage examples enough times such that it can be statistically more common in either the correct usage or misuse examples. Therefore, a sufficient number of graphs that contain the subgraph must be labeled. Otherwise this subgraph, regardless of how potentially discriminative it can be (given a hypothetical fully-labeled dataset), cannot be selected as a feature.

However, it is prohibitively expensive for us to label enough graphs to have information about every subgraphs. As described earlier in Section 3.2, performing active learning on graphs requires the selection of query graphs in each round. These query graphs are the usage examples that the human annotator will annotate in the coming round. Existing studies [240, 199] suggest a good selection of query examples should be both informative and

representative. A query graph should not be similar to graphs already labeled (i.e. informative), and it should be similar to other graphs that are unlabeled (i.e. representative).

Informativeness Using these principles, we identify heuristics for picking examples to label. To pick informative examples, we only select queries from the graphs uncovered by the currently selected features. Thus, the query graphs will be dissimilar from already-labeled graphs as they will not share features. Due to the coupling between the choice of examples to label and subgraph features, we propose that the informativeness of labeled examples can be viewed through the informativeness of the subgraphs features they contain. As such, we measure the informativeness of subgraphs using the notion of coverage following the work of Wang et al. [401], but viewed through the lens of graphs and subgraphs:

- Coverage: the proportion of total examples containing at least one selected subgraph. Having a high coverage indicates that we can characterize the space of the API usage using the selected subgraphs. The more the coverage increases when a subgraph is selected, the more informative the subgraph is.

In the example of labeling `java.util.Map`, if ALP has already seen enough examples to know that a null-check following *get* is a common and correct usage pattern, ALP focuses the labeler's attention to examples that do not contain a null-check following *get*.

Representativeness To prevent the selection of non-representative examples, i.e. outliers, we filter out subgraphs that appear in too few graphs. These subgraphs can never be discriminative even if all graphs are labeled, as the number of graphs they appear in is too small. The number of graphs, *min_signif*, required to be significant is pre-computed based on the Chi-Square test of independence and the threshold of statistical significance (p-value=0.05). We pick only query graphs containing the remaining subgraphs. In other words, we only select graphs containing potentially discriminative subgraphs, which are subgraphs that are contained in at least *min_signif* graphs. Consequently, graphs containing such subgraphs share at least one feature with *min_signif* $-$ *1* other graphs, and are less likely to be outliers.

To pre-compute *min_signif*, we simply enumerate possible values starting from 1 and pass these values to the Chi-Square test, given the number of occurrences of correct and incorrect uses among the currently labeled set of examples. We pick the smallest value that satisfies the threshold of statistical significance (0.05) and set *min_signif* to this value.

```
String rule = inputSourceMap.get(ruleName);
...
InputSource is = new InputSource(
  StreamUtil.stringTOInputStream(rule));
```

Figure 3.16: An example usage of `java.util.Map` that is not representative. It uses project-specific code (`StreamUtil.stringTOInputStream`). ALP has little to gain even if this example was labeled.

In the example of labeling `java.util.Map`, ALP should avoid wasting the labeler's attention on outliers, such as uncommon usages that are specific to a given client project (e.g. the example shown in Figure 3.16). Knowing the labels of the outliers would not help ALP to learn discriminative subgraph features.

Constraint solving For each iteration, we view the selection problem as an optimization problem where we pick unlabeled examples to optimize quality metrics with respect to some constraints. The first constraint is that for each potentially-informative subgraph, we only want to pick the minimal number of graphs for the subgraph to be selected as a discriminative feature. The second constraint is that, in each batch, we pick at most 0.5% of graphs to label. Therefore, solving the optimization problem can be seen as the selection of a minimal number of graphs to maximize our knowledge of subgraphs that cover many graphs.

We encode information about the graphs, subgraphs, and subgraph isomorphisms as a logic program. We pass the logic program as input to an off-the-shelf logic program solver, Clingo [168], selected for its ease-of-use and its strong performance against other systems [100]. The solution to the optimization problem is the next set of examples to label. These graphs are labeled and passed to the subgraph mining algorithm. This process is continued until one of two stopping criteria is satisfied: 1. more than 95% of the training dataset has been covered by the identified subgraph features, or 2. we have labeled 5% of the dataset. As only 0.5% of graphs are labelled in each batch, it may take up to 10 batches to satisfy the stopping criterion of 5% of the dataset being labeled. These values were picked arbitrarily. We assumed that a limited number of examples were collected and 5% was chosen as a sweet spot. Empirically, we collected an average of 2330 usage examples per API, and on average, only about 1% of the examples were labeled.

In this logic program, we consider a set of predicates. First, we define `graph(i)` for all uncovered graphs in the dataset. Next, we define

`subgraph(j)` for all **frequent** subgraphs that were mined as an intermediate product of the discriminative subgraph mining process. Note that these are frequent subgraphs, and not discriminative subgraphs, as we are trying to determine the next set of graphs to label. Therefore, these graphs do not yet contain a subgraph that we have found to be discriminative. After this, `covers(i, j)` is defined over `graph(i)` and `subgraph(j)`, and it holds subgraph if `j` is present in the graph `i`. As earlier described, `coverage` gives us the total number of graphs that have been covered; containing at least one subgraph that may be discriminative. The output of executing the logic program on Clingo is the answer set containing the query graphs, which are the graphs that are `selected`. These are the usage examples that will be labeled in the next iteration. Logically, we express the two desired properties of informativeness and representativeness by maximizing the coverage of graphs, given a set of subgraphs that may be discriminative subgraphs.

```
1  % each graph can be selected only once
2  { selected(G) : graph(G) } <= 1 :- graph(G).
3
4  % a subgraph may_be_discriminative if n graphs are selected
5  % and contains the subgraph
6  may_be_discriminative(SG) :- #count {
7    G : selected(G), covers(G, SG), graph(G) } >= n,
8    subgraph(SG).
9
10 % s or less graphs are selected
11 :- { selected(G)} > s.
12
13 % a graph is covered if a subgraph that may_be_discriminative
14 % is contained in it
15 coverage(X) :- X = #count {G: covers(G,SG), graph(G),
16 may_be_discriminative(SG) }.
17
18 #maximize {X : coverage(X) }.
```

Figure 3.17: Expressing the constraints as a logic program, trying to find the best `selected` graphs based on the subgraphs that may be discriminative.

We show the encoding of these properties in Figure 3.17. After the logic program is executed, the output are the query graphs, which are the graphs that are `selected`. Line 2 specifies a constraint that each graph can be selected for labelling at most once. Lines 5-8 determine if a subgraph may

be discriminative given the currently selected graphs. A frequent subgraph has to appear a statistically significant number of times (`n` in Figure 3.17) among the labelled graphs for it to be considered a discriminative subgraph. Therefore, if a subgraph appears in fewer labeled graphs than this threshold, it cannot be discriminative. In this formulation, we consider that a subgraph *may be discriminative* if it appears in at least `n` query graphs, where `n` is the support required for a subgraph to be discriminative. Line 101denotes that the total number of query graphs should not exceed `s`, set to be 0.5% of the total number of graphs. This number is the number of instances that the human annotator has to label in the next iteration. Lines 15 to 18 declares that the choice of `selected` graphs should maximize the coverage. In short, we find a small set of graphs such that labelling them may help to identify the set of subgraphs that maximises coverage.

Observe that, at each iteration of selecting a batch of query examples, the coverage can only increase as we only select graphs that were not previously covered. This helps in identifying query graphs that are more informative than if we had randomly selected examples. Next, the selection of query graph targets only subgraphs that may be discriminative. This helps to select only graphs that are representative.

We visualize this process in Figure 3.18, and to improve the intuition of this process, we work backwards from our objective. Our objective is to maximise coverage of all unlabeled graphs. A graph is covered when a subgraph is discriminative and is contained in this graph. A sub-objective is, then, to determine the choice of discriminative subgraphs that can maximise coverage. To determine if a graph is discriminative, ALP requires a statistically significant number of graphs to be labeled. In Figure 3.18, to simplify our example, we assume that only one graph is required to be labeled for a subgraph to be discriminative. Then, in this simple example, three subgraphs will suffice to cover 9 out of 10 examples. The next sub-objective is to select the graph for labeling. One graph is selected for each potentially discriminative subgraph. As there are three subgraphs, three graphs that contain these subgraphs are selected. By labelling these three selected graphs, we will have information about the subgraphs they contain, and these subgraphs cover most of the examples.

3.3.7 Graph classification

The final part of ALP is the classification module which uses classification with a reject option. Apart from (M)isuses and (C)orrect, we introduce a third decision, *reject*, expressing uncertainty and inability to classify the usage instance accurately. This module comprises of a machine learning clas-

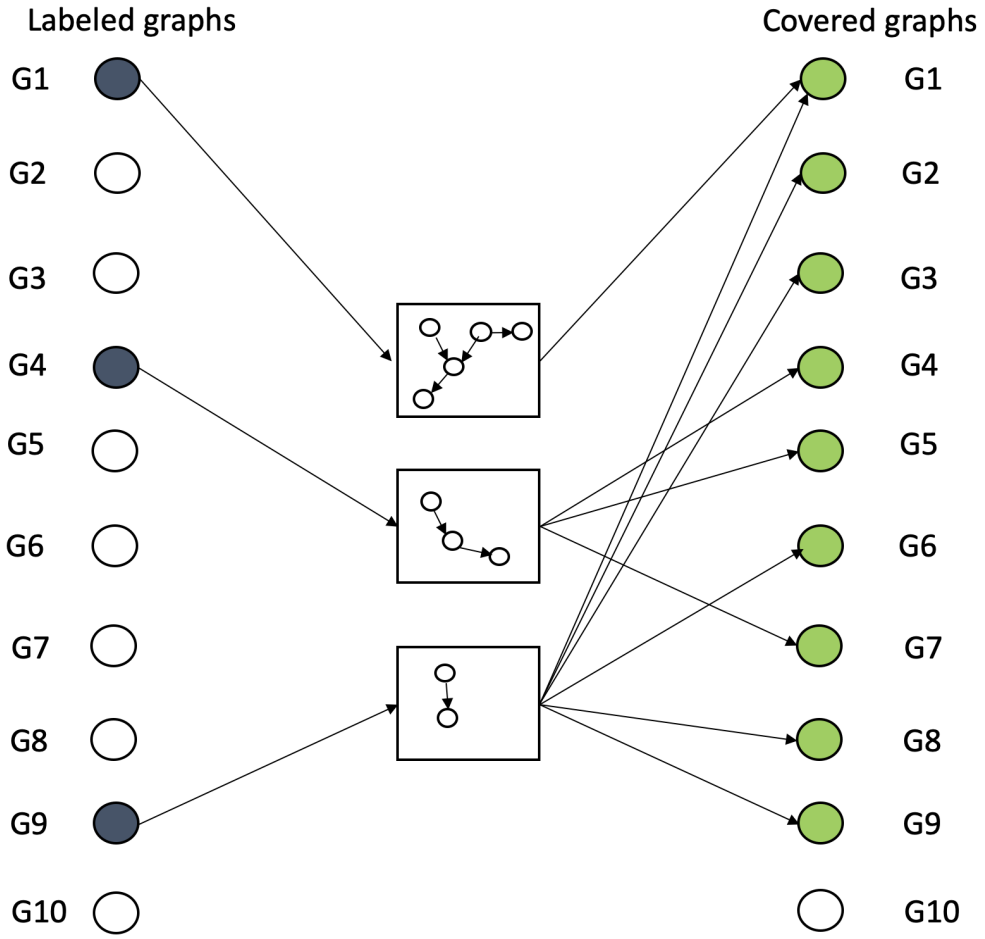


Figure 3.18: Our objective is to maximise coverage (green dots on the right, each dot represents a one usage example) while minimizing the number of labeled examples (black dots on the left). In this example, by labeling 3 graphs, we gain enough information about the subgraphs that can collectively cover 9 graphs.

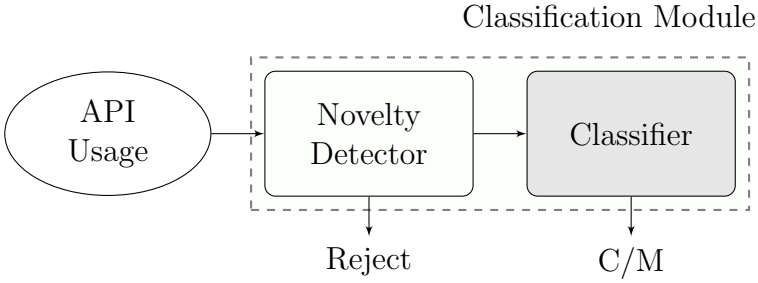


Figure 3.19: ALP's Classification Module. Given an input method, it either signals that it cannot be classified (Reject), or classifies it as a misuse (M) or correct use (C).

sifier and a novelty detector, as seen in Figure 3.19. To classify an API usage instance, the input API usage is encoded into an EAUG, G, and subgraph isomorphism tests are performed for each discriminative subgraph [180]. This gives us the feature vector, v.

$$v_i = \begin{cases} 1 & \text{if } F_i \sqsubseteq G \text{ (i.e. } F_i \text{ is a subgraph of } G) \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

v is a binary indicator vector, given in Equation 3.3. v_i is set to 1 if the graph contains the i-th discriminative subgraph feature, F_i.

To train the classification module, a grid-search is done on the training data to identify the best-performing (in terms of F1) classifier in 5-fold cross validation. As the dataset is imbalanced, we oversample instances of the minority class. SVM (with linear, RBF kernel), K-nearest neighbors and Bayes Classifier (Naive and Complement [340]) are included in the grid search. Our choice of classifiers exclude Deep Learning models as they need voluminous data, but our dataset typically have only about 50 labels for each API. We train one model per API.

Novelty detection The novelty detector allows the classification module to identify that a code usage is abnormal and is unrepresented in the training dataset. Novel usage patterns can appear during practical usage, and cannot be characterized by the set of patterns identified from the training dataset. As we train our classification model, the novelty detector is also tuned on the training dataset. To perform novelty detection, we use the Local Outlier Factor [94] algorithm. For each data point, the algorithm measures how isolated is it with respect to its neighbors. The novelty detector can be

viewed as a binary classifier, categorizing an example as an outlier or not. If it is an outlier, then ALP withholds its judgement. We use an off-the-shelf implementation[1] of this algorithm in this work.

This algorithm was selected for its effectiveness on data distributions where the data forms clusters of different densities. We expect that API usage examples follows this distribution. Due to alternate usage patterns, we expect that majority of examples belong to a core-pattern cluster, with smaller, loose clusters containing other examples using alternative patterns. An advantage of this technique is that it provides an outlier factor, which is a measure of how much of an outlier a point is, rather than just producing a binary output. We train one model for each API.

3.4 Empirical Evaluation

3.4.1 Benchmarks

To evaluate the effectiveness of ALP, we run an empirical evaluation using two benchmarks. MUBench was used to evaluate prior work [59, 60, 61], and the AU500 dataset that was constructed from API usage instances in real-world projects. The training dataset comprises usage examples of the API selected by ALP from GitHub, To evaluate ALP on the benchmarks, we took care to construct the training dataset such that API usage examples from projects in the benchmarks were not labeled. Therefore, ALP takes advantage of data from many projects from GitHub, but this dataset is separate from the testing dataset. For each API, an average of 2330 usage examples (at the granularity of a method) were collected from GitHub. Using the procedure described in Section 3.3.6, we use an average of 23 labels for each iteration, and an average of slightly over 2 iterations were required for each API. This quantity is fewer than other studies that use active learning for tasks related to defect and fault prediction [382, 278], as well as studies using active learning for inferring state machine specifications [397], which may require the user to answer several hundred queries.

We studied the same APIs as prior work [61], which are 57 APIs from both the Java standard library as well as third party libraries. Through the discriminative mining process, we mined about 6 discriminative subgraphs per API. As the size of the vector passed to the machine learning classifier is equal to the number of discriminative subgraphs, the average vector size in our experiments is also 6.

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html`

All experiments were run on a *Ubuntu* server with an *Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz* and 64GB of RAM.

MUBench. First, we perform the experiments using the API Misuse Detector benchmark, MUBench [59], that was used to systematically evaluate the existing misuse detectors. These existing misuse detectors are Tikanga [413], Jadet [414], DMMC [291], GrouMiner [303], MUDetect [61], MUDetectXP (MUDetect on its cross-project setting, where it mines multiple projects [61]). Tikanga, JADET, DMMC, GrouMiner, and MUDetect mine patterns from the project with the misuse. Instead of only a single project, MUDetectXP uses Boa [143] to search for examples on GitHub, mining API usage patterns from multiple projects. We report the results that were previously reported by Amann et al. [61]. Similarly, we use examples from GitHub to train ALP. While the existing approaches are unsupervised, ALP harnesses the power of limited human supervision to boost its performance.

We follow the experimental setup from prior work [60] for computing Recall and Precision. For computing Recall, an experimental procedure known as Experiment R was proposed. In Experiment R, the findings of the detectors are inspected to count the number of known misuses reported. The extended version of MUBench was used, containing 208 methods with a misused API. Consisting of misuses identified in prior studies, MUBench allows comparison between misuse detectors over these known misuses. This procedure allows us to compare and contrast the misuses found by each detector. We use ALP to train models of the APIs studied in Experiment R.

As studies [153, 239, 215] have shown that developers rarely use tools producing many false positives, an experimental setting for computing Precision is provided by MUBench. In this experimental setup, Experiment P, ten projects from the MUBench are selected and the misuse detectors are run on these projects. We use the same projects as the previous studies: Of these ten projects, five were used by Amann et al. [59] in the original version of MUBench, while another five were included in the extended version of MUBench [61]. These projects were selected as they were among the projects where previously studied detectors could successfully run on, and at most one detector did not report any findings [61].

To compute precision, the findings of the misuse detectors are investigated manually. Because of the large volume of findings that are reported by existing misuse detectors, only the top-20 findings of each detector on each project are investigated, allowing us to compute Precision@20. This is the same procedure used for evaluating Precision in previous studies [60, 59, 61]. This keeps the amount of effort required to investigate the findings reasonable. ALP does not produce a ranking of findings. To allow for comparison against the other detectors, we sample 20 reported misuses using the outlier

Table 3.1: Number of usage sites in our evaluation dataset, AU500

Category	Project	Commit	Sites
GUI	Apache FOP	1942336d7	59
	SwingX	820656c	16
	JFreeChart	893f9b15	51
	iTextPdf	2d5b6a212	71
Commons	Apache Lang	0820c4c89	34
	Apache Math	1abe3c769	22
	Apache Text	7d2b511	13
	Apache BCEL	5cc4b163	3
Security	Apache Fortress	a7ab0c01	6
	Santuario	3832bd83	15
	Apache Pdfbox	72249f6ff	17
	Wildfly-Eytron	a73bbba0f0	14
Database	JackRabbit	da3fd4199	90
	H2Database	0ea0365c2	84
	Curator	2af84b9f	3
	Apache BigTop	c9cb18fb	3
		Total	500

factors produced by the novelty detector, which may be interpreted as a degree of confidence that ALP has in a particular finding. As the outlier factors are not comparable between different APIs, we randomly sample (without replacement) 20 misuses inversely weighted by its outlier factor; we are less likely to sample a misuse if the misuse is likely to be an outlier.

MUBench [59] also provided an experiment setup for the Recall Upper Bound. In this setting, perfect examples to mine patterns of, i.e. only correct usage examples, are provided to the detectors for each misuse. However, this is irrelevant to ALP and MUDetectXP [61], as they mine GitHub for usage examples.

AU500.

To gain a different perspective of the overall effectiveness of ALP, we construct a new dataset. To avoid bias from focusing on the misuses collated in MUBench, which were previously identified by existing misuse detectors, we sample 500 API usage instances randomly from 16 open-source Java projects, which are clients of the APIs we study. We use the same client projects as those investigated by Wen et al. [416] These projects were identified as projects that were from diverse domains, categorized based on their official definition and GitHub topics [416] . Wen et al. [416] studied the APIs in

Table 3.2: Breakdown of correct and misuses in AU500

Project	**# Misuses**	**# Correct usage**
Apache FOP [7]	18	42
SwingX [40]	3	15
JFreeChart [25]	10	42
iTextPdf [23]	13	56
Apache Lang [3]	4	30
Apache Math [4]	3	16
Apache Text [5]	1	12
Apache BCEL [11]	1	2
Apache Fortress [8]	0	6
Santuario [31]	2	13
Apache Pdfbox [9]	6	11
Wildfly-Eytron [43]	4	10
JackRabbit [24]	5	85
H2Database [16]	43	41
Curator [6]	0	3
Apache BigTop [2]	2	1
Total	115	385

MUBench, as well as popular APIs discussed on StackOverflow [449]. However, in this work, we look only for uses of the APIs studied by Amann et al. [61], as we use MUDetectXP as our baseline. The details of the projects that are given in Table 3.1. We use exactly the same projects studied by Wen et al. [416], identifying the right commits based on the dates that the projects were accessed by Wen et al. [416]. Each usage instance of an API was labeled by human annotators as either a correct use or a misuse. The dataset and labeling guideline are available on the artifact website.

As we randomly sampled usages from the clients projects, the distribution of API usages in the AU500 reflects the actual distribution of 9972 API usage locations in the client projects. A further breakdown of the API usages into misuses and correct usages is given in Table 3.2.

We had 4 annotators label the dataset. Only one of the annotators is an author of the paper reporting this study. One of the annotators has a working experience of over 10 years in industry and all the annotators have at least one year of experience. Each usage instance was independently labeled by at least 2 annotators. When there were disagreements, consensus was reached through a discussion between the annotators.

A typical annotator took about 4 minutes to annotate each usage instance,

Table 3.3: The number of misuses with a particular violation type, based on the MUC, in the AU500

Violation Type	**# violations**
Missing Method Call	51
Missing Condition	62
null check	21
value or state	41
Missing Exception Handling	2

but first spent up to 32 minutes to understand the labeling guideline and the APIs through reading its Javadoc, related StackOverflow posts, and examples of misuses. Similar to previous studies on API misuses, we consider usages at the granularity of a method. When considering a usage example, an annotator was expected to look at the rest of the source code in the same file. The usage example is presented in its original source code with code comments intact. We instructed the annotators to use the projects' Javadoc documentation and code comments of other files in the project when required. If a class or method from a third-party library is used, then the annotators were instructed to use the documentation of the third-party library.

For inter-annotator agreement, we computed Fleiss' Kappa [154] of 0.49, which is interpreted as Moderate Agreement. Disagreements were caused by differences in opinion about other mistakes in the same function and if they should be considered as a violation of the API's usage constraint. Another source of disagreement was over the enclosing class' undocumented invariants and usage constraints.

Next, we used the API-Misuse Classification (MUC), developed by Amann et al. [60] to label each misuse with the type of violation. The MUC allows for the comparison of different API misuse detectors in terms of the types of violation they can detect. the MUC distinguishes between misuses with missing or redundant program elements. The breakdown of violation types in the AU500 is given in Table 3.3. Compared to MUBench, the AU500 has 4 categories of violation types related to missing API usage elements, while MUBench has 7 categories of missing API usage elements. In the AU500, there are no violations with redundant program elements, which are the minority of misuses in MUBench. However, similar to MUBench, the vast majority of misuses in the AU500 are related to missing method calls or condition checks. While MUBench had a single instance for misuses related to missing or additional synchronization, context, iteration, the AU500 does not have any misuses related to them. In the construction of the AU500, we

selected the same APIs as the APIs studied in MUBench. However, while MUBench largely consists of misuses identified in prior studies, the instances in the AU500 were randomly sampled from 16 projects. Therefore, the distribution of misuses in the AU500 are representative of the distribution of misuses in the 16 projects.

Compared to the projects evaluated in MUBench's Experiment P, AU500 share 3 projects (IText, JFreeChart, Apache Math), but different versions (as identified by their commits) of each project were used in the AU500 and MUBench.

For the AU500, we compute Recall, Precision and F-measure by running the misuse detectors on all the projects, and retain only the reported misuses among the 500 annotated instances. As every test instance was annotated, we can compute Precision instead of computing Precision@20 as we have done for MUBench. Precision can be computed similarly to its use in machine learning literature, in which it is the proportion of true misuses among the reported misuses in the 500 labeled instances. True positives is the size of the intersection of the reported misuses and the instances labeled as a misuse. False positives is the size of the intersection of the reported misuses and the instances labeled as a correct usage. False negatives is the size of the intersection of the reported correct usages and the instances labeled as a misuse.

$$\text{Precision} = \frac{\text{TP}}{\text{FP+FP}} \text{ Recall} = \frac{\text{TP}}{\text{FP+FN}}$$

Finally, the F1 is the harmonic mean of *Precision* and *Recall*:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision+Recall}}$$

To obtain the results for MUDetectXP, we run MUDetectXP on the 16 projects, but we do not retrain it on new data as it already had models for the APIs we study. Both ALP and MUDetectXP learn their models from data obtained from GitHub.

3.4.2 Research Questions

Our evaluation aims to answer these research questions:

RQ1. Is ALP able to detect misuses previously detected by existing tools?

This research question concerns the ability of ALP to detect misuses found by existing misuse detectors. We compare ALP against existing misuse detectors on MUBench.

RQ2. Does ALP find more misuses compared to existing approaches?

The objective of this research question is to evaluate the effectiveness of ALP. Rather than focusing on the misuses found in previous studies, we evaluate ALP on the AU500, constructed from API usage instances in real-world projects.

RQ3. What is the effect of having a reject option and EAUG in ALP?

In this research question, our objective is to have an ablation study, where we determine if the reject option and our extensions to the AUG are extraneous. To minimize false positives, ALP leverages a novelty detector to reject classifying test usage instances that are dissimilar to training examples. Also, to incorporate more signals beyond program elements used in the execution of a program, ALP used an extended version of the AUG that captures more information. Using the same metrics, we compare the performance of ALP with and without the two components on the AU500 dataset.

RQ4. What is the effect of using a different training dataset?

ALP was trained with usage examples of APIs mined from GitHub. In machine learning-based approaches, the quality of data used for training is known to affect effectiveness. Therefore, in our last research question, we investigate the change in performance of ALP when a different training dataset is used. We observe and compare the change in performance metrics when ALP is trained on the same dataset used by MUDetectXP [61]. This training data was constructed through the 2015 GitHub dataset from BOA [143], containing up to 1,000 usage examples for each API.

3.4.3 Experimental Results

RQ1. Performance of ALP on MUBench.

Table 3.4 and 3.5 summarize the results of our evaluation on MUBench. During our manual inspection of the results, we find that one of the misuses in MUBench, identified by MUDetect in prior work, was, in fact, a false positive. This case, identified as a misuse of `java.util.Map`, is actually a usage of Multimap[2]. Unlike Map, Multimap does not return null, and instead returns an empty collection when *get(key)* is invoked with a key not contained in the map. As its usage in this case is not a misuse, we removed this case from our evaluation. Another misuse site was in a project that was no longer publicly accessible, and as such, we also removed this case from our

[2]`https://guava.dev/releases/23.0/api/docs/com/google/common/collect/Multimap.html#get-K-`

Table 3.4: Statistics of running MUBench's Experiment P. Other than ALP, the results of the detectors were taken from prior work [61].

Detector	**Experiment P**		
	# findings	**# true misuses**	**Prec. @ 20**
Tikanga	85	7	8.2%
JADET	91	8	8.8%
DMMC	161	12	7.5%
GrouMiner	156	4	2.6%
MUDetect	146	32	21.9%
MUDetectXP	91	31	34.1%
ALP	164	72	**43.9**%

Table 3.5: Statistics of running MUBench's Experiment R. # unique is the number of misuses found only by one detector. Other than ALP, the results of the detectors were taken from prior work [61].

Detector	**Misuses in Experiment R**		
	# found	**# unique**	**Recall**
Tikanga	13	0	6.3%
JADET	7	1	3.4%
DMMC	21	4	10.1%
GrouMiner	5	0	2.4%
MUDetect	42	4	20.2%
MUDetectXP	90	6	43.3%
ALP	117	35	**56.3**%

evaluation. Without using examples from GitHub, MUDetect outperforms the other existing tools in both precision and recall, and by using examples from GitHub, MUDetectXP improves substantially over MUDetect and is the strongest baseline.

For Precision@20, following the procedure described in MUBench [60], we[3] manually inspected the top 20 cases found by ALP on each project, and finished with a precision of 43.9%. We computed Cohen's Kappa to measure the agreement between the two annotators and obtained a Kappa value of 0.75 – usually interpreted as substantial agreement [248, 368]. According to Landis and Koch [248], a Kappa value between 0.40 to 0.6 corresponds to moderate agreement, while 0.6 to 0.8 is substantial agreement, and a value

[3]Two annotators, including a non-author of the paper that reported this research work who was not informed about the purpose of the labels

above 0.8 is almost perfect agreement. In total, ALP reports 164 violations in the top-20 findings in the ten projects, identifying 72 true misuses. Compared to existing detectors, ALP outperforms the baseline detectors in terms of Precision, with the strongest baseline, MUDetectXP, achieving a precision of 34.1%. Table 3.4 summarizes Experiment P's results.

For Recall, ALP identifies 117 misuses out of the 208 misuse (56.3%) in MUBench. In contrast, MUDetectXP identifies 90 cases out of 208 misuses (43.3%). Thus, ALP improves over the state-of-the-art by 13%.

The results of Experiment R are summarized in Table 3.5. From the results, there are 35 misuses that only ALP can find, and there are just 15 misuses that the existing detectors can find that ALP is unable to. This shows that ALP is capable of detecting misuses that none of the existing misuse detectors can detect.

Next, we construct a composite baseline detector built from all baselines. This detector reports a misuse at a particular location as long as a single baseline detector reports a misuse. This composite detector would correctly report 111 misuses, and have a resulting Recall of 53.4%, which is still fewer than the 117 misuses and recall of 56.3% ALP detects. Comparing the composite detector and ALP, there are 15 misuses found only by the composite detector, while there are 35 misuses found only by ALP. This indicates that ALP is complementary to prior techniques.

Compared to MUDetectXP, ALP was able to mine more usage constraints with fewer spurious patterns. Together with the extensions to the AUG, ALP was able to mitigate the effect of self-usages (where the internal implementation of an API calls itself) without the need for handcrafted heuristics. We will provide a further discussion of the differences between ALP and MUDetectXP in Section 3.5.1.

We investigated the true misuses that ALP did not find. For misuses related to some APIs, we find that there are too few examples on GitHub to learn any meaningful patterns. Neither ALP nor MUDetectXP was able to detect misuses related to these APIs as both correct and incorrect usages of these APIs are too rare. These APIs include:

- *org.apache.jackrabbit.core.config.ConfigurationParser*,
- *org.apache.commons.lang.text.StrBuilder*,
- *org.apache.commons.httpclient.auth.AuthState*.

We did not observe any trend regarding the misuses that ALP could detect in the AU500 dataset based on the categories of misuses in the API-Misuse Classification (MUC) [60] framework, which categorizes misuses based on the

Table 3.6: Experimental results on the AU500 dataset

Detector	Prec.	Recall	F1
ALP	**44.7**%	54.8%	**49.2**%
ALP (w/o reject option)	36.0%	**62.6**%	45.7%
ALP (w/o EAUG)	19.0%	45.2%	26.7%
MUDetectXP	27.6%	29.6%	28.6%

type of violation (e.g. missing null-check). ALP was able to detect misuses in the AU500 belonging to every category of the MUC. Consistently, the remaining misuses that were undetected by ALP are spread across various categories.

RQ2. Performance of ALP on the AU500. Next, we compare the evaluation metrics of the best-performing baseline, MUDetectXP [61], against ALP on the AU500. As MUDetectXP was shown to outperform the other existing baseline detectors, we focus on it.

Table 3.6 shows the evaluation metrics of ALP and MUDetectXP on our evaluation dataset. Of the 500 usage instances to classify, ALP reports 141 of them to be misuses, identifying 63 true misuses correctly, while MUDetectXP reports misuses for 123 of them, identifying 34 true misuses correctly. ALP finds 29 more misuses than MUDetect. In total, there are 115 true misuses among the 500 usage sites. Therefore, ALP has a recall of 54.8% while MUDetectXP has a recall of 29.6% (a difference of 25.2%). ALP has a precision of 44.7% while MUDetect has a precision of 27.6% (a difference of 17.1%). Overall, ALP achieves an F1 of 49.2%, a substantial improvement (of 20.6%) over MUDetectXP, which achieves an F1 of 28.6%.

Among the true misuses detected by either tool, ALP managed to find 36 misuses that MUDetectXP did not identify, while MUDetectXP identified 7 misuses that ALP did not detect. There are 27 misuses that both MUDetectXP and ALP identified. These numbers indicate that if combined, ALP and MUDetectXP will detect 70 of the 115 true misuses (a recall of 60.9%). This represents an increase in Recall of about 6% compared to the use of ALP alone, and it suggests that ALP is complementary to MUDetectXP. Overall, ALP outperforms the state-of-the-art approach, MUDetectXP, but a developer trying to detect as many misuses as possible, regardless of the amount of developer effort needed, should use both tools together.

RQ3. Effect of reject option and EAUG on ALP

The first two rows of Table 3.6 shows the performance of ALP with and

without the reject option. Without the reject option, ALP reports 59 more instances are buggy (among the 500 annotated usage instances), however, its precision is lowered by 8.7%. Among the 59 more findings, only 9 of them were true misuses (or about 15.3%). This indicates that the novelty detector was helpful in withholding inaccurate decisions. ALP's recall increases by about 7.8% without the reject option, but overall, it attains a reduced F1 of 45.7% compared to 49.2%.

On MUBench Experiment R, ALP finds 1 more misuse when the reject option is removed. Without the use of the novelty detector, we have no means to rank the misuses reported by ALP. Therefore, we are unable to compare the change in performance in Precision@20, which requires the top 20 misuses for each project. Instead, we run ALP with and without the reject option, and we report the total number of misuses reported among the projects. Without the reject option, ALP reports 543 misuses in total, while with the reject option, it reports 440 misuses in total. Thus, by removing the reject option, ALP would report 23% more usage locations. We sampled 30 of the 103 rejected misuses, and found that just 3 of them were true positives. Therefore, we see that the reject option helps to mitigate **Challenge 2**, and overall, improves F1.

Next, we evaluate ALP with and without the extensions made to the AUG. The third row of Table 3.6 shows the performance of ALP without the EAUG, i.e., using only the original AUG. We found that the performance of ALP dropped substantially without the EAUG, with a reduction in precision of over 20% on the AU500. On MUBench's Experiment R, the Recall of ALP dropped to 24.5%, a decline of 30%. We observed that the decrease in effectiveness was caused primarily by reporting many false positives related to `ResultSet`, which implements the `Closeable` interface. Without the EAUG distinguishing between parameters and local variables, ALP would report that any usage of the `ResultSet` without a corresponding *close* method call as a misuse, even if the `ResultSet` was passed in as an argument or was a field. Therefore, we conclude that capturing this information, as done in the EAUG, is important for distinguishing between correct and incorrect usages of some APIs.

RQ4. Effect of using a different training dataset

In our final research question, we compare the performance of ALP when using the same training data as MUDetectXP [61]. Referring to the dataset used by MUDetectXP as the GitHub 2015 dataset, Table 3.7 summarizes the differences in performance metrics. On both MUBench and the AU500, overall performance declined when the GitHub 2015 dataset was used. On MUBench, both Precision@20 and Recall drops from 43.9% to 36.3% and from 56.3% to 47.6%. On the AU500, Precision drops from 44.7% to 28.2%.

Table 3.7: Summary of differences in performance metrics when ALP is trained with the same dataset (GitHub 2015) as MUDetectXP

Detector	**MUBench**		**AU500**		
	Prec.@20	**Recall**	**Prec.**	**Recall**	**F1**
ALP (original data)	**43.9%**	**56.3%**	**44.7%**	54.8%	**49.2%**
ALP (GitHub 2015)	36.3%	47.6%	28.2%	**58.3%**	38.0%
MUDetectXP	34.1%	43.3%	27.6%	29.6%	28.6%

While Recall increased from 54.8% to 58.3%, the overall performance, in terms of F1, dropped from 49.2% to 38.0%.

Compared to MUDetectXP, all performance metrics of ALP remained higher. In particular, the F1 of ALP is almost 10% higher than the F1 of MUDetectXP on the AU500. The results suggest that ALP improves over MUDetectXP even when using the same training dataset, validating our findings and design decisions, even though the choice of training dataset influences overall effectiveness.

Overall, the experimental results suggest that the effectiveness of ALP depends heavily on having a training dataset that is sufficiently large. This supports the intuition that ALP leverages the diversity in the training dataset to mine discriminative subgraphs and construct an accurate classifier. Qualitatively, we observed that the smaller dataset had the consequence of having lower diversity among the usage examples used for training. This hindered the identification of discriminative subgraphs for many APIs. For some APIs, there were extremely few examples of incorrect usage in the smaller dataset, preventing ALP from learning a model of that API.

3.5 Discussion

3.5.1 Qualitative Analysis

So far, we have shown a quantitative evaluation of ALP. Next, we perform a qualitative evaluation of ALP. In this section, we look into the features that were identified by ALP and MUDetectXP. We also discuss cases of misuses of the most common APIs, which provide insights as to why ALP was able to detect some misuses that MUDetectXP missed, as well as some cases that ALP failed to detect a misuse.

Developers may implement a class that extends `Enumeration`. In their implementation of the *nextElement* method, the obligation to check for *has-*

MoreElements falls on the client of the new class. Take, for example, the code snippet from a real GitHub project given in Figure 3.20. Reporting *new FileInputStream(fileNames.nextElement())* as a misuse of *nextElement* results in a false positive. ALP correctly identifies that the information about implementing the `Enumeration` interface is an important feature for detecting misuses of *nextElement*. This is possible as ALP includes subtyping in its model of source code, the EAUG. Therefore, ALP is able to correctly detect that this usage is a correct usage, avoiding a false positive. On the other hand, existing misuse detectors are incapable of encoding any pattern that captures such information, showing the difficulty of **Challenge 3**, which motivates an expressive representation of programs.

```
class MyInputStreamEnumerator implements
       Enumeration<FileInputStream> {
    private Enumeration<String> fileNames;

    public MyInputStreamEnumerator(
            Enumeration<String> fileNames) {
        this.fileNames = fileNames;
    }
    @Override public boolean hasMoreElements() {
        return fileNames.hasMoreElements();
    }
    @Override public FileInputStream nextElement() {
        FileInputStream ret = null;
        try {
            ret = new FileInputStream(
                    fileNames.nextElement());
        } catch(FileNotFoundException ex) {
            ex.printStackTrace();
        }
        return ret;
    }
}
```

Figure 3.20: A client of *nextElement*. A salient feature recognised by ALP is that the class implements *Enumeration*, highlighted in red, which helps ALP to correctly judge that this is not a misuse without the need for handcrafted heuristics.

Next, we investigate the patterns mined by ALP and MUDetectXP as both MUDetectXP and ALP produce subgraph patterns as a by-product. As each usage instance was compared against the mined patterns, having many patterns that are unrelated to the API usage will lead to lower effectiveness. We analyze these subgraph patterns for `java.util.Map`. As mentioned previously, we focus on `Map` due to its prevalence in virtually all Java projects.

The results are given in Table 3.8. While MUDetectXP did not detect

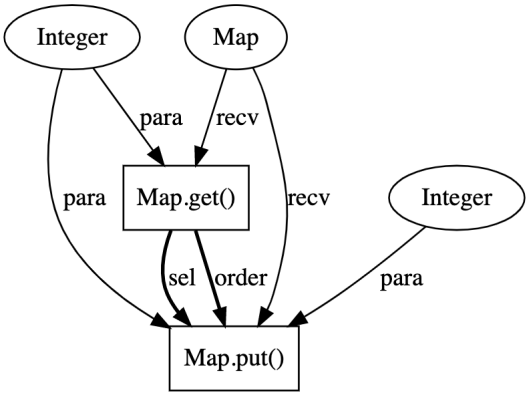


Figure 3.21: A spurious usage pattern mined by MUDetect. While fetching an object from a map and putting it into another is frequent, its violation leads to no consequences.

Table 3.8: Patterns discovered by MUDetectXP and ALP

Detector	**total patterns**	**# unrelated**
MUDetectXP	81	72
ALP	35	9

any misuse of `java.util.Map` in MUBench, MUDetectXP mines 81 patterns related to it. Of these patterns, 72 (88%) do not relate any of the program elements (e.g. method invocations, parameter values) of Map, and instead are spurious patterns (e.g. related to printing data to standard output) that occur frequently with Map in the dataset. These patterns can never identify a misuse of Map, only producing false positives. The remaining patterns are related to Map, but do not represent usage constraints. For example, the pattern in Figure 3.21 was identified by MUDetectXP and led to several false positives. While this pattern frequently appeared, it is not a usage constraint; when violated, there are no consequences. This demonstrates that MUDetectXP faces **Challenge 1 and 2**; it is difficult to automatically and reliably determine the correctness of a pattern, and when given a violation of a pattern, it is difficult to determine if the violation really is a misuse.

ALP mines 35 patterns of `java.util.Map`. Of these patterns, just 9 (25%) of them are patterns that do not show any relationship between program elements of Map. The patterns mined for each API are available on the artifact website. **This suggests that ALP addresses Challenge 1**

and 2; it reduces the number of spurious patterns that are mined, and largely picks up subgraphs that are more likely to affect the correctness of the API usage.

One reason that ALP detects some misuses that are missed by MUDetectXP is that MUDetectXP failed to mine the usage constraint using its subgraph mining algorithm. For example, MUDetectXP is not able to detect misuses related to `java.util.List`, which was often misused in cases when its size was not checked before invoking *get(index)* on it. We hypothesize that this limitation is caused by the different ways that the size can be checked before invoking *get(index)*, for example, within a for-loop or checking for *isEmpty* before invoking *get(0)*, as well as the possibility of having arbitrary amounts of code in between the check on the size and the invocation to *get(index)*. **On the other hand, ALP is able to represent a usage using multiple disjoint subgraphs, and the use of human labels allows it to filter out subgraphs that are incidentally mined from the arbitrary code in between the relevant code.**

We also inspected some cases that ALP did not successfully detect. Because ALP makes a decision based on subgraphs which are disjoint from one another, it may fail to detect a misuse when a single method contains multiple instances of the object. For instance, if there are multiple uses of an API in a method, it cannot detect cases where the first object is correctly used, but the second object is not. The first correct usage masks the incorrect second usage. In contrast, MUDetectXP is able to detect that the second usage violates a usage constraint.

In Section 3.4.3, we observed that ALP was less effective when using the smaller and, as a result, less diverse dataset. This is related to another limitation of ALP, in which it cannot identify patterns which occur extremely rarely. Both correct and incorrect usage patterns that are rare cannot be identified as discriminative subgraphs as ALP will consider them outliers. MUDetectXP shares a similar limitation. If it fails to identify any patterns from a limited set of examples, then it cannot detect any misuses. However, unlike ALP, MUDetectXP only requires examples of correct usages.

Ultimately, while ALP outperforms MUDetectXP quantitatively, both tools are complementary, outperforming the other under different circumstances. Fundamentally, both approaches rely on different strategies, having strengths and limitations that are different from each other. As mentioned previously in Section 3.4.3 (RQ2), and they should be used together if the goal is to detect as many misuses as possible.

To conclude, our takeaways from the qualitative evaluation are as follows:

- ALP's extensions to the API Usage Graph allowed it to correctly detect

self-usages (calling another method from within the same interface) without the need for handcrafted heuristics.

- ALP has some success in addressing the challenges we described at the start of this section. It identifies usage constraints while minimizing the number of irrelevant patterns mined.
- By representing each graph using a vector of disjoint subgraphs, ALP avoids some difficulties faced by prior studies.
- ALP and MUDetectXP may outperform each other under different circumstances and have different limitations, and to detect as many misuses as possible, both tools should be considered.

3.5.2 Threats to Validity

To mitigate threats to internal validity, we double checked our source code and data, however, errors may remain. Whenever possible, we reused existing, off-the-shelf implementations of algorithms to reduce the risk of implementation mistakes. We also make our source code publicly available on the artifact website. Another threat is bias in our dataset. To mitigate this bias, each instance is labeled by at least one annotator with industry experience.

For minimizing threats to construct validity, we reused the same benchmark and evaluation metrics as previous studies. For the AU500 dataset, as we have a complete set of labels, we reuse evaluation metrics that are more commonly used for various classification problems in other domains.

The projects used by ALP differ from those used by MUDetectXP, however, both sets of projects were obtained from GitHub. Since the projects were sampled from GitHub, we do not expect substantial differences between the projects used. However, we note that ALP downloaded a larger corpus of usage examples (about 2000 examples of API usage examples compared to only 1000 for MUDetectXP). When we modified the source code for MUDetectXP to mine patterns from up to 2000 examples, we found that MUDetectXP faces scalability issues. Running on a machine with 64GB of RAM, MUDetectXP exhausts all the available memory while mining patterns. Therefore, we are unable to complete the experiments without changing the subgraph mining process used by MUDetectXP, which would fundamentally change how MUDetectXP identifies patterns. Furthermore, in RQ4 (see Section 3.4.3), we have investigated the performance of ALP when using the same dataset as MUDetextXP and found that ALP still outperformed MUDetectXP on both MUBench and the AU500 dataset. Fundamentally, ALP mines only discriminative patterns considering informative

usage examples, while MUDetectXP mines all frequent patterns considering all examples.

To mitigate threats to external validity, we have used two benchmarks to increase the spread of API usage examples considered. A threat is that our findings may not apply to APIs that were not studied. However, we study the same APIs as prior work, and both the APIs and projects studied are from diverse domains.

3.6 Related Work

Dynamic analysis Researchers have proposed the use of runtime verification to detect violations of API specifications [214], these specifications can be both automatically mined [331] or written by hand. However, Legunsen et al. [255] have shown that both manually written specifications and automatically mined specifications have high false positive rates. This suggests the need for more tooling in mining and writing API specifications. Their analysis revealed that the manually written JavaMOP specifications did not sufficiently encode valid alternative usages. ALP is complementary to these approaches; ALP can help to discover salient patterns for writing specifications. Some patterns may not be obvious and ALP may assist specification writers in discovering unusual usage patterns.

Catcher [229] uses search-based testing to detect API misuses. Focusing on the API of the Java standard library, it generates test cases directed at code using the API of interest to find a test case triggering an exception from the API. MUTAPI [416] is an approach using mutation testing for discovering API misuse patterns. MUTAPI mutates correct usage of an API within client projects. In contrast to our work and other API misuse detectors, MUTAPI is focused on misuse pattern discovery. Its precision was not evaluated on MUBench. While the misuse patterns found by MUTAPI achieved a recall of 49% on the original version of MUBench [59] (with about 50 misuses), our work achieves a recall of about 56% on the extended version of MUBench [60, 61] (with 208 misuse locations). Both Catcher and MUTAPI are limited to misuses which cause exceptions to be thrown, while our approach can detect misuses with other consequences, such as resource leaks. We do not directly compare the performance of ALP and MUTAPI as MUTAPI relies on the execution of a test suite to detect misuses while ALP relies on static analysis to detect misuses. MUTAPI can work only on projects with an available test suite, while ALP is able to detect misuses in projects without test suites or with test suites with limited coverage. Existing research has also shown that

many projects have poor code coverage and that developers do not always write test cases[237, 235].

Other work on API There are other research efforts on other ways to assist development using APIs. Researchers have worked on better search for API usage examples [175, 176, 401], recommendation of APIs [300, 302, 157], generating or improving documentation [445, 131], studying API usage in StackOverflow answers [449, 339], and finding API workarounds [246]. Lamothe et al. [246] showed that developers may intentionally use APIs in ways that are not officially supported by the API developers; these workarounds may be viewed as alternative usage patterns.

3.7 Summary

We propose ALP, which represents programs as graphs and classifies them to detect API misuses. ALP is a human-in-the-loop technique that identifies examples for mining discriminative subgraphs. Through the principles of active learning, a small but informative number of examples are identified and labeled. These examples are used to train a classifier with a reject option, which reserves judgement when encountering programs that it is uncertain about. On both MUBench and our newly constructed AU500 dataset, ALP substantially outperforms existing approaches and the state-of-the-art tool, MUDetectXP.

Chapter 4

(Static Analysis + API) Detecting False Alarms from Automatic Static Analysis Tools

4.1 Overview

It has been 15 years since Findbugs [75] was introduced to detect bugs in Java programs. Along with other automatic static analysis tools (ASATs) [350, 353, 137], FindBugs aims to detect incorrect code by matching code against bug patterns [75, 196], for example, patterns of code that may dereference a null pointer. Since then, many projects have adopted these tools as they help in detecting bugs at low cost. However, these tools do not guarantee that the warnings are real bugs. Many developers do not perceive the warnings by ASATs to be relevant due to the high incidence of effective false alarms [216, 391, 353]. Prior work has suggested that the false positive rate may range up to 91%. While the overapproximation of static analysis may cause false alarms, false alarms do not only refer to errors from analysis or overapproximation, but include warnings that developers did not act on [216, 354, 353]. Developers may not act on a warning if they do not think the warning represents a bug or believe that a fix is too risky.

To address the high rate of false alarms, many researchers [404, 188] have proposed techniques to prune false alarms and identify actionable warnings, which are the warnings that developers would fix. These approaches [181, 186, 232, 241, 351, 435, 417, 364, 234] consider different aspects of a warning reported by Findbugs in a project, including factors about the source

code [181], repository history [417], file characteristics [263, 435], and historical data about fixes to Findbugs warnings [241] within the project. Wang et al. [404] completed a systematic evaluation of the features that have been proposed in the literature, and identified 23 "Golden Features", which are the most important features for detecting actionable Findbugs warnings. Using these features, subsequent studies [429, 432, 431] show that any machine learning technique, e.g. SVM, performs effectively, and that the use of a small number of training instances can train effective models. In these studies, performances of up to 96% Recall, 98% Precision, and 99.5% AUC can be achieved. A perfect predictor has a Recall, Precision, and AUC of 100learning techniques using the Golden Features are almost perfect.

Although the Golden Features have been shown to perform well, we do not know why they are effective. Therefore, in this work we seek to get a deeper understanding of the Golden Features. We find a few issues: First, the ground-truth label was leaked into the features measuring the proportion of actionable warnings in a given context. Second, warnings in the test data were used for training. To understand their impact, we addressed the two flaws and found that the performance of the Golden Features declines. Our results show that the use of the Golden Features do not substantially outperform a strawman baseline that predicts all warnings are actionable.

Next, we investigate the warning oracle used to obtain ground-truth labels when constructing the dataset. To evaluate any proposed approach, a large dataset should be built, where each warning is accurately labeled as either an actionable warning or false alarm. Many studies [404, 429, 432] use a heuristic, which we term the *closed-warning heuristic*, as the warning oracle to determine the actionability of a warning, checking if the same warning is reported in a *reference revision*, a revision chronologically after the *testing revision*. If the file is still present and the warning is not reported in the reference revision, then the warning is *closed* and is assumed to be fixed. It is, therefore, assumed to be actionable. Conversely, a warning that remained *open* is a false alarm. A revision made a few years after the simulated time of the experimental setting is used as the reference revision. Prior studies [404, 429, 432] selected reference revisions set 2 years after the testing revision. However, no prior work has investigated the robustness of the heuristic.

There are several desirable qualities of a warning oracle. Firstly, it should allow the construction of a sufficiently large dataset. Secondly, it should be reliable; the labels should be robust to minor changes in the oracle. Thirdly, it should generate labels that human annotators and developers of projects using ASATs agree with. An advantage of the closed-warning heuristic is that it enables the construction of a large dataset. However, our experiments demonstrate the lack of consistency in the labels given changes in the choice

of the reference revision. This may allow different conclusions to be reached from the experiments. Our experiments also uncover that the oracle does not always produce labels that human annotators or developers agree with. These limitations show that alone, the heuristic do not always produce trustworthy labels. After removing unconfirmed actionable warnings, the effectiveness of the Golden Features SVM improves, indicating the importance of clean data.

We highlight lessons learned from our experiments. Our results show the need to carefully design an experimental procedure to assess future approaches, comparing them against appropriate baselines. Our work points out open challenges in the design of a warning oracle for the construction of a benchmark. Based on the lessons learned, we outline several guidelines for future work.

Subsequently, we analyze the use of pretrained models of code for filtering false alarms. Recently, Kharkar et al. [231] have found that pretrained models of code can be employed as a filter to remove false alarms from a static analyzer, Infer. Their experiments showed that both a zero-shot approach and finetuned pretrained model were able to filter out false alarms while retaining a majority of the true alarms.

In this study, we reproduced the work of Kharkar et al. [231] but find that the performance of the zero-shot approach may not be sufficiently effective. Considering the objectives of both retaining true alarms and filtering out false alarms, the zero-shot approach fails to outperform the use of Infer without any postprocessing, making tradeoffs between precision and recall. We were able to replicate the other proposed approach, and found that it was effective at filtering false alarms, but the process of finetuning the model requires a large number of labelled warnings. Automatically generated labels may be of poor quality [220] and manually labelled data is expensive. This motivates approaches that require only a small number of labels.

To this end, we propose a new approach, TrailMarker. TrailMarker uses in-context learning [96] with a large language model of source code, Incoder [160]. With the use of transformer-based architectures [392], large language models of source code are pretrained with enormous datasets [201]. Surprisingly, they have been shown to exhibit effective performance on many tasks that may require some understanding of program semantics [254, 228, 420]. In-context learning aims to utilize the large amount of background knowledge captured by these models. Instead of providing a large training dataset for finetuning or retraining, in-context learning requires a *demonstration* of a small number of labelled warnings to be provided. Conditioning on the demonstration as context, the large language model can be immediately used for a different task than its pretraining task.

To minimize the number of labels required, TrailMarker uses a novel

strategy of selecting a small number of warnings to present to the human-in-the-loop, who provides the labels of these warnings. TrailMarker aims to leverage in-context learning for learning and generalizing from the demonstration of a a small number of labelled examples. Apart from minimizing the number of labels required, the selection of warnings to use the demonstration would impact the effectiveness of the approach. While Kharkar et al. [231] considers only the program location at which a warning is reported, the execution paths of programs often cross functions and code from multiple files. Incorporating more information may improve the filtering, hence, TrailMarker guides its selection of warnings using information from the execution path analyzed by Infer when providing demonstrations to the large language model of code. Considers the seletion of warnings as a set cover problem, TrailMarker selects warnings such that there is some information for every trace event on the path reported by the static analyzer. As such, TrailMarker constructs informative demonstrations to a large language model.

As the code and dataset used in the experiments of the prior study [231] are not publicly available, we reimplement their proposed approaches and construct a new dataset. Our experimental results shows that TrailMarker achieves an F1 of 87.1% compared to the F1 of 82.8% from the state-of-the-art approach while requiring labels on only a quarter of all training warnings. We also perform an ablation analysis and find that both the use of in-context learning and the set cover strategy for selecting labelled warnings are important to TrailMarker.

4.2 Background

4.2.1 Automatic Static Analysis Tools

Many researchers have proposed Automatic Static Analysis Tools (ASATs), such as Findbugs [75], to detect possible bugs during the software development process. Research has shown these tools are useful and succeed in detecting bugs that developers are interested in with low cost. Compared to program verification or software testing, these tools rely on bug patterns written by the authors of the static analysis tools, matching code that may be buggy. Findbugs includes over 400 bug patterns that match a range of possible bugs, such as class casts that are impossible, null pointer dereferences, and incorrect synchronization.

Studies have also shown that ASATs are able to detect real bugs [383, 179]. Indeed, static analysis tools are adopted by large companies [74, 353,

137] and open source projects [84] to detect bugs at low cost. Developers may run them during local development, use them in Continuous Integration [436] and during code review to detect buggy code to catch bugs early [391, 384, 77, 312]. Projects may configure the tools [436], for example, to suppress false alarms by configuring a filter file to exclude specific warnings [46].

Developers largely perceive ASATs to be relevant, and the majority of practitioners have used or heard of ASATs [375, 391, 280]. Still, these tools are characterized by the large amounts of false alarms that they produce, and among other reasons, this has led to resistance in adopting them in many software projects [216].

4.2.2 Distinguishing between Actionable Warnings and False Alarms

To minimize the overhead of inspecting false alarms, researchers have proposed approaches based on machine learning to rank or classify the warnings. A large number of features have been designed over the past 15 years; for example, based on software metrics (e.g. size of the file, number of comments in the code), source code history (e.g. number of lines of code recently added to a file), and characteristics and history of the warnings (e.g. the number of revisions where the warning has been opened).

Researchers have evaluated their proposed tools through datasets of warnings produced by Findbugs [181, 186, 187, 351, 364, 435, 189]. Recently, Wang et al. [404] performed a systematic analysis of the features proposed in the literature. From 116 features, they identified 23 *Golden Features*, which are the features that achieve effective performance. The features are listed in Table 4.1. These features include metrics such as the code-to-comments ratio [263], and the number of lines added in the past [187, 351]. Of note are several features of the "Warning combination" feature type. We will refer to three of these features, **warning context in method**, **warning context in file**, and **warning context for warning type**, as the **warning context** features. We refer to another two features, the **defect likelihood for warning pattern**, and **discretization of defect likelihood** as the **defect likelihood** features. These features are various measures of the proportion of actionable warnings within a population of warnings, building on top of the insight that warnings within the population share the same label, e.g. if a warning was previously fixed in a file, it is more likely that the other warnings in the same file will be fixed too.

Further research [429] on the Golden Features of Wang et al. [404] showed the lack of influence of the choice of machine learning model on effectiveness.

Table 4.1: The Golden Features studied in prior work [404, 429, 432]. A *warning context* is defined [404] as the difference of the number of actionable warnings and false alarms divided by the total number of warnings reported in a given method/file, or for a warning pattern. We provide more descriptions of each feature in our replication package [28].

Feature type	**Feature**
Warning combination	warning context in method warning context in file warning context for warning type defect likelihood for warning pattern discretization of defect likelihood average lifetime for warning type
Code characteristics	comment-code ratio method depth file depth # methods in file # classes in package
Warning characteristics	warning pattern warning type warning priority package
File history	file age file creation developers
Code analysis	parameter signature method visibility
Code history	LOC added in file (last 25 revisions) LOC added in package (past 3 month)
Warning history	warning lifetime by revision

They suggested that a linear SVM was optimal since it requires a lower cost of training. In contrast, while a deep learning approach achieves similar levels of effectiveness, it has a longer training time. Their analysis [429] suggested that the detection of false alarms is an intrinsically easy problem. A different study [432] demonstrated that, with the Golden Features, only a small proportion of the dataset has to be labelled to train an effective classifier. The Golden Features are a subject of our study. In Section 4.4, we analyze them in detail.

Closed-warning heuristic. The procedure to construct and label the

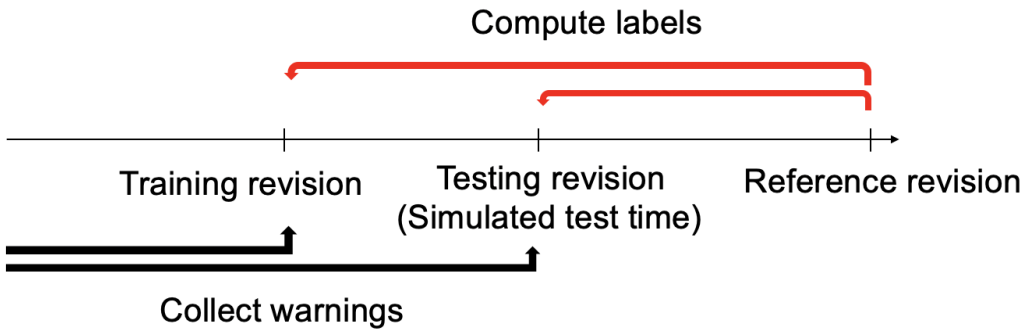


Figure 4.1: The dataset comprises warnings created before the training and testing revisions. The labels of each warning are determined by the closed-warning heuristic; if a warning is closed at the reference revision and the file has not been deleted, then it is actionable.

ground-truth dataset can be visualized in Figure 4.1. To assess an approach that detects false alarms, a dataset of Findbugs warnings is collected. While some researchers [186, 364] construct a labelled dataset through manual labelling of the warnings in a single revision, other researchers collect a dataset through an automatic ground-truth data collection process [181, 186, 404, 432, 429]. Data for a *testing revision* and at least one *training revision*, set chronologically before the testing revision, is collected. This simulates real-world use of the tool, in which training is done on the history of the project, and then used at the time of the testing revision.

Using the *closed-warning heuristic* as the warning oracle, each warning in a given revision is compared against a *reference revision* set in the future of the test revision. If a specific warning is no longer present in the reference revision (i.e., a *closed warning*), the heuristic assumes that the warning is actionable. If the warning is present in both the given and reference revision (i.e., an *open warning*), then the heuristic assumes that it is a false alarm. If the file that contains the code with the warning has been deleted, then the warning is labelled *unknown* and is removed from the dataset. In other words, according to the the closed-warning heuristic, a closed warning is always actionable as long as the file has not been deleted, and an open warning is always unactionable. Beyond studies for detecting actionable studies, researchers have also applied the heuristic to identify bug-fixing commits to mine patterns of bug fixes [270, 271]. The heuristic is a subject of our study, and we assess its robustness and its level of agreement with human oracles in Section 4.5.

4.3 Study Design

4.3.1 Research Questions

RQ1. Why do the Golden Features work?

This research question seeks to understand the Golden Features. While previous studies have highlighted their strong results, there has not been an in-depth analysis of their practicality. We study the Golden Features and the dataset used in the experiments by Wang et al. [404] and Yang et al. [429]. We investigate the aspects of the features and dataset that allow accurate predictions by the best performing machine learning model, an SVM using the Golden Features. We replicate the results of the previous studies and validate the predictive power of the Golden Features. To understand the importance of different features, we use LIME [341] to narrow our focus down to the features that contributes the most to the predictions. Afterwards, we switch to increasingly simpler classifiers and analyze the experimental data to better understand why the choice of classifiers did not influence the results in prior studies.

RQ2. How suitable is the closed-warning heuristic as a warning oracle?

This research question concerns the suitability of the *closed-warning heuristic* as a warning oracle. A good oracle should be robust, and its judgments should agree with the analysis of a human annotator. We investigate the robustness of the heuristic, checking the consistency of labels under different choices of the reference revision. While previous studies used a 2-years interval between the test revision and reference revision, we investigate if different conclusions can be reached with a different time interval. Next, we compute the proportion of closed warnings that human annotators labelled actionable, and the proportion of open warnings that project developers suppressed as false alarms.

4.3.2 Evaluation Setting

To analyze the performance of machine learning approaches that identify actionable Findbugs warnings, we use the same metrics as prior studies [432, 429, 404]. A true positive (TP) is an actionable Findbugs warning correctly predicted to be actionable. A false positive (FP) is an unactionable Findbugs warning incorrectly predicted to be actionable. Note that we use the term *false alarm* to refer to unactionable Findbugs warning. A *false positive*, therefore, refers to a false alarm that is incorrectly determined to be an ac-

tionable warning. A false negative (FN) is an actionable warning incorrectly predicted to be a false alarm. A true negative is a unactionable warning correctly predicted to be a false alarm.

We compute Precision and Recall as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$$

Finally, we compute and present F1, the harmonic mean of Precision and Recall. F1 is known to capture the trade-off between Precision and Recall, and is used in place of accuracy given an imbalanced dataset. F1 is computed as follows:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision+Recall}}$$

The Area Under the receiver operator characteristics Curve (AUC) is a measure of the predictive power of a machine learning approach to distinguish between true and false alarms. Ranging between 0 (worst discrimination) and 1 (perfect discrimination), AUC is the area under the curve of the true positive rate against the false positive rate, and recommended over accuracy when the data is imbalanced. A strawman classifier that always outputs a single label has an AUC of 0.5.

Our dataset consist of projects that were studied by Yang et al. [429, 432] and Wang et al. [404]. Similar to previous studies [432, 429], we use one training revision and one testing revision. We use the same testing revision as previous studies [432, 429, 404]. We train one model for each project.

4.4 Analysis of the Golden Features

To answer the first research question, we investigate the performance of the Golden Features by first using the same dataset used by Yang et al. [429]. The dataset includes two revisions from 9 projects. The testing revisions are the revisions of the projects on 1 January 2014, and the training revision is a revision of the projects up to 6 months before the testing revision. In total, 31,058 warning instances were obtained by running Findbugs over the training and testing revision. On average, 14.1% of the warnings in the dataset were actionable. Table 4.2 shows the breakdown of the warnings.

We successfully replicate the performance observed in the experiments of Yang et al. [429] and Wang et al. [404], obtaining high AUC values of

Table 4.2: The number of training, testing instances, and the percentage of actionable warnings (Act. %) in the dataset. The testing revision is the last revision checked into the main branch on 2014-01-01.

Project	**Training**	**With duplicates**		**W/o duplicates**	
		testing	**Act. %**	**testing**	**Act. %**
ant	1229	1115	5%	21	71%
cassandra	2584	2601	14%	551	70%
commons	725	786	5%	4	50%
derby	2479	2507	5%	499	31%
jmeter	604	613	24%	57	19%
lucene	3259	3425	34%	893	59%
maven	813	818	3%	149	14%
tomcat	1435	1441	23%	227	41%
phoenix	2235	2389	14%	214	22%

up to 0.99, identical to the study of Yang et al. [429]. An average F1 of 0.88 was obtained, with F1 ranging from 0.65 to 0.95. Table 4.3 shows the experimental results that we have obtained.

Yang et al. [429] found that the dataset was intrisically easy as the data was inherently low dimensional. To further analyze their findings, we used tools from the field of explainable AI, in particular LIME [341], to identify the most important features contributing to each prediction. LIME is an explanation technique that identifies the most important features that contributed to an individual prediction. To identify the most important features, we sampled 50 predictions made by the Golden Features SVM, and used LIME to identify the top features contributing to the predictions. We found that two features, **warning context of file** and **warning context of package**, appeared in the top-3 features of every prediction.

Warning context and defect likelihood features. On analyzing the source code of the feature extractor developed by Wang et al. [404], we found a subtle data leak in the implementation of the warning context and defect likelihood features. These features utilize findings from previous studies [241] that found that the warnings within a population (e.g. warnings in the same file) tend to be homogenous; if one warning is a false alarm, then the other warnings in the same population tend to be false alarms as well. Including **warning context of file** and **warning context of package**, there are another 3 features computed similarly (**warning context of warning type**, **defect likelihood for warning pattern**, **Discretization of defect likelihood for warning pattern**). At a high-level, the warning context features

Table 4.3: Effectiveness of an SVM using the Golden Features after separately removing the leaked features and removing the duplicate warnings between the training and testing dataset. The numbers in parentheses are the F1 obtained by the baseline classifier that predicts all warnings are actionable.

Project	**Golden Features**		**— leaked features**		**— data duplication**	
	F1	**AUC**	**F1**	**AUC**	**F1**	**AUC**
ant	0.94 (0.09)	1.00	0.11 (0.09)	0.67	-	-
cassandra	0.92 (0.24)	1.00	0.45 (0.24)	0.86	0.9 (0.41)	0.99
commons	0.65 (0.10)	0.99	0.16 (0.10)	0.65	0.75 (0.25)	0.97
derby	0.95 (0.09)	1.00	0.39 (0.09)	0.93	0.97 (0.28)	0.97
jmeter	0.94 (0.38)	0.99	0.53 (0.38)	0.76	1.00 (0.14)	1.00
lucene-solr	0.87 (0.51)	0.97	0.59 (0.51)	0.74	0.87 (0.53)	0.98
maven	0.86 (0.07)	1.00	0.27 (0.07)	0.9	0.95 (0.24)	0.99
tomcat	0.93 (0.37)	1.00	0.48 (0.37)	0.73	0.95 (0.70)	1.00
phoenix	0.89 (0.25)	1.00	0.42 (0.25)	0.78	0.83 (0.37)	0.99
Average	0.88 (0.23)	1.00	0.38 (0.23)	0.76	0.90 (0.37)	0.99

Table 4.4: Effectiveness of an SVM using the Golden Features after removing both the leaked features and removing the duplicate warnings between the training and testing dataset. The numbers in parentheses are the F1 obtained by the baseline classifier that predicts all warnings are actionable.

Project	**Golden Features**		**— leak, duplication**	
	F1	**AUC**	**F1**	**AUC**
ant	0.94 (0.09)	1.00	-	-
cassandra	0.92 (0.24)	1.00	0.29 (0.41)	0.54
commons	0.65 (0.10)	0.99	0.11 (0.25)	0.49
derby	0.95 (0.09)	1.00	0.30 (0.28)	0.59
jmeter	0.94 (0.38)	0.99	0.25 (0.14)	1.00
lucene-solr	0.87 (0.51)	0.97	0.23 (0.53)	0.62
maven	0.86 (0.07)	1.00	0.27 (0.24)	0.58
tomcat	0.93 (0.37)	1.00	0.65 (0.70)	0.39
phoenix	0.89 (0.25)	1.00	0.40 (0.37)	0.63
Average	0.88 (0.23)	1.00	0.31 (0.37)	0.59

are computed as follows:

$$\frac{|W_{relevant}^{actionable}| - |W_{relevant}^{false\ \ alarm}|}{|W_{relevant}|}$$

$W_{relevant}$ refers to the set of warnings relevant to the feature type. For example, $W_{relevant}$ of **warning context of file** considers the warnings that are reported in a given file, while $W_{relevant}$ of **warning context of warning type** considers all warnings for the given category of patterns (e.g. `STYLE`, `INTERNATIONALIZATION`). Note that a warning pattern refers to a specific bug pattern in Findbugs (e.g. "ES_COMPARING_STRINGS_WITH_EQ"), and a warning type is a category of patterns. The **defect likelihood for warning pattern** [364] feature computes the proportion of warnings that were actionable out of all warnings with the given bug pattern, p:

$$D(p) = \frac{|W_{relevant}^{actionable}|}{|W_{relevant}|}$$

The **discretization of defect likelihood for warning type** [364] feature, computed for each type/category T of bug patterns, is a measure of the difference in defect likelihood from the defect likelihood of T for each bug pattern in the category:

$$\frac{1}{|T|-1} \sum_{p \in T} (D(p) - D(T))^2$$

The five warning context and defect likelihood features require information about the actionability of each warning in the population of warnings considered. A data leakage occurs when the classifier utilizes information that is unavailable at the time of its predictions [387, 227]. As shown in Figure 4.2, while the ratio of actionable warnings are computed over the warnings reported in the past (the black line in Figure 4.2), the closed-warning heuristic to determine the ground-truth label of a warning (the red lines in Figure 4.1 and Figure 4.2) is utilized to determine if these warnings were actionable. To compute the warning context of a given warning, W_t in the testing revision, the labels of all warnings in the population of warnings (e.g. all warnings in the same file), including W_t, are obtained based on comparison to the reference revision.

Since the ground-truth label is also obtained based on comparison to the reference revision, the ground-truth label is *inadvertently leaked* into the computation of the warning context. This is not a realistic assumption in practice; at test time, the ground-truth label of the warning context of W_t is the target of the prediction. While checking if a warning will be closed 2 years in the future is possible within an experiment, there is no way to check

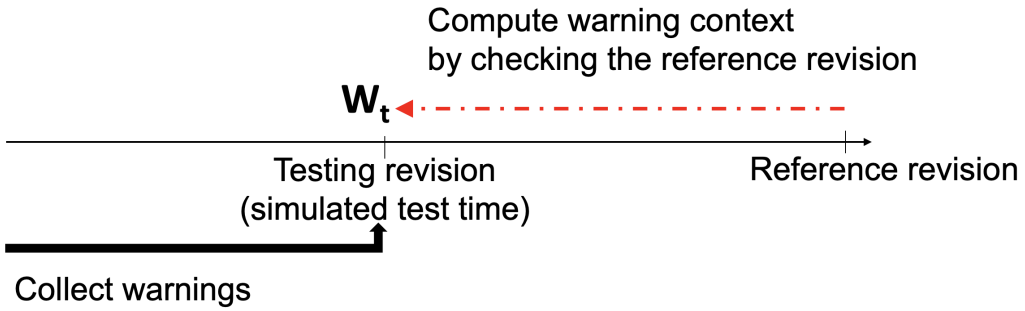


Figure 4.2: The warning context and defect likelihood features use labels derived through the closed-warning heuristic, using information from the reference revision, chronologically in the future of the test revision. In a realistic setting, this information will not be present at test time.

if the warnings will be closed 2 years into the future in practice. Table 4.3 shows the large drop in F1, from an average of 0.88 to 0.38, when these five features are dropped. We refer to these features as *leaked features*.

Baseline using data leakage. Data leakage leads to an experimental setting that overestimates the effectiveness of the classifier under study [387, 227]. In Table 4.5, we show that a baseline equivalent to the Golden Features can be developed using only the five leaked features. Using just the leaked features with an SVM, we construct a baseline that achieves performance comparable to the use of the Golden Features. An SVM using the leaked features has a Recall of 0.79, about 0.10 lower than the Golden Features SVM, however, they achieve identical Precision of 0.94, which results in an F1 of 0.83, just 0.05 lower than the Golden Features. This indicates that the strong performance of the Golden Features in the experiments depends largely on the leaked features, and is an optimistic estimate of their effectiveness.

The computation of the *warning context* and *defect likelihood* features caused data leakage, as it used labels determined by comparison against the reference revision, chronologically in the future of the testing time.

Data duplication. Next, we progressively selected simpler machine learning models and surprisingly, found that a k-Nearest Neighbors (kNN) classifier performs effectively. In particular, we found a surprising trend where the lower values of k led to better results. The results of the experiment where we iteratively lowered k to consider in the prediction are shown in Table 4.5.

Surprisingly, a kNN classifier with k=1 (i.e., only one neighbor is considered to make a prediction) produces the best result, obtained a Precision

of 0.87, a Recall of 0.90, with an F1 of 0.84. With k=1, the classifier was selecting a single most similar warning in the training dataset. In typical usage of kNN, a low value of k may cause the classifier to be influenced by noise and outliers, which makes the strong results surprising. To analyze the results further, we observed that the number of training (15,363) and testing instances (15,695) were similar, and we investigated the data carefully. We found that many testing instances appeared in both the training and testing dataset.

The data duplication was caused by the data collection process, in which all warnings produced by Findbugs for both the training and test revisions were included in the training and testing dataset. Say we have a warning at the training revision, determined to be open and, therefore, unactionable by the closed-warning heuristic. In other words, the warning remained open in the period before the training revision to the reference revision. Then, the warning would certainly be opened at the testing revision, which is chronologically before the reference revision but after the training revision. Likewise, if we have a warning only closed after the testing revision, but was open during the testing revision, then the same warning would be present at both the training and testing revision with the same "actionable" label. Consequently, a large number of warnings appear in both the training and testing dataset. This contributes to an unrealistic experimental setting.

Table 4.5: Average Precision (Prec.), Recall, and F1 of various approaches on the original dataset by Yang et al.

Technique	**Prec.**	**Recall**	**F1**
Golden Features SVM	0.84	0.94	0.88
– leaked features	0.26	0.70	0.38
– data duplication	0.88	0.93	0.90
– data duplication and leaked features	0.27	0.57	0.31
+ reimplemented leaked features	0.32	0.57	0.38
Golden Features kNN (with k=10)	0.91	0.57	0.68
Golden Features kNN (with k=5)	0.86	0.72	0.78
Golden Features kNN (with k=3)	0.87	0.78	0.82
Golden Features kNN (with k=1)	0.87	0.90	0.84
Only leaked features SVM	0.79	0.94	0.83
Repeat label from training dataset	0.72	0.80	0.75

Baseline using duplicated data. Data duplication creates an artificial experimental setting that inflates performance metrics [54]. To confirm that the data duplication contributes to the ease of the task, we construct a weak

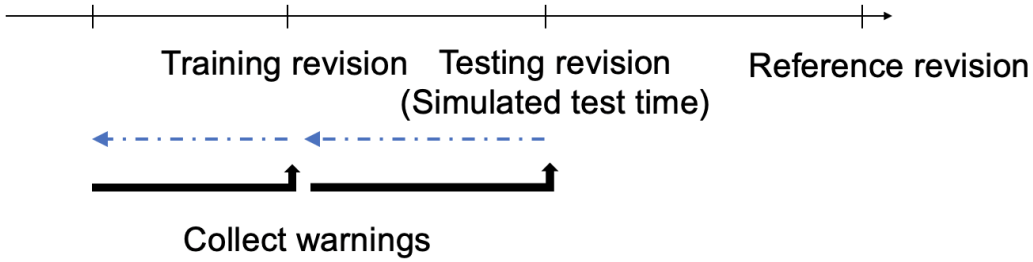


Figure 4.3: We reimplemented the leaked features. The reimplemented features use only information (represented by the blue, dashed lines) available at the present (i.e., either the training or test revision) to determine if a warning (i.e., created before the training or test revision) has been closed. Under this setting, no information from the reference revision is used for making predictions.

baseline, a dummy classifier, that leverages the duplication of testing data in the training dataset. Given a warning from the testing dataset, the classifier heuristically identifies the same warning from the training dataset by searching for a training warning based on the class name (e.g. "BooleanUtils") and bug pattern name (e.g. "ES_COMPARING_STRINGS_WITH_EQ"). If there are multiple warnings with the same class name and bug pattern name, a random training instance is selected from among them. The classifier then outputs the label of the training instance. If there is no training instance with the same class and bug pattern type, then the classifier defaults to predicting that the warning is a false alarm, which is the majority class label.

Table 4.5 shows the comparison of various approaches, including the baseline approaches, on the dataset. The dummy classifier achieves a strong performance, achieving a Precision of 0.72, a Recall of 0.80, and an F1 of 0.75. While the dummy classifier underperforms the model using the leaked features, it outperforms the Golden Features SVM without the leaked features. This indicates that using just two attributes, (1) the class name and (2) the bug pattern of the warning, is enough to obtain strong performance on a dataset with data duplication. Therefore, we conclude that the data duplication between the training and testing dataset contributes to the strong performance observed in previous studies.

The experimental results are summarized in Table 4.5. With both the leaked features and duplicated data, the average F1 was 0.88. After the data leakage features are removed, F1 decreased to 0.38. After removing the duplicated data, F1 decreases further to 0.31. The average project's AUC decreased from 1.00 to 0.59. In comparison, using a strawman baseline that

predicts that every warning is actionable produces an F1 of 0.52 (with an AUC of 0.5).

All warnings reported by FindBugs on both the training and testing revisions were included in the datasets. Warnings reported at the training revision may still be reported at the testing revision, leading to data duplication between the training and testing dataset.

Experiments under a more realistic setting. To better understand the performance of the Golden Features SVM, we ran another experiment where the two issues of data leakage and data duplication have been fixed. First, we deduplicated the test data from the training dataset. Instead of including all warnings in the testing revision, we only consider new warnings introduced between the time after the training revision and before the testing revision. Figure 4.3 shows our procedure. As compared to the previous dataset construction process in Figure 4.1, only the warnings created after the training revision and before the testing revision are used for testing. This better reflects real-world conditions where all warnings prior to usage are used for training, but none of the testing data involves warnings that have already been classified. In total, the number of warnings in the testing revisions decreased from a total of 15,695 to 2,615 after deduplication. Without the duplicated data and without using the leaked features, the average F1 drops from 0.88 to 0.31 as seen in Table 4.3 and Table 4.4.

Next, we reimplemented the leaked features to investigate the effectiveness of Golden Features SVM. To prevent data leakage, we modified the definition of the leaked features. Figure 4.3 visualizes the computation of the warning context and defect likelihood features. Instead of considering all warnings, we consider only warnings that were introduced in the 1 year duration before the training or testing revision. Instead of using the reference revision, we use the given revision (i.e., either the training or testing revision) to determine if the warning was closed. A warning is closed at a given revision if Findbugs does not report it. In other words, for the training revision, only the warnings created within the past year before the training revision are considered. For testing, only the warnings created within one year before the testing revision are considered. A time interval of 1 year was selected in contrast to the study by Wang et al. [404], which used time intervals of up to 6 months. Unlike Wang et al. [404], for the testing revision, we consider only warnings created after the training revision to prevent data duplication. Consequently, we found fewer newly created warnings in the short time interval between the training and testing revisions.

Note that after reimplementing the warning context and defect likelihood

features, we could not run the experiments for the project Phoenix as we faced many difficulties building old versions of the project. Moreover, their revision history did not go back beyond 3 years, required for computing the warning context and defect likelihood features for the training revision. This limitation is not present for Wang et al. [404], as they compute the features by checking if the given warning is closed in the reference revision, set in the future of the test revision (causing data leakage). As such, we omit Phoenix for the rest of the experiments.

Table 4.5 shows the performance of the Golden Features SVM using the reimplemented features. Without the leaked features, the Golden Features SVM achieves an F1 of 0.38. Even with the reimplementation of the leaked features, the Golden Features SVM underperforms the strawman baseline, which predicts all warnings are actionable, with an F1 of 0.43.

Answer to RQ1: After removing the data leakage and data duplication, our experimental results indicate that the Golden Features SVM underperforms the strawman baseline, although its AUC (> 0.5) suggests that the Golden Features have predictive power.

4.5 Analysis of the Closed-Warning Heuristic

Next, given that the quality and realism of the dataset heavily influences the evaluation of the Golden Features SVM, we perform a deeper analysis of the construction of the ground-truth dataset. In previous studies [404, 429, 432], the warning oracle is the *closed-warning heuristic*; a warning is heuristically determined to be actionable if it was closed (i.e., reported by Findbugs in a revision but was not reported by Findbugs in the reference revision, and the file was not deleted), and is a false alarm if it was open (i.e. reported by Findbugs on both the training/test and reference revision).

In the first part of our analysis, we investigate the consistency in the warning oracle given a change in the reference revision. Next, we determine if human annotators consider closed warnings as actionable warnings. Then, we match open warnings against Findbugs filter files in projects that have configured them for suppresssing false alarms. Finally, we observe if cleaner data leads to increased effectiveness of the Golden Features SVM.

4.5.1 Choosing a different reference revision

We perform a series of experiments to determine how the time interval between the test revision and the selected reference revision influences the ground truth label of the warnings. We hypothesize that the longer the time interval between the test and reference revision, the greater the proportion of closed warnings. Based on the closed-warning heuristic, this would cause more warnings to be labelled actionable. If so, the lack of consistency in labels should call the robustness of the heuristic into question. If many bugs are fixed only after many years, then an open warning at any given time may, in fact, be actionable. Besides that, if changing the reference revision leads us to a different conclusion about the Golden Features SVM, then it limits the level of confidence that researchers can have in the experimental results.

In our experiments, we use three reference revisions set two, three, and four years after the test revision. By switching the reference revision, we observe changes in the average actionability ratio. While the actionability ratio remained consistent for the 4 out of 8 projects, the actionability ratio increased by over for the other 4 projects, as seen in Table 4.6, Table 4.7, and Table 4.8. Overall, the average actionability ratio increased by 14% when varying the time interval between the test and reference revision from 2 to 4 years. Considering all projects, we performed a Wilcoxon signed-rank test and found that the change in actionability ratio is statistically significant (p-value=$0.03 < 0.05$).

In terms of the effectiveness of the Golden Features SVM, its average F1 increased from 0.39 to 0.57, as seen in Table 4.6, Table 4.7, and Table 4.8. Considering all projects, the Golden Features SVM underperformed the strawman baseline. Our experiments showed some variation of the Golden Features SVM's effectiveness given a change in the reference revision. For instance, the Golden Features SVM achieved a low F1 of 0.06 in Derby when the time interval between the test and reference revision was 2 years, but had a high F1 of 0.72 with a time interval of 4 years.

By changing reference revisions, the problem exhibits different characteristics. Using a reference revision 4 years after the test revision, actionable warnings would be the majority class, while they were the minority class when using the other reference revisions. 4 of 8 projects have an AUC that flipped from one side of 0.5 to the other (e.g. the Golden Features SVM's AUC is under 0.5 on Derby given a 2-years interval, but the AUC increases above 0.5 given a 4-years interval). In short, different conclusions about the task and the effectiveness of the Golden Features may be reached.

Table 4.6: The number of training, testing instances, and the percentage of actionable warnings (Act. %) in the dataset when varying the reference revision to be 2 years after the training revision. The numbers in parentheses are the F1 obtained by the baseline classifier that predicts all warnings are actionable. The testing revision is the last revision checked in to the main branch before 2014-01-01.

Project	**# testing instances**	**2 years**		
		Act. %	**F1**	**AUC**
ant	21	24	0 (0.38)	0.43
cassandra	551	41	0.56 (0.59)	0.52
commons	4	50	0.66 (1.00)	1.00
derby	489	10	0.06 (0.18)	0.33
jmeter	57	17	0.13 (0.16)	0.58
lucene	993	44	0.56 (0.62)	0.58
maven	149	17	0.25 (0.29)	0.41
tomcat	226	42	0.53 (0.59)	0.51
Average	311	40	0.39 (0.43)	0.54

Changing the reference revision may affect the distribution of the actionable warnings, which may impact the conclusions reached from experiments on the effectiveness of the Golden Features SVM.

4.5.2 Unconfirmed actionable warnings

Next, we investigate if closed warnings are truly actionable warnings. There are multiple reasons for a warning to close. Code containing the warning could be deleted or modified while implementing a new feature, and the warning may only be closed incidentally.

To further understand the characteristics of closed warnings, and to determine how likely is a closed warning an actionable warning, we sampled 1,357 warnings (which is more than the statistically representative sample size of 384 warnings) that were closed. Two authors of this study independently analyzed each warning to determine if they were removed for a bug fix. If the warning was closed due to code changes unrelated to the warning, then we do not consider the warning as actionable. If the code containing the warning was modified such that it was not easily discernible if the warning was closed with the intention of fixing the warning, then we consider it "unknown". If the original version of the code had any comments indicating that Findbugs reported a false alarm (e.g. explaining the reason that a seemingly

Table 4.7: The number of training, testing instances, and the percentage of actionable warnings (Act. %) in the dataset when varying the reference revision to be 3 years after the training revision. The numbers in parentheses are the F1 obtained by the baseline classifier that predicts all warnings are actionable. The testing revision is the last revision checked in to the main branch before 2014-01-01.

Project	**# testing instances**	**3 years**		
		Act. %	**F1**	**AUC**
ant	21	43	0.13 (0.60)	0.48
cassandra	551	46	0.61 (0.63)	0.41
commons	4	50	1.00 (0.67)	1.00
derby	489	59	0.58 (0.74)	0.46
jmeter	57	26	0.12 (0.27)	0.4
lucene	993	49	0.58 (0.66)	0.57
maven	149	16	0.27 (0.28)	0.44
tomcat	226	61	0.51 (0.57)	0.48
Average	311	42	0.48 (0.55)	0.53

buggy behavior was expected behavior), then we consider the warning a false alarm. When the labels differed between the annotators, they discussed the disagreements to reach a consensus. We computed Cohen's Kappa to measure the inter-annotator agreement, and obtained a value of 0.83, which is considered as strong agreement [248].

Finally, after labelling, 176 (13%) of the heuristically-closed warnings were considered as false alarms. Another 520 warnings (38%) were categorized as "unknown". Lastly, 660 (49%) warnings were still considered actionable after labelling.

For an example of a warning labelled "unknown", Figure 4.4 shows a fragment of code where Findbugs complains about the use of the `Long` constructor, indicating that `Long.valueOf` would be more efficient. Even though the warning is removed in the reference revision, the entire functionality of the code fragment was changed as shown in Figure 4.5. In such cases, we label the warning as "unknown" instead of "actionable" or a "false alarm", as there is no evidence that the warning was fixed or ignored. We consider that the warning was removed incidentally, and that the annotators are unable to accurately label the warning.

While the closed-warning heuristic considered that a warning could be removed through the deletion of a file, it does not consider other cases where a warning could be incidentally removed through code modification that

```
// remove the old entry in the Conglomerate directory, and add the
// new one.
        if (is_temporary)
        {
    // remove old entry in the Conglomerate directory, and add new one
                if (tempCongloms != null)
                        tempCongloms.remove(new Long(conglomId));
                tempCongloms.put(new Long(conglomId), conglom);
        }
```

Figure 4.4: Example of code that Findbugs reports a warning on. Findbugs warns against using `new Long`, recommending the more efficient `Long.valueOf` to instantiate a `Long` object.

```
-       // remove the old entry in the Conglomerate directory, and add the
-       // new one.
-            if (is_temporary)
+       // Set an indication that ALTER TABLE has been called so that the
+       // conglomerate will be invalidated if an error happens. Only needed
+       // for non-temporary conglomerates, since they are the only ones that
+       // live in the conglomerate cache.
+       if (!is_temporary)
             {
-                   tempCongloms.put(new Long(conglomId), conglom);
-            }
-            else
-       {
```

Figure 4.5: The warning from Figure 4.4 is removed through a change in functionality, unrelated to the warning otherwise.

Table 4.8: The number of training, testing instances, and the percentage of actionable warnings (Act. %) in the dataset when varying the reference revision to be 4 years after the training revision. The numbers in parentheses are the F1 obtained by the baseline classifier that predicts all warnings are actionable. The testing revision is the last revision checked in to the main branch before 2014-01-01.

Project	**# testing instances**	**4 years**		
		Act. %	**F1**	**AUC**
ant	21	43	0 (0.60)	0.32
cassandra	551	43	0.58 (0.60)	0.48
commons	4	50	1.00 (0.67)	1.00
derby	489	66	0.72 (0.80)	0.52
jmeter	57	91	0.71 (0.95)	0.52
lucene	993	67	0.63 (0.8)	0.53
maven	149	16	0.27 (0.28)	0.44
tomcat	226	52	0.64 (0.69)	0.54
Average	311	54	0.57 (0.67)	0.54

does not fix the bug indicated by the warning. Our results indicate that the heuristic may not be sufficiently robust.

Only 47% of closed warnings were labelled actionable by human annotators, implying that many closed warnings are not actionable. Many closed warnings were only closed incidentally.

4.5.3 Unconfirmed false alarms

The lack of consistency of the closed-warning heuristic raises questions about its robustness in labelling the warnings in the dataset. Moreover, our findings from Section 4.5.1 indicate the possibility that some actionable warnings would only be closed given a longer time interval between the test revision and the reference revision. This may reflect real-world conditions, where developers may not prioritize reports from ASATs and may take a long time before inspecting them. Thus, this raises the possibility that open warnings could represent actionable warnings that the developers would only notice with enough time. We run an experiment to understand this effect, focusing on projects that have shown evidence of using Findbugs in their development practices. In this experimental setup, we remove open warnings that are not confirmed by the project developers to be false alarms.

Table 4.9: Number of open warnings in each project matched by their Findbugs filter file. If a warning was filtered, it indicates that the project's developers consider it a false alarm.

Project	**# open warnings**	**# filtered**	**% filtered**
jmeter	710	6	1%
tomcat	1624	9	1%
commons-lang	106	19	18%
flink	4934	4754	96%
hadoop	3053	269	9%
jenkins	1212	178	15%
kudu	1873	464	25%
kafka	4668	2993	64%
morphia	65	0	0%
undertow	347	113	33%
xmlgraphics-fop	949	909	96%
Average (Mean)	1666	818	31%
Average (Median)	1212	178	18%

Some projects, which have integrated Findbugs into their development process, have a Findbugs filter file [46], in which Findbugs can be configured to ignore false alarms. The filter file allows developers to suppress warnings of specific bug patterns on the indicated files. Developers may add warnings to the Findbugs filter file after inspecting the warnings and identifying false alarms. On projects that have created and maintained a Findbugs filter file, we assume that a developer would either fix the buggy code or update the Findbugs filter file after inspecting a warning. If so, then an open warning that is not matched by the Findbugs filter file may not be a false alarm, but has not been inspected by a developer. If an open warning matches the filter, then it has been confirmed by the developers to be a false alarm.

To investigate the proportion of open warnings that are confirmed to be false alarms by project developers, we identified 3 projects (JMeter, Tomcat, Commons-Lang) that have already configured the Findbugs filter file from Wang et al.'s dataset [404], used in the preceding experiments. Next, we searched GitHub for large, mature projects that showed evidence of using Findbugs and have configured a Findbugs filter file. Using the GitHub Search API, we looked for XML files containing the term `FindbugsFilter`, which is a keyword used in Findbugs filter files, in projects that were not forks, filtering out projects with less than 100 stars or had less than 10 lines in the Findbugs filter file. We obtained 8 projects.

Table 4.10: Effectiveness of the Golden Features SVM after removing unconfirmed actionable warnings/false alarms. Act. % refers to the proportion of actionable warnings. The numbers in parentheses are the F1 of the strawman baseline.

Dataset	**Act. %**	**F1**	**AUC**
Original dataset [429, 432]	39.9	0.39 (0.43)	0.54
– unconfirmed actionable warnings	40.0	0.61 (0.57)	0.66
Projects using Findbugs	38.0	0.43 (0.44)	0.62
– unconfirmed false alarms	40.0	0.41 (0.46)	0.60

The statistics of the warnings reported by Findbugs on the projects are displayed in Table 4.9. On average, 31% of the open warnings are matched by the Findbugs filter configured by the developers, although the proportion varies for each project. As there are usually more false alarms than actionable warnings, our results suggest that the majority of open warnings remain uninspected by developers, which may contribute to noise in a dataset.

Only 31% of open warnings have been explicitly indicated by developers to be false alarms. This suggests that only a minority of open warnings may be false alarms. While the rest of the open warnings could be false alarms, they could also be actionable warnings that have not been inspected yet.

Next, we investigate the impact of the unconfirmed actionable warnings and false alarms on the Golden Features SVM. We hypothesize that cleaning up the data will improve its effectiveness.

To study the impact of unconfirmed actionable warnings, we used the dataset of warnings from the projects by Wang et al. [404] and Yang et al. [429, 432]. These projects were the same projects studied earlier in Section 4.5.2. We construct a dataset of warnings with only the warnings confirmed by the human annotators to be actionable warnings. We randomly sampled a subset of open warnings to retain a similar actionability ratio.

For evaluating the effect of unconfirmed false alarms, we used the warnings from the projects that used Findbugs (from Section 4.5.3). However, we omit 4 projects (JMeter, Tomcat, Hadoop, Morphia) where less than 10% of open warnings matched the filter file, as the low percentage may indicate that the Findbugs filter files are not kept up to date in these projects. From the other projects, only open warnings that match the filter file are included. We sampled a subset of closed warnings to retain a similar actionability ratio.

The outcome of our experiment is shown in Table 4.10. Removing unconfirmed actionable warnings led to an increased AUC from 0.54 to 0.68, and

an increased F1 from 0.39 to 0.64. This outperforms the strawman baseline which has an F1 of 0.57, suggesting that cleaner data may increase the effectiveness of the Golden Features SVM. However, removing unconfirmed false alarms did not help. The results may indicate that cleaner data may help and removing unconfirmed actionable warnings, which is the minority class, may have a positive effect on the effectiveness of a classifier.

Answer to RQ2: The closed-warning heuristic conflates closed warnings for actionable warnings and open warnings for false alarms. We find having cleaner data by removing unconfirmed actionable warnings may boost the performance of the Golden Features SVM, motivating the need to denoise data.

4.6 Discussion

4.6.1 Lessons Learned

To detect actionable warnings, the Golden Features are not a silver bullet. Far from being easy, our results indicate that the state-of-the-art machine learning approach only has marginal improvements over a strawman baseline that always predicts that a warning is actionable. Note that our work does not show that the use of machine learning for classifying warnings from static analyzers is impractical or impossible; the AUC of the Golden Features SVM is above 0.5, indicating that the features have predictive power.

All that glitters is not gold; it is essential to qualitatively analyze and understand the reasons for seemingly strong performance. Despite achieving excellent performance, the Golden Features have bugs related to subtle data leakage and data duplication. This emphasizes the importance for a deeper analysis of experimental results, and both quantitative and qualitative analysis are essential. We call for the need for more replication studies, as such works can highlight the opportunities and challenges for future work. While previous studies [432, 429] compared their proposed approaches to other techniques, they did not compare them against strawman baselines. Our work reemphasizes the need to compare newly proposed techniques to simple baselines [161].

The closed-warning heuristic for generating labels allows a large dataset to be built, but is not enough for building a benchmark. Our work sheds light on the limitations of the closed-warning heuristic, suggesting that it may not be sufficiently accurate; warnings may be closed incidentally, and actionable warnings may stay open for years before they are closed.

As a benchmark is essential for charting research direction [369], the construction of a representative dataset is important. Several studies have proposed similar processes relying on the closed-warning heuristic to build a ground-truth dataset [404, 181, 232, 189], while others have relied on manual labelling [186, 187, 364, 351, 435, 234]. Heuristics enables automation, allowing for a dataset of a greater scale. However, heuristics may not be robust enough. On the other hand, solely labelling warnings through manual analysis is not scalable, and may be subject to an annotator's bias. We suggest that datasets proposed in future should rely on **both** heuristics and manual labelling; apart from its greater scale, the closed-warning heuristic enables rich information to be gathered from the activities of the developers to help the manual labelling process. For example, code commits provide richer information, such as the commit message, simplifying the task for human annotators. In contrast, prior studies [186, 187, 364, 351, 435, 234] have relied on annotators who inspected only the source code that warnings are reported on. Our experiments suggest using the closed-warning heuristic, followed by manual labelling is promising – the annotators had a strong agreement (Cohen's Kappa > 0.8), while no strong agreement in manual labelling has been demonstrated in prior work.

A good benchmark requires scale and should be labelled by many annotators. Fields such as code clone detection have created large benchmarks through community effort [349]. This motivates the need for community effort to build a benchmark for actionable warning detection too. As a derivative of this empirical study, we have labelled 1,300 closed warnings, usable as a starting point.

4.6.2 Threats to Validity

A possible threat to **internal validity** is the incorrect implementation of our code. To mitigate this, we reused existing data and code whenever possible, including the dataset by Wang et al. [404] and Yang et al. [429, 432], and the feature extractor by Wang et al. [404]. Our code and data are available [28].

Threats to **construct validity** are related to the appropriateness of the evaluation metrics. We considered the evaluation metrics used in prior studies [404, 429, 432], and also computed F1, which have been used in many classification tasks [233, 336, 450]. F1 captures the tradeoff between Precision and Recall, and is a more appropriate measure on an imbalanced dataset.

Threats to **external validity** concern the generalizability of our findings. We studied nine projects used in previous studies, and we considered another set of projects that actively uses Findbugs. All considered projects were large, mature projects.

Another threat is the focus on Findbugs and Java projects. Our analysis may not generalize to warnings of other ASATs, such as Infer [137]. Findbugs detects a wide range of bug patterns, including bugs patterns shared by other ASATs, and the features are not language-specific. Moreover, we used the same dataset as prior studies [404, 429, 432]. Findbugs is among the most commonly used ASATs [436], having been downloaded over a million times.

4.7 Towards a new approach

Using pretrained models of code. Recently, Kharkar et al. [231] found that pretrained models of code are able to filter false alarms from Infer, outperforming the use of simple handcrafted features. These approaches work directly on the source code, without the need for extracting handcrafted features. They explore the use of both a zero-shot approach and a finetuned transformer-based model. The transformer-based model that is finetuned on labelled dataset of warnings is their best performing technique. Similar to other approaches [405, 430, 433, 220], Kharkar et al. [231] formulates the task of filtering false alarm as a binary classification problem. After receiving an input string of the code context, their transformer-based model outputs a label that indicates if a warning is a true or a false alarm. Their experiments suggested that language model of code are able to recognize and adapt to code idioms that static analyzers struggle with. However, from the experimental results reported in the paper, the improvements in precision came at a cost of incorrectly filtering out some true alarms.

```
final InputValidateResult<Integer> selectIndex =
    validateUserInputAsInteger(input, entities.size(),
 "You have input a wrong value %s.");
if (selectIndex.getErrorMessage() == null) {
  return wrap(entities.get(selectIndex.getObj() - 1)); // warning
      location
}
return error(selectIndex.getErrorMessage());
```

Listing 4.1: Source code at the warning location. Infer believes that `selectIndex.getObj()` may return `null`.

```
  // (0)
  // InputValidateResult.java
  public static <T> InputValidateResult<T> error(String
      errorMessage) {
  final InputValidateResult<T> res = new InputValidateResult<>();
```

```
res.errorMessage = errorMessage;
> return res;
...
// (1)
// DefaultPrompter.java:
// Taking true branch
InputValidateResult<Integer> selectIndex =
    validateUserInputAsInteger(input, entities.size(), "You have
    input a wrong value %s.");
> if (selectIndex.getErrorMessage() == null) {
    return wrap(entities.get(selectIndex.getObj() - 1));
}
...
// (2)
// InputValidateResult.java:
// return from a call to Object InputValidateResult.getObj()
public T getObj() {
>    return obj;
}
...
// (3)
// DefaultPrompter.java:
if (selectIndex.getErrorMessage() == null) {
>    return wrap(entities.get(selectIndex.getObj() - 1));
}
return error(selectIndex.getErrorMessage());
```

Listing 4.2: Part of the traces reported by Infer. Initially, a `InputValidateResult` is constructed. An `InputValidateResult` has either a non-null `getObj()` or a non-null `getErrorMessage()` Subsequently, the path includes a trace event where both the error message and underlying object is null.

Example. Figure 4.1 shows a simplified example of a program where Infer incorrectly reports a possible null pointer dereference (that is a false alarm). Figure 4.2 shows the path of trace events that Infer analyzed. A false alarm occurs when the static analyzer takes a path that cannot be taken in reality. Such a path would contains at least one trace event that is impossible. Infer may not be aware of code idioms that human developers understand when reading the code. For example, in Figure 4.2, `InputValidateResult` is a class that can either have an error message (`getErrorMessage()` would return a non-null value) or an underlying object (`getObj()` would return a non-null object). Therefore, on taking a branch where `getErrorMessage()`

`== null`, we can expect `getObj()` to return a non-null value. As Infer is unaware of the natural language semantics of the program, it overapproximates the possible paths, and reports a path where both `getErrorMessage()` and `getObj()` are null. Note that these trace events may correspond to source code from other methods (i.e., interprocedural analysis) in files other than the file where the warning is reported on.

Large amount of data. Given enough training data, language models of code are able to learn the code contexts where these overapproximations occur, and can filter out these warnings by recognizing that a warning was reported under a similar code context. However, they rely on a large corpus of training data to be finetuned for this task. Our approach exploits a much smaller number of labelled data. It is able to do so by through the use of in-context learning. Our approach exploits the insights that a single impossible trace event makes a path infeasible, i.e., *to be certain that a path is infeasible, every trace event has to be inspected*, and that the mistakes made by the static analyzer are systematic, i.e., *the same impossible events occur in multiple, different paths*. Through in-context learning, TrailMarker analyzes each test warning considering a few other labelled training warnings. When considered together, the selection of labelled training warnings maximizes the number of trace events shared between the training warnings and the test warning. To reduce the number of warnings that require labelling from an oracle, TrailMarker favours the reuse of existing labels when it constructs a demonstration to the large language model for each test warning.

4.7.1 In-context learning

The effective of large pretrained models has been shown in Software Engineering research [228, 281, 254]. For example, CodeBERT has been shown to be effective for type inference [228], program repair [281, 420], and call graph analysis [254]. These models usually use a transformer [392] architecture and are trained on enormous corpus of data obtained from GitHub. These techniques are often pretrained on the masked language modelling objective, in which they learn to recover masked tokens given a code context. For example, the Incoder [160] model was proposed by Facebook and was trained using a variant of the masked language modelling objective. During pretraining, the models capture a large amount of information about programs and their source code.

Finetuning. To employ a pre-trained model, finetuning has to be performed on the downstream task. This enables the feature representations obtained from the pretrained model to be customized to be more relevant to

the downstream task. For example, Kharkar et al. [231] add a classification head to CodeBERTa, a model similar to RoBERTa [272]. They then tuned the parameters of the classification head while freezing the parameters of CodeBERTa.

In-context learning. Recently, large language models of code have been shown to be few-shot learners. In particular, *in-context learning* [96] has been proposed for large language models of code. Instead of tuning the model parameters, k input-output warnings are provided as a demonstration to the pre-trained model, whose parameters are kept frozen after pretraining. By using the background knowledge already captured during pretraining, the model may be able to recognize latent concepts from the examples in the demonstration. In-context learning transforms classification problems to the next-token prediction task, a classification is generated by inspecting the next token *label*, $label^{test}$, generated by the large language model given the *test instance*, X^{test}, and a series of *training examples*, X^{trg}, and their *labels*, $label^{trg}$. In other words, a demonstration is a tuple:

$(X_1^{trg}, label_1^{trg}, X_2^{trg}, label_2^{trg}, ..., X_k^{trg}, label_k^{trg}, X^{test})$

As X^{test} is not accompanied by its label, in-context learning leverages the ability of the large language model to output a token *label* with the highest predicted probability.

Note that there is a limited budget of providing warnings as a demonstration. The input of the large language model only allows for 2048 tokens to be provided, as such, we cannot provide a large volume of training warnings. Care must be taken to provide informative warnings that are useful for a particular testing instance. In this study, the training examples are snippets of code from the source code on which warnings are reported on and the labels indicate if the corresponding warning was a true alarm ("yes") or a false alarm ("no"), i.e., binary classification.

4.8 Few-shot in-context filtering of false alarms

4.8.1 Overview

We propose TrailMarker, which uses in-context learning and a novel strategy of selecting training examples to be labelled and provided in the demonstrations to the large language model. An overview of TrailMarker is given in Figure 4.6. TrailMarker formulates the problem of selecting training warnings as a set cover problem on the trace events of the warning under classification ((1) in the Figure); it aims to select warnings that

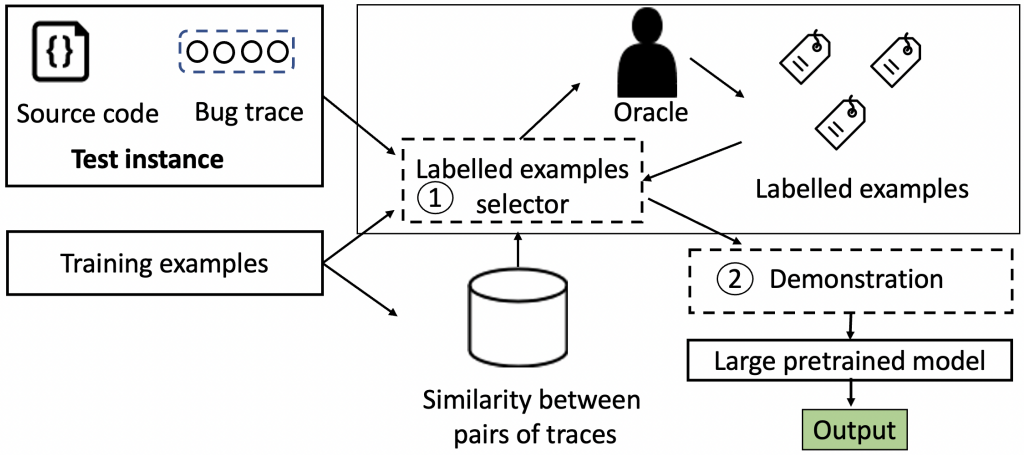


Figure 4.6: Overview of TrailMarker. TrailMarker does not require a fully labelled dataset. Given a warning from the testing dataset, it selects some training warnings that should be passed to the large language model (1). The selected training warnings are passed to the oracle, a human-in-the-loop, for their labels. Next, these warnings form the demonstration which is passed to the large language model (2), which produces the output of TrailMarker.

provide useful information for making predictions about the test instance. By using the 1 billion-parameter model of Incoder [160], which is a large language model of code, TrailMarker performs in-context learning ((2) in the Figure), which enables it to accurately classify a large proportion of the dataset while having access to only a small number of labelled warnings. Using in-context learning and the novel selection strategy, TrailMarker reduces the burden of requiring expensive manually labelled data while providing a high level of effectiveness in identifying false alarms.

4.8.2 Problem Formulation

In this study, the task of filtering false alarms from static analyzers is formulated as follows.

Prior knowledge: A dataset of training warnings, W_{train}. Initially, all warnings are unlabelled. Each training example, W^i_{train}, is comprised of the source code of the method in which the warning are reported on and the path of trace events, E^i_{train}, reported by Infer (see Figure 4.2 for an example).

Input: A warning instance, W_{test}, from the testing dataset. As before, each warning is associated with the source code of the method on which the

warning is reported on and path of trace events, E_{test}.

Output: A boolean label, $label_{test}$, indicating if the warning, W_{test}, was a true alarm or a false alarm.

Oracle: An oracle, which is assumed to be the human-in-the-loop, can add labels to warnings in the training dataset. The oracle can generate labels for multiple training warnings given one test input, i.e., multiple warnings can be labelled at any time in order to produce an output for each input test instance. Once a warning has been labelled, the labels can be subsequently without having to query the oracle again.

The total number of training warnings that are labelled is of interest in this study. Manually labelling warnings is expensive and automatically determined labels are inaccurate [220]. We seek to reduce this cost in this study.

4.8.3 Selection of training warnings

Motivation. Unlike prior studies [96] that randomly select labelled warnings to use a demonstration, TrailMarker adopts a novel strategy of selecting informative warnings. Randomly selected warnings may not share a similar context to the test instance, and the the information provided in the randomly selected warnings may not help in accurately classifying the test instance [269]. Our strategy aims to minimize the number of training warnings that require labels while having a high level of effectiveness.

Set Cover. Given a test warning, which includes the path of trace events, P^i_{train}, that leads to a null pointer dereference, TrailMarker treats the task as a set cover problem. The set cover problem is a classic combinatorial problem to identify the smallest subcollection of items whose union is the entire universe of items. TrailMarker aims to select warnings from the training warnings that contain trace events that overlap with the traces in the test warning. TrailMarker is given the collection of trace events, E_{test}, (in the path, P^i_{train}, that results in a null pointer dereference) of the warning. For each test warning, TrailMarker considers the collection of traces from the warning as a universe of trace events. For each training example, it filter out traces that are not sufficiently similar to any trace in the testing trace (i.e., the trace cannot cover any member of E_{test}). We later elaborate on the computation of the similarity. Then, we apply a greedy algorithm to identify the subcollection of training warnings, $W_{selected}$, associated with set of trace events, E_{train}, that maximizes the coverage the trace events, E_{test}. In this problem setting, there is no guarantee that the traces in the test warning appear or are similar to any trace in the training dataset. Hence, some traces cannot be covered regardless of the selection of training warnings, i.e.,

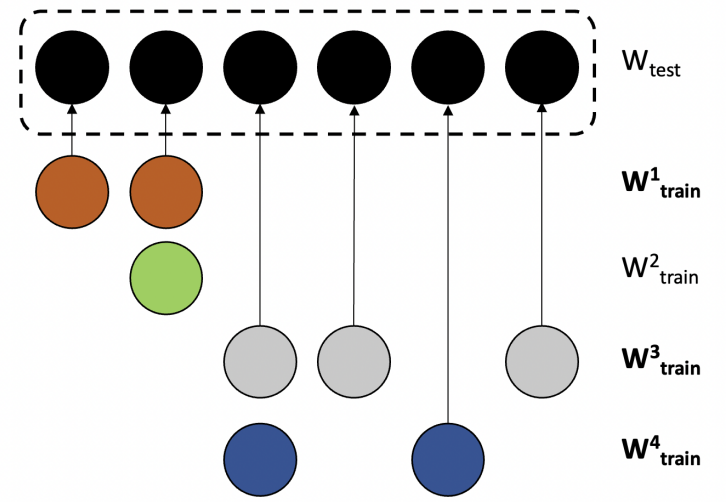


Figure 4.7: The selection of warnings to provide a demonstration is done by selecting warnings to cover the traces in the testing instance. Warnings 1,3 ,4 are selected as they collectively cover all the traces of the test warning.

$|E_{train}| <= |E_{test}|$. The algorithm terminates once progress can no longer be made.

Example. An example is shown in Figure 4.7. In the figure, the testing warning that comprises 6 trace events in the path that leads to a dereference. TrailMarker selects warnings 1, 3 and 4, which are training warnings that contain traces that match the traces in the testing instance. For all the traces to be covered, Warning 1 matches the first two traces, warning 3 matches the 3rd, 4th, and last trace, while warning 4 covers the 5th trace. Note that warning 2 is not selected as the only trace it shares with the testing instance is already covered by warning 1. The set cover strategy closely matches our intuition of the problem; we wish to obtain some information about every trace event as just one impossible trace event is enough to determine that a path is infeasible. As such, our goal is to maximize the number of trace events that are covered by at least one warning instead of covering any particular trace event more than once.

TrailMarker selects examples from the training dataset as shown in **Algorithm 1**. First, the traces associated with the testing warning is retrieved (line 2). Based on *sim*, TrailMarker determines which trace events in the testing trace can be covered by each training trace. It enumerates over every trace event in the training traces, E_{train} (lines 4–15) and checks if the similarity between each trace event and each of the testing trace events are higher than *sim* (lines 7–12). Given any training trace,

Algorithm 1: Selecting k training warnings based on the overlap in their traces and the traces of the test instance.

Inputs:

- $k \leftarrow$ number of warnings to use in a demonstration
- *sim* $\leftarrow$ similarity threshold to determine if two traces are a match
- test_warning $\leftarrow$ the test warning from W_{test}. The source code where the warning is reported and the bug trace can be retrieved.
- trg_warnings $\leftarrow$ the training warnings from W_{train}.
- oracle $\leftarrow$ a function that returns the labels of a warning, e.g. by involving a human-in-the-loop
- similarity(a, b) $\leftarrow$ a function that accepts two traces, returning a value between 0.0 and 1.0
- previously_seen $\leftarrow$ labels, L_{train}, of previously selected warnings

function SelectWarnings(k, *sim*, test_warning, trg_warnings, oracle, similarity, previously_seen)
test_trace $\leftarrow$ traces(test_warning)
trg_traces $\leftarrow$ []
for *trg_warning* $\leftarrow$ *trg_warnings* **do**
  trg_trace $\leftarrow$ {}
  old_traces $\leftarrow$ traces(trg_warning)
  for *old_trace* $\leftarrow$ *old_traces* **do**
    for *test_event* $\leftarrow$ *test_trace* **do**
      if *similarity(old_trace, test_event) > sim* **then**
        trg_trace $\leftarrow$ trg_trace $\cup$ test_event
      end
    end
  end
  trg_traces $\leftarrow$ trg_traces :: (trg_warning , trg_trace)
end
selected_warnings $\leftarrow$ SetCover(trg_traces, test_trace, similarity, *sim*)
if *selected_warnings.length* $< k$ **then**
  selected_warnings $\leftarrow$ selected_warnings :: $k -$ *selected_warnings.length* most similar warnings
end
for *selected_warning* $\leftarrow$ *selected_warnings* **do**
  if *selected_warning* $\notin$ *previously_seen* **then**
    previously_seen $\leftarrow$ previously_seen $\cup$ oracle(selected_warning)
  end
end
return selected_warnings, previously_seen

$E^i_{train} \in E_{train}$, TRAILMARKER considers that E^i_{train} covers a trace event, E^j_{test}, only if their similarity is greater than sim, i.e., $sim(E^i_{train}, E^j_{test}) > sim$. Afterwards, considering the coverage of testing trace events, E_{test}, covered by each warning from the training dataset, W_{train}, a subset of the training warnings are selected by treating the problem as a set cover problem (line 16, later elaborated upon in Algorithm 2). If fewer than k warnings are obtained from the set cover (line 17), more warnings are selected based on the similarity of the method content where the warning was reported (line 18). Finally, if there are any selected warnings that have not been previously labelled, the oracle is queried for their labels (lines 20–24). The algorithm returns the selected warnings and their labels (line 25).

The set cover strategy of selecting labelled warnings is given in **Algorithm 2**. The algorithm returns a list of selected warnings (initially initialized as an empty list in line 3). While not all traces in the test warning have been matched to a trace among the training warnings (line 4), the algorithm greedily selects the next warning (line 6) by picking the warning that covers the most of the remaining uncovered test traces (line 8). If multiple warnings cover the same number of the remaining trace events, warnings that have already been labelled are favoured (lines 13–18). Once the next warning has been selected, the set of covered test traces are updated (lines 23–24), then the loop restarts to select the subsequent warning. If some test traces cannot be covered, the algorithm will fail to make progress as it cannot find a next warning that can improve the coverage. When this occurs, the algorithm terminates (lines 20–22).

Similarity between trace events. Algorithm 2 requires the computation of the similarity between pairs of trace events. This can be obtained by passing each trace event, consisting of a single snippet of code (i.e., the expression on which a null dereference occurs, or a function invocation) and a piece of natural language text describing the occurrence of control flow (e.g. "taking the true branch", "invoking a function call"), into the pretrained large language model, Incoder [160], used in this study. Then, we obtain the representation of the trace by obtaining the hidden state of the last layer of the transformer model. This gives us a vector corresponding to a trace event. After obtaining vectors of both trace events, we compute the cosine similarity between them.

4.8.4 In-context learning

Constructing a demonstration. We apply prompt templates, which are instantiated to be a sequence of tokens provided as input to the large language model. The prompt templates are as follows:

Algorithm 2: The set cover strategy of selecting warnings, $W_{selected}$ from the training dataset warnings to maximize its coverage of the trace events, E_{test} of a testing warning.

Inputs:

- trg_traces ← the trace events of each training example. A list containing sets of trace events.
- test_trace ← the trace events in the testing instance. A set of trace events.
- similarity(a, b) ← a function that accepts two trace events and return a value between 0.0 and 1.0
- *sim* ← similarity threshold to determine if two traces are a match
- previously_seen ← a set of previously labelled warnings

```
function SetCover(trg_traces, test_trace, similarity, sim)
result_set ← {}                          ▷ Tracks which trace events have been covered
result ← []                              ▷ Tracks which warnings have been selected
while test_trace \ result_set ≠ {} do
    add_set ← {}
    selected_warning ← −1
    remaining_traces ← test_trace \ result_set   ▷ trace events that have not been
    covered
    for warning, trg_trace ←trg_traces do
        ▷ Greedily pick the warning that covers the most of the remaining trace
        events
        if | remaining_traces \ trg_trace | < | remaining_traces \ add_set | then
            selected_warning ← warning
            add_set ← trg_trace
        end
        if | remaining_traces \ trg_trace | = | remaining_traces \ add_set | then
            if warning∈ previously_seen then
                selected_warning ← warning
                add_set ← trg_trace
            end
        end
    end
    if selected_warning = -1 then
        break                            ▷ Terminate if further progress is impossible
    else
        result_set = result_set ∪ add_set
        result = result :: selected_warning
    end
end
return result
```

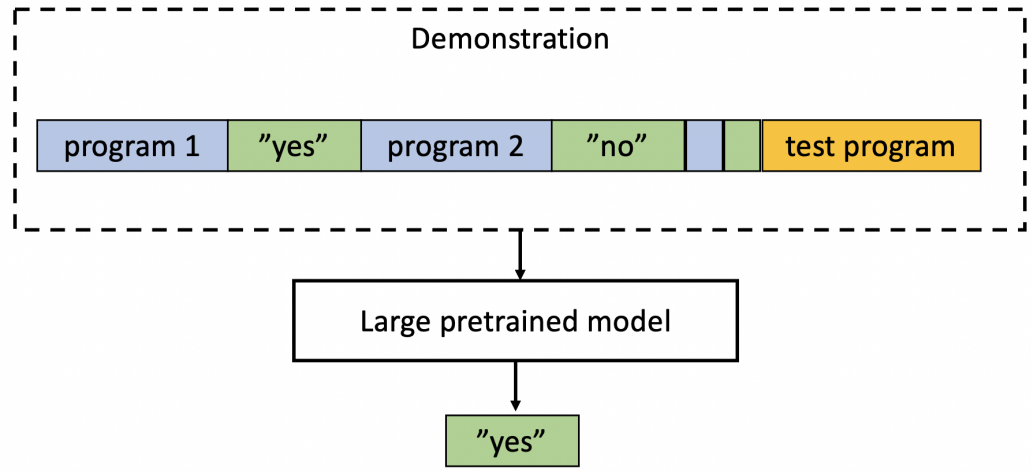


Figure 4.8: Given a demonstration comprising of multiple training warnings and their labels, in-context learning queries the large language model for the label of the test instance.

$warning(W) =$ "$code:$ ‘‘‘X’’’"
$label(x) =$ "$warning: L$."

X is replaced with the content of the method on which the warning W is reported. L is either "yes" or "no", corresponding to the binary labels indicating if the warning was a true or false alarm.

Inference. As seen in Figure 4.8, a demonstration is a sequence of ($warning(W^1_{selected})$, $label(l^1_{selected})$,$warning(W^2_{selected})$, $label(l^2_{selected})$, ...,$warning(W^k_{selected})$, $label(l^k_{selected})$, $warning(W_{test})$), in which k training warnings are provided along with their labels, and a single testing warning, W_{test}, is provided. Once the templates have been instantiated, they are concatenated together. The large language model then completes the line of the provided "warning:" prefix. The predicted label is the prediction l^{test}.

Preprocessing. To maximize the amount of information provided, the source code of each example provided is preprocessed to removed consecutive whitespaces (e.g., the indentation at the start of each line). To fit the limited budget of 2048 tokens, each example is truncated to contain the preceding $2048/(k+1)$ tokens. Thus, k, the number of warnings selected for each demonstration, influences the amount of information provided and the value of k represents a trade-off between the number of warnings and the amount of information given in each example. The inputs to the large language model are tokenized by the tokenizer associated with the particular large language model (i.e., used during their pretraining). The tokenizer uses byte

pair encoding [360], which reduces the size of the vocabulary by breaking up uncommon long tokens into subtokens seen during training, mitigating the issue of out-of-vocabulary tokens.

4.9 Experimental Setup

4.9.1 Dataset

Table 4.11: Statistics of the datasets used

Project	**#LoC**	**# true alarms**	**# false alarms**
Ambry	192,565	10	28
Azure Maven Plugin	29,899	4	30
Azure SDK	259,125	539	1,068
Nacos	200,053	12	50
total	681,642	565	1,176

As the dataset of Kharkar et al. [231] is not publicly available, we constructed a new dataset of Java programs based on the projects used in their dataset. We omitted the Playwright project as it was not a Java project. The other two projects used in the experiments of the Kharkar et al. [231] study were internal projects that are not publicly available. Table 4.11 shows the details of the dataset. All of these projects are under active development and are widely used. We ran Infer on the revisions of the project indicated in the Table. We labelled each warning produced by Infer. In total, we obtained 565 true alarms and 1,176 false alarms. In our experiments, using a 75%/25% split, we obtain a training and testing dataset.

4.9.2 Baselines

To assess and understand the effectiveness of TrailMarker, we use the following baseline approaches in our experiments:

CodeBERTa. Similar to Kharkar et al. [231], a pretrained CodeBERTa model is finetuned on a fully labelled training dataset. The model is finetuned for 5 epochs. Based on the validation dataset, the best-performing model is selected.

CodeBERT. While the DeepInferEnhance approach of Kharkar et al. [231] is not publicly available, we trained a CodeBERT model that is similar to DeepInferEnhance. A classification head is added to the base CodeBERT

model ("microsoft/codebert-base") and is finetuned on our fully labelled training dataset. The model is finetuned for 5 epochs. Based on the validation dataset, the best-performing model is selected.

Zero-shot. We use the Incoder model based on the description of the zero-shot learning approach described by Kharkar et al. [231]. The GPT-C [374] model used by Kharkar et al. is not publicly available. However, the improvements we obtain using Incoder compared to the strawman baseline are consistent with the improvements of GPT-C over the strawman. As in any zero-shot approach, we do not train or finetune the model further. Instead, the content of the source code where a warning is reported on is provided to the pretrained model. We truncate the content up to the point where the warning is reported and add an incomplete statement as a *prompt* to the model. Next, we process the output generated by the Incoder model as it generates code based on the context. This approach relies on the intuition that many null dereference warnings are addressed by the addition of a null-check before the dereference occurs. Similar to Kharkar et al. [231], we add seven prefixes of possible null checks (e.g. *if (*, *assert*) as prompts and obtain five completions considered by the language model to be the most likely completions. If any of the completions produced by the model contains the term *null* or *NULL*, then we consider that the model predicts the warning is a true alarm.

Strawman. Prior research [220] has shown the importance of comparing newly proposed approaches against a strawman approach. In this task, one strawman approach simply predicts that all warnings are true alarms. In other words, this is the true performance of the static analyzer. If a proposed approach of postprocessing the warnings underperforms the strawman approach, then it implies that the approach did not improve over the use of Infer without any postprocessing.

TrailMarker$_{randomselection}$. In an ablation analysis, to validate the utility of using the set cover strategy of selecting warnings, we compare TrailMarker against a baseline that randomly selects warnings. In-context learning usually randomly selects warnings [96].

TrailMarker$_{voting}$. In an ablation analysis, to validate the utility of using in-context learning, we compare TrailMarker against a baseline that produces its output based on voting strategy of the labelled warnings selected by the set cover strategy.

TrailMarker$_{methodcontent}$ In an ablation analysis, to validate the utility of analyzing the traces in the path to select warnings for demonstration, we compare TrailMarker against a baseline that selects warnings based on the similarity of their method content where the warning was reported.

TrailMarker$_{similartraces}$ In an ablation analysis, to validate the utility

of using the set cover strategy of selecting warnings, we compare TrailMarker against a baseline that selects warnings based on warnings with similar traces.

4.9.3 Evaluation Metrics

We use widely used, standard metrics in this study that were used in prior studies [231, 220]. Due to prevalence of false alarms, the true warnings are the minority class in our task. A true positive (TP) is a warning correctly predicted to be a true warning. A false positive (FP) is a false alarm incorrectly predicted to be a true warning. We use the term false alarm to refer to warning of a null dereference that cannot occur in reality. A false positive, on the other hand, refers to a false alarm that is incorrectly determined to be a true warning. A false negative (TN) is a true warning incorrectly determined to be a false alarm.

We compute Precision and Recall as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$$

Next, we compute the harmonic mean of Precision and Recall, F1, which captures the trade-off between Precision and Recall. F1 is used in place of accuracy given an imbalanced dataset where the instances of majority class occurs much more frequently than the minority clas. F1 is computed as follows:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision+Recall}}$$

Finally, an objective of this study is to reduce the number of training warnings that require labels, which are costly to obtain in reality. Hence, we report the number of labelled warnings required by the approaches.

4.9.4 Research Questions

Our experiments are guided by the following research questions:

RQ1. How effective is TrailMarker? We investigate to what extent is TrailMarker able to distinguish true alarms from false alarms produced by Infer. We compare TrailMarker against other approaches using pre-trained models of code, as well as the trivial strawman baseline. We assess

Table 4.12: Experimental results showing the effectiveness of the approaches. TrailMarker achives the highest F1 (87.1%) while requiring the least number of labelled warnings (335).

Approach	Precision	Recall	F1	# labels
`Strawman`	31.6%	**100%**	48.0%	N/A
`Zero-shot`	31.7%	97.5%	47.9%	**0 (0%)**
`CodeBERT`	76.9%	89.8%	82.8%	867 (67%)
`CodeBERTa`	76.1%	88.3%	81.8%	867 (67%)
TrailMarker	**85.8%**	88.3%	**87.1%**	335 (25.7%)

the approaches based on the effectiveness in identifying false alarms as well as its number of labelled training warnings required.

RQ2. Which components of TrailMarker contributes to its effectiveness? The improvements by TrailMarker stems from its use of in-context learning and the novel strategy of labelling only the training warnings selected through solving the set cover problem. In this research question, we analyze how each technique contributes to TrailMarker.

RQ3. How do the parameters of TrailMarker influence its effectiveness? TrailMarker has several parameters that may affect its performance, including the number of warnings provided in the demonstration as well as the similarity threshold that determines if a particular trace matches a trace from the test instance. We analyze how modifying the values of the parameters influences effectiveness of TrailMarker.

4.10 Experimental Results

4.10.1 RQ1. On the effectiveness of TrailMarker

We evaluate the effectiveness of TrailMarker by comparing it against the state-of-the-art approaches proposed by Kharkar et al. [231]. Table 4.12 shows the results of our experiments. In the study of Kharkar et al. [231], the best performing model is the finetuned transformer-based model. In our replication of the zero-shot approach, we find that it does not always outperform the strawman approach which always predicts that each warning is a true warning (F1 of 48.0% vs 47.9%). While some false alarms are filtered out, a similar number of true alarms are also removed. This suggests that there may not be any advantages to deploying the zero-shot approach. This result is consistent with the findings of the Kharkar et al. [231] paper,

Table 4.13: An ablation analysis using several baselines. Each baseline is identical to TrailMarker except in one component. TrailMarker$_{voting}$ replaces the use of in-context learning with a voting strategy of predicting labels. TrailMarker$_{randomselection}$, TrailMarker$_{methodcontent}$, and TrailMarker$_{similartraces}$ replace the the set cover strategy.

Approach	**Prec.**	**Recall**	**F1**	**# labels**
TrailMarker	**85.8%**	**88.3%**	**87.1%**	335 (25.7%)
Voting	78.9%	21.9%	34.3%	335
Random selection	41.7%	32.8%	36.7%	841
Method content	76.7%	83.5%	79.9%	433
Similar traces	39.6%	73.7%	51.5%	**154**

where they found the increase in Precision (15.1%) is also accompanied by a decrease in Recall of a similar magnitude (11.7%). We successfully replicate the finetuned transformer-based approach using both CodeBERTa and CodeBERT. After finetuning CodeBERTa, we obtain an F1 of 81.8%. On finetuning CodeBERT, we obtained a stronger baseline performance of 82.8% F1 on a precision of 76.9% and 89.8%. Both finetuned models outperform the strawman approach.

Next, we find that TrailMarker outperforms the finetuned CodeBERT model, improving on F1 by 4.3%. Most of the improvements come from the increased precision, increasing from 76.9% to 85.8%, or an improvement of 8.9%. Importantly, TrailMarker has an increased effectiveness while requiring only a fraction of the training data used to finetune CodeBERT. TrailMarker requires only 335 (or 25.7%) of the training dataset to be labelled. In contrast, finetuning CodeBERT involves 975 training warnings, with N warnings held out as a validation dataset used to determine when to stop the finetuning.

Answer to RQ1:
TrailMarker outperforms the prior state-of-the-art approach, a finetuned CodeBERT, by 4.3% in F1 while requiring just 25.7% of the training dataset to be labelled.

4.10.2 RQ2. On the components of TrailMarker

We assess the two component of TrailMarker, as seen in Table 6.8. TrailMarker uses in-context learning and a strategy of selecting examples to label by viewing the coverage of the trace events as a set cover problem.

To determine if the in-context learning was essential to the technique, we dropped the use of the large language model of code, and replaced it with a nearest neighbor algorithm. This assesses if the strategy of selecting informative warnings was already enough for an effective detector. The performance of TrailMarker decreases from 87.1% to just 34.3%, indicating that the use of in-context learning was important.

Next, we assess if the set cover strategy of selecting warnings was helpful to the approach and if the use of in-context learning was sufficient for high effectiveness. We replaced the set cover strategy by random selection, as is usually done in other studies using in-context learning [96]. We find that the F1 decreases from 87.1% to just 36.7%. This shows that the set cover strategy was essential to TrailMarker to select informative warnings from the training dataset. The reduction in the number of labelled warnings required from 841 to 335 (39.8%) also indicates the amount of effort required saved through the use of TrailMarker.

Apart from randomly selecting warnings, another strategy to select examples would be to pick the warnings that have the most similar traces to the traces from testing warning. However, we find that doing so would reduce the F1 from 87.1% to just 51.5%. This validates our intuition guiding the set cover strategy that maximizing information for every trace event is essential to constructing a good demonstration for in-context learning.

While TrailMarker considers the similarity of traces to identify the warnings to include in the demonstration, another strategy is to pick warnings based on the similarity of the method content at the location of the warning. In this strategy, the top k warnings reported at locations most similar to the source code at the test warning location is selected. This causes a reduction of 7.2% in F1 while increasing the number of required labelled warnings from 335 to 433. Overall, the decreased performance suggests that the use of traces to identify informative examples was important.

Answer to RQ2: Both the use of in-context learning and the set cover strategy of selecting labelled training warnings were essential to TrailMarker. Without in-context learning, F1 decreases from 87.1% to 83.0%. Without the set cover strategy, F1 decreases to 36.7%.

4.10.3 RQ3. On the parameters of TrailMarker

TrailMarker is parameterized by k, the number of labelled warnings to include in a demonstration, and sim, the similarity threshold used to determine if two traces are sufficiently similar to be considered a match. We

Table 4.14: Experimental results when varying the number of examples used in a demonstration, k.

Number of examples, k	**Precision**	**Recall**	**F1**	**# labels**
1	82.4%	85.4%	83.9%	**227**
3	**85.8%**	**88.3%**	**87.1%**	335 (25.7%)
5	78.1%	86.1%	81.9%	485 (37.3%)
7	76.1%	86.1%	80.8%	578 (44.4%)
10	79.2%	86.1%	82.5%	690 (53.0%)

Table 4.15: Experimental results when varying the similarity threshold, *sim*, which controls for the selection of traces that match one another.

Similarity threshold, *sim*	**Precision**	**Recall**	**F1**	**# labels**
0.95	83.2%	83.2%	83.2%	369 (28.4%)
0.9	**85.8%**	**88.3%**	**87.1%**	335 (25.7%)
0.8	82.9%	**88.3%**	85.5%	329 (25.3%)
0.7	83.8%	86.9%	85.3%	**324 (24.9%)**
0.6	83.0%	82.5%	82.8%	**324 (24.9%)**

investigate how the values of these parameters influence the effectiveness of TrailMarker. Table 4.14 shows the experimental results of varying k. A larger value of k would mean that more warnings are provided in the demonstration, but to fit the limited budget of 2048 input tokens to the large language model, a greater number of tokens from each example would be truncated. Moreover, including a larger k would mean that less informative warnings would be provided as a demonstration, which may be a form of noise that reduces the effectiveness of in-context learning. By increasing k, both the Precision and Recall of TrailMarker decreased. Varying k from 5 to 10 does not change Recall. Overall, the F1 decreased from 87.1% down to 80.8% while the number of labels required increased as k increases.

When varying *sim*, we found that F1 decreases as the value of *sim* decreases while the number of labelled warnings decreases. Decreasing *sim* makes it more likely that a previously labelled example can be used to cover the traces of a test instance, however, it also increases the possibility that a poorly matched warning, which may be less informative, training warning is used in the demonstration for in-context learning. F1 decreases down to 85.3% as *sim* is reduced to 0.7 and the number of labelled warnings needed decreases slightly from 335 to 324. Overall, the results suggest that there are only small amount of savings in terms of the number of labels to be gained

from decreasing *sim* while having a slight decrease in F1.

Answer to RQ3: Increasing k leads to both a lower level of effectiveness and an increased number of labelled warnings required. Varying *sim* leads to a greater number of labelled warnings required while slightly decreasing its effectiveness.

4.11 Discussion

4.11.1 Sample efficiency

Our experimental results indicates that TrailMarker performs effectively given a small amount of labelled data. The experimental results validate our design decisions. In particular, our experiments show that the use of traces enable a small amount of labels compared to the use of the method content as the warning context. Using the method content as the context, TrailMarker requires 433 instances compared to just 335. Moreover, the use of the set cover strategy for selecting warnings is a large improvement over a random selection of warnings, reducing the number of labels required from 841 to 335.

While we have analyzed TrailMarker and found is effective given a small amount of data, we perform a more in-depth analysis of using CodeBERT. We extract a training dataset of the training instances selected using the set cover strategy, and finetuned CodeBERT only on these warnings. The results are shown in Table . We find that the performance of CodeBERT drops substantially. Its F1 decreases from 82.8% to 76.4%. In comparison, as previously seen, TrailMarker achieves an F1 of 87.1% while using the same limited number of training warnings.

4.11.2 Implications

The set cover strategy was motivated by the the insight that identifying an infeasible path requires the checking if the path contains one impossible trace event. In our experiments, we find that the set cover strategy improves over both a random selection of warnings and the selection of the warnings with the most similar traces. This validates our insight that obtaining *some* information about each trace event is helpful.

Our insight that the false alarms are systematic and recurring allows us to exploit already labelled warnings during the selection of warnings. To understanding the effect of favouring already labelled warnings, we run one

more experiment where the set cover strategy no longer favours previously labelled example. This substantially reduces the number of labelled warnings required. If TrailMarker does not favour already labelled warnings, the set cover strategy requires 415 labelled warnings. Moreover, F1 *slightly* decreases from 87.1% to 85.6%, which suggests some benefits to effectiveness from the reuse of already labelled warnings.

4.11.3 Qualitative analysis

We qualitatively analyze our results and discuss two case studies to better understand the false alarms reported by Infer.

```
1  if (!TakeContinuationToken.tryParse(topContinuationToken,
       outTakeContinuationToken)) {
2      String message = String.format("INVALID JSON in
           continuation token %s for Top~Context",
3              topContinuationToken);
4      CosmosException dce = BridgeInternal.createCosmosException(
5          HttpConstants.StatusCodes.BADREQUEST, message);
6      return Flux.error(dce);
7  }
8
9  takeContinuationToken = outTakeContinuationToken.v;
10 if (takeContinuationToken.getTakeCount() > topCount) {
11 ...
```

Listing 4.3: One reason for false alarms appears to be Infer's inability to precisely reason about conditional nullness. In this example, Infer reports that takeContinuationToken could be `null`. However, if `TakeContinuationToken.tryParse` fails to set outTakeContinuationToken to be non-null, the program would have taken the error path leading to `return Flux.error(dce)`

Listing 4.3 shows an example where the static analyzer reports a false alarm. The code uses a parser for a Domain Specific Language[1]. A possible null pointer dereference of `takeContinuationToken` is reported on line 10. The variable `takeContinuationToken` was assigned to `outTakeContinuationToken.v` on line 9. However, had `outTakeContinuationToken.v` been null, the error branch on lines 2–6 would have been taken as `tryParse` would have returned null, and line 9 would not be reached. Our approach predicts the warning to be a false alarm because the idiom (of `tryParse` returning false if the `.v`

[1] `https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/samples-java#query-examples`

field of its second parameter failed to be set to a non-null value) would have been used multiple times (for parsing and processing each type of token). As a warning is reported for each time this idiom was used, TrailMarker exploits the recurrence of similar code context and requires only just one such warning to be labelled.

```
public DnsRecordSetImpl withoutIPv6Address(String ipv6Address) {
    this.recordSetRemoveInfo.aaaaRecords().add(new
        AaaaRecord().withIpv6Address(ipv6Address));
    return this;
}
```

Listing 4.4: The static analyzer reports that `aaaaRecords()` may return null. However, this is extremely unlikely as `recordSetRemoveInfo` always initializes a non-null `aaaRecord` when it is constructed. The use of traces enables our approach to determine that this context is identical to Listing 4.5

Listing 4.4 shows the source code of one warning reported by the static analyzer. The code is related to DNS records ("AAAA" and "SRV" are DNS record types). While the warning flags `aaaaRecords()` as potentially returning null, this is unlikely as the constructor used in instantiating `recordSetRemoveInfo` sets the field to be non-null. This warning is similar to the warning in Listing 4.5. As TrailMarker considers the traces when selecting example warnings, it selects Listing 4.5 and its label ("no") as part of its demonstration. Notice that both methods are short, have little overlap in the identifiers used, and have different intents (one modifies a single field, the IP address (`ipv4Address`), and the other modifies multiple fields such as the port and priority). Determining that the relatedness of the warnings presents a challenge when considering only their source code. This validates our motivation of considering the traces to select warnings for a demonstration.

```
public DnsRecordSetImpl withoutRecord(String target, int port, int
    priority, int weight) {
        this.recordSetRemoveInfo
            .srvRecords()
            .add(new SrvRecord().withTarget(target).withPort(port)
            .withPriority(priority).withWeight(weight));
}
```

Listing 4.5: The static analyzer reports that `srvRecords()` may return null. However, this is extremely unlikely as `srvRecords` is always initialized to be non-null in the constructor invoked when `recordSetRemoveInfo` is initialized.

4.11.4 Threats to Validity

Threats to External Validity. Our study considers only one large language model of code and the results of our experiments may change depending on the choice of large language model used. However, Another possible threat is related to the size of the dataset. Our dataset is small compared to other datasets used for classification tasks as it is challenging and costly to obtain labels for each warning. However, the dataset includes warnings from multiple projects from different domains. As such, we believe that experiments on the dataset should generalize to static analyzer warnings on other projects.

Threats to Construct Validity. As we have used the same evaluation metrics as prior studies, we share the same threats to construct validity as prior work. These metrics are standard for evaluating approaches that perform classification. As such, we believe that there are minimal threats to construct validity.

4.12 Related Work

Null pointer dereferences. Null pointer dereferences are a common class of bugs that can lead to severe consequences, including security weaknesses. A range of static analyzers include rules and and detectors for finding potential null pointer deferences. These tools include Infer [137] and FindBugs/SpotBugs [75], among other approaches [135, 313, 78]. An empirical analysis found that all tools fail to find the majority of null pointer exceptions in the real world, while generating a large number of warnings [385]. This indicates that these challenges are foundational, and provide motivation for our research in filtering out the false alarms.

Many studies have performed retrospectives of the state-of-the-art for various Software Engineering tasks. Some papers [441, 264, 273, 192, 173] study the limitations of existing tools, and others [265, 334, 221] assess the applicability of the tools when applied to situations to a different setting from the original experiments. Our study not only uncovers limitations of the Golden Features, but investigates the performance of the Golden Features under different settings (a different warning oracle in our study).

Other studies have shown the need to carefully consider data used in experiments [387, 54, 452, 218]. Similar to Allamanis et al. [54], we show that data duplication may cause overly optimistic experimental results. Our work is similar to the work of Tu et al. [387] in highlighting the problem of data leakage, where information from the future is used by a classifier. Similar to

Kalliamvakou et al. [218], we suggest that researchers should be careful about interpreting automatically mined data. Our analysis indicates that there may be delays before developers inspect static analysis warnings. Related to this, Zheng et al. [452] found that the status of many issues in Bugzilla may only be changed after large delays. These delays have implications for heuristics that are used to automatically infer labels from historical data (in our case: if a warning is actionable). Kochhar et al. [238] investigated three types of bias that affect datasets used to evaluate bug localization techniques.

Improving static analyzers. False alarms may be caused by bugs in the implementation of the static analyzers. Some researchers propose techniques for differential testing to detect bugs among static analyzers that report warnings of the same types of bugs [402]. Other studies propose methods of inspecting patches from open-source repositories for the manual design of bug detection rules [298, 297].

4.13 Summary

In our replication study study, we show that the problem of detecting actionable warnings from Automatic Static Analysis Tools is far from solved. In prior work, the strong performance of the "Golden Features" were contributed by data leakage and data duplication issues.

Our study highlights the need for deeper study of the warning oracle to determine ground-truth labels. By changing the reference revision, different conclusions about performance of the Golden Features can be reached. Furthermore, the oracle produce labels that human annotators and developers of projects using static analysis tools may not agree with. It highlights the need for community effort to build a reliable benchmark and to compare newly proposed approaches with strawman baselines. A replication package is provided at https://github.com/SA-retrospective/study.

Next, we propose a new approach of filtering false alarms from static analyzers. Our proposed approach, TrailMarker, uses in-context learning using a large language model of code and a strategy of selecting informative warnings for use in the demonstrations for in-context learning. Unlike existing techniques, TrailMarker requires substantially fewer training warnings while achieving a higher F1.

Chapter 5

(Dynamic Analysis + API) Adversarial Specification Mining

5.1 Overview

Researchers have proposed many techniques to automatically infer specifications and usage models, frequently in the form of Finite State Automata (FSA). These models represents the possible transitions between program states, and shows how to reach a state from another. These techniques require execution traces of the software as input. It is assumed that these traces are representative of the software and that all correct behavior are reflected in these traces. Unfortunately, it was found that automatically mined specifications are still inaccurate [255]. One reason for this may be that the traces used to construct these specifications are not representative and are not sufficiently diverse. Other researchers have proposed techniques [121, 99] to determine if enough traces have been seen, but these techniques only consider the traces that have already been seen and metrics are computed only over the observed traces. They are unable to reason about execution traces which are possible but are uncommon.

Test generation may be a way to generate new tests that specification miners can learn from, and in several studies, researchers have used test generation to mine specifications. Tautoko [126], for example, refines a FSA-based specification by generating tests to cover missing transitions. Deep Specification Miner (DSM) [252] leverages random test generation to produce a large number of traces to learn language models from. Still, as we investigate later in this study, these techniques are not sufficient for produc-

ing highly accurate models. DSM relies on randomized test generation, and even when provided with traces of uncommon usage, it is not able to leverage these traces to produce more accurate models. Tautoko relies on methods that reveal the state of an object to detect a state change, which may limit it from working effectively for all types of objects. As such, we hypothesize that existing test generation strategies do not completely address the problem of uncommon usage patterns.

For ensuring that uncommon usage is represented, we propose a process which we term *adversarial specification mining*. In the first phase, we mine specifications from traces collected from running a set of test cases of the software under test. In the second phase, test generation is guided towards the discovery of counterexamples of the mined specifications. In the third phase, a specification miner uses the new counterexamples to construct an accurate model. We developed a prototype, *DICE* (**Di**versity through **C**ounter-**E**xamples). DICE mines FSA models through an adversarial specification mining process. For the purpose of inferring a more accurate model, DICE produces more example execution traces given an initial set of temporal specifications, aiming to find inaccuracies in them. This is done through a search-based test generation process is adversarial to the input specifications, searching for tests that exercise the software under test in ways that the input specifications would not accept as correct usage. DICE contains two main components: **DICE-Tester**, which drives test generation towards uncommon patterns, and **DICE-Miner**, which converts execution traces into a Finite-State Automata (FSA) model. This is the first study that makes use of search-based testing for mining specifications.

DICE-Tester uses a search-based testing framework, Evosuite[158], guiding it towards the generation of counterexamples of the input specifications by representing traces that will falsify the specifications as search goals. The modifications made by DICE-Tester prevents the search algorithm in Evosuite from getting caught in a local optima, enabling Evosuite to efficiently search for counterexamples. We use the DynaMOSA algorithm, introduced in a previous study [311], to allow Evosuite to dynamically select objectives instead of trying to achieve pareto optimality.

To characterize a set of traces, we first mine specifications as properties in Linear Temporal Logic (LTL), a formalism of constraints on event-ordering, that hold on the traces. Prior work has shown the relationship between LTL and specification mining [142, 250, 85, 86, 428], and has applied data mining techniques to infer temporal properties [251, 103]. Still, automatically mined properties are typically not completely accurate. As such, this motivates work on boosting the accuracy of identifying temporal properties. Adversarial specification mining solves this problem by filtering out temporal

properties that can be invalidated. In this study, we use six LTL property templates introduced in previous studies [250, 372, 85]. However, we propose a reformulation of three properties, in which we use knowledge of method purity derived from low-cost heuristics and static analysis, to reformulate the properties, tackling shortcomings of the properties described in a recent study [372]. In this work, we use the terms "pure" and "side-effect-free" interchangeably.

As a result of guiding test generation to search for counterexamples, the traces collected by DICE-Tester include uncommon, but correct, usage patterns of the library. To infer an FSA model, we use the traces and temporal properties in a FSA inference algorithm that we propose. We borrow insights from prior work [128, 242], characterizing the states in an FSA based on the methods that are enabled and which can be invoked from it. We make two observations of limitations in existing model inference algorithms and modify our algorithm to address them. In our evaluation of DICE, we find that the models produced by DICE outperform models from existing specification miners, such as the state-of-the-art specification miner, Deep Specification Miner [252]. Finally, we compare DICE and DSM by using the models they infer in server fuzzing, and we find that the models learned by DICE helps in increasing line and branch coverage on an FTP server.

5.2 Background

5.2.1 Specification Mining

K-tails and its variants. Many specification mining algorithms have been proposed. To infer FSA, specification mining algorithms have to provide abstractions over states, and determine if two traces result in the same state. A classic algorithm that infers a Finite State Automaton from traces is the k-tails algorithm [62]. The k-tails algorithm [62] first uses the input execution traces to build a Prefix Tree Acceptor (PTA). A PTA is a tree-like deterministic finite automaton (DFA) where states are grouped and merged based on the prefix that they share. This automaton is consistent with the input traces and will accept all of them. Next, the algorithm merges states that have the same sequences of invocations in the next k steps. The value of the parameter, k, can vary. This trades off precision and recall; a small value of k results in more spurious merges while a large value of k leads to lower generality.

Studies have extended the traditional k-tails algorithm. These studies often keep the first step of the original k-tails algorithm, using a PTA to

create a tree-like automaton that directly represents the input traces. These algorithms thus inherit the assumptions made by k-tails in its first step, that states with the same prefix are equivalent, typically only modifying the second step of the k-tails algorithm, changing the equivalence criteria of states before merging them.

Lo et al. [275] propose to mine temporal rules that hold over the input traces and prevent any merge that will result in a violation of the rules. Lorenzoli et al. [277] introduce GK-tail, which mines extended FSA where transitions are labelled not only with method calls, but includes parameter values. They introduce different merging criteria, including criteria that do not require exact matches of the transitions, and allow for more general conditions of the parameter values. Krka et al. [242] introduce multiple algorithms in their work, including SEKT, which extends k-tails by adding another condition for equivalence: States are merged only if they correspond to the same abstract state, which are defined by the invariants extracted by Daikon. Le et al. [252] propose a deep-learning based approach, Deep Specification Miner (DSM), to determine if a set of states are equivalent. They train a Recurrent Neural Network-based model to produce features characterizing each state in a high-dimensional space. After clustering the states in this space, states are merged according to the clusters they belong to. Therefore, each cluster is mapped to a single state in the output FSA. In this study, states are characterized by a feature vector built for each state, which includes the likelihood of each possible transition label based on their prefix.

State abstraction. While k-tails and its variants combine states based on their prefixes and an equivalence criteria, other approaches have proposed other methods to infer the states in an FSA. de Caso et al. [128] propose CONTRACTOR, which uses program invariants to characterize states of an FSA based on the *enabledness* of methods. A method is enabled if the invariants of the state hold. States are thus a combination of enabled methods, where the pre-conditions of the methods are consistent with one another. CONTRACTOR was proposed as a method to validate pre- and post-conditions specifications by presenting a state machine abstraction of the specifications. The finite state machine help in revealing potential inaccuracies among the pre- and post-conditions. Constructing all possible combinations of enabledness of the methods result in number of states exponential to the number of methods. To avoid this state space blowup, CONTRACTOR models the dependencies between method enabledness to reduce the number of states. Afterwards, only the states reachable from the initial state are retained. Krka et al. [242] enhance the CONTRACTOR model by proposing CONTRACTOR++, filtering invariants inferred

by Daikon [146] and including the output value of method invocations in the labels of transitions.

Finally, approaches such as ADABU [127] and Tautoko [126] identify a set of inspectors for each class. Inspector methods are heuristically identified based on their return type (not void), a lack of parameters, and the lack of side-effects. Abstract states are characterized by the return values of these inspector methods, which are abstracted over to prevent a large number of states. For example, an integer return value is abstracted into one of three abstract values based on its relative value to 0 (either >0, $= 0$, < 0). These approaches may not perform well in the absence of inspectors.

Temporal properties. Several techniques have shown the use of temporal properties in inferring FSA from traces [142, 250, 85, 86, 428]. As mentioned earlier, Lo et al. [275] use temporal properties to prevent erroneous merges. Data mining has been used for the identification of these rules; however, the number of false positives of inferred rules can be high, motivating the need for better ways to identify temporal rules [380]. Le et al. [251] have studied the use of different interestingness measures, while Cao et al. [103] proposed the use of learning-to-rank algorithms composing different interestingness measures to identify accurate properties. Le et al. [250] have also built a meta-model, SpecForge, over existing algorithms in order to decompose mined FSAs into temporal rules, and recompose selected ones back into an FSA. In recent work, Sun et al. [372] used crowdsourcing for identifying correct temporal properties, however, this process was not done automatically, relying on human annotators.

5.2.2 Test Generation for Specification Mining

Test generation for specification mining have been studied previously. Xie and Notkin [422] propose a feedback loop between specification inference and test generation. However, there is no publicly available version of a tool that implements this strategy and this strategy was not empirically evaluated. Tautoko [126] uses test generation to further refine a specification. Tautoko mutates an initial test suite and a given FSA model to find missing transitions in the FSA model. DSM [252] uses Randoop [306], which performs randomized test generation, and traces are collected from the test cases generated to train a Recurrent Neural Network on.

The above studies use randomized testing or mutate an existing test suite for mining specifications. These test generation techniques do not systematically diversify the test suite or use any strategy to ensure sufficient diversity in the test cases.

5.2.3 Search-based test generation

In this study, we use search-based test generation to create test cases to learn specifications from. The generation of test cases are guided towards search goals that we define. We opt to use a search-based test generation tool, Evosuite [158]. Evosuite is a unit test generation tool for Java that uses a evolutionary approach to search for high-quality test cases that fulfils a specified set of coverage criteria. Evosuite evolves a population of tests through multiple generations, and in each generation, discards tests that are less fit while mutating surviving tests. This acts as a search process that iteratively improve the test cases to cover the search objectives. Many optimizations have been proposed and implemented in Evosuite since its inception [67, 346]. Evosuite is automated and does not require any manually written tests as input. Developers can extend the search algorithm or add new coverage and fitness goals. Evosuite comes with a variety of coverage goals, ranging from structural coverage to method coverage goals. Structural coverage goals include line coverage and branch coverage, while method coverage goals guide Evosuite towards tests that invoke every constructor and method of the class. We select Evosuite instead of alternative tools due to its strong performance among state-of-the-art test generation tools [289].

Multiple objective formulation. In the past, test case generation focused on optimising for various coverage criteria independently of each other. Recently, Rojas et al. [345] generated tests while optimising multiple objectives simultaneously, aggregating fitness functions through a weighted sum. However, other studies show the limitations of aggregating multiple fitness goals as a single measure. For example, one such limitation is that the weighted sum aggregation assumes that each fitness goal is independent of each other, which is not true of structural coverage goals (for example, conditional dependencies mean that line and branch coverage goals may depend on one another). Instead of optimizing tests towards a single aggregated fitness value, other researchers have applied multi-objective search algorithms [310]. These algorithms presents several advantages, including preventing the search process from getting stuck in a local minima, and can generate high quality test cases [310]. Indeed, Gay [167] showed that optimising for multiple objectives at the same time instead of enumerating through the objectives one by one lead to test suites that better detect faults.

There are problems specific to test generation when formulated as a multi-objective search. When faced with a large number of search goals, it is impossible to rank many of the individual test cases when considering all of the goals. Due to this, the search process may degrade to become essentially random [310]. Another problem is that a test case that may be fit, when con-

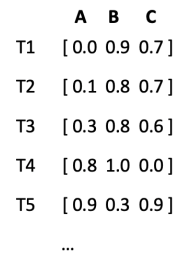


Figure 5.1: Example of several objective function vectors of test cases that are non-dominated. There may be more than a few non-dominated test cases and each of them have an equal chance to get included. We use DynaMOSA to address this problem.

sidering every search goal, but may not fully cover any individual search goal. In other words, although the multi-objective formulation of test generation may produce test cases that are pareto-optimal, with the tests representing optimal trade-offs between fitness goals, it may not produce a resulting test suite that completely covers an objective. This is detrimental to our study as we require test cases that contradict a temporal specification, instead of just *being close* to covering it, regardless of the number of other fitness goals the tests are close to covering. Such a set of test cases will provide us with no value. For example, given several test cases which are scored as the vectors shown in Figure 5.1, all the tests are non-dominated (each test is no worse than another with respect to at least one search goal). In this example, the individual values in the vector represent the distance for a particular search goal. A lower distance is better and a distance of 0.0 indicate that the test case covers that goal. While objective A is covered by test T1 and objective C by T4, objective B is not covered by any test. As none of the tests dominate each other, they have equal probability of getting selected for the next generation. Objective B is a difficult objective to cover. In our study, it is important that we retain and evolve test case T5 in the next generation as it is closest to covering objective B.

To address these problems, the DynaMOSA multi-objective algorithm [311] has been proposed for Evosuite. The DynaMOSA algorithm, at a high-level, is given in Algorithm 3. It evolves an initial randomly generated population of tests through multiple generations. Given a set of coverage goals, the algorithm evolves a population of test cases through the usual mutation and cross-over operators (line 5). This gives us the offspring, Q, a set of test cases containing new tests as well as retaining some test cases from the previous generation. The test cases in Q are ranked and binned into a list of fronts, which partitions Q. The first front contains the best test case with

Algorithm 3: Simplified version of the DynaMOSA algorithm. Given a program, P, and the set of coverage goals, C, DynaMOSA constructs a test suite, TS.

```
   Input: A set of coverage goals C.
   Input: Program, P.
   Input: Population Size M.
   Output: A test suite, TS, which is a collection of test cases
 1 D = GetControlDependencies(P)
 2 P = RandomPopulation();
 3 A = InitArchive(P, C);
 4 C' = UpdateCurrentGoals(A, C, D);
 5 while Search budget is not expanded do
 6 |   Q = GenerateOffspring(P);
 7 |   P = {};
 8 |   A = UpdateArchive(A, Q, C');
 9 |   C' = UpdateCurrentGoals(A, C', D);
10 |   fronts = Rank(Q);
11 |   for front ← fronts do
12 |   |   if P.size >= M then
13 |   |   |   break;
14 |   |   end
15 |   |   for TC ← fronts do
16 |   |   |   if P.size + 1 > M then
17 |   |   |   |   break;
18 |   |   |   end
19 |   |   |   AddToPopulation(P, TC);
20 |   |   end
21 |   end
22 end
23 TS = A.getTestCases();
```

respect to each coverage goal. After the first front, each subsequent front ranks the remaining test cases by their pareto-optimality. The length (number of statements) of the test case is used as a tiebreaker when two test cases have the same score, preferring shorter test cases which is more likely to run in shorter time. Then, the top ranked test cases form the population of the next generation and the offspring of this population are generated, and the process continues until the search budget is exhausted.

Archive. DynaMOSA maintains an archive, A (used in lines 8 and 23), similar to other search-based test generation strategies. During the test generation process, the archive stores test cases covering previously uncovered goals and provides a way to retrieve the best test case for a particular search goal. The archive accounts for accidental coverage; a goal may be collaterally covered by a previous search for another set of goals. When a test case for a particular search goal is stored in the archive, this search goal is removed from the current set of goals (line 8). As a consequence, the current set of search goals contains only the uncovered goals and focuses the search process on them. The archive is updated whenever the search goal is covered, but also when a test case that is shorter than the current test and covers the same goal. At the end of the test generation process, the test cases in the archive are retrieved and are the output test suite (line 22).

```
1  if (functionA())) { // Initially targeted as it does not depend on another line
2    if (functionB()) { // Targeted only after line 1 is covered
3      functionC();   // Targeted only after lines 1 and 2 are covered
4    }
5  } else {
6    functionD();     // Targeted only after line 1 is covered
7  }
```

Figure 5.2: DynaMOSA uses the control dependencies to dynamically target only search goals that can be covered. Line 3 is targeted only after lines 1 and 2 are covered. After lines 1 and 2 are covered, line 3 is added to the current set of goals.

Dynamic selection of targets. The key feature of DynaMOSA is that it allows Evosuite to dynamically select targets based on the control dependencies between one another. Initially, only a subset of the coverage goals that are independent of other goals are targeted. Dynamically selecting targets allows Evosuite to be more efficient when trying to cover multiple structural goals.

For example, statements within branches require the if-statement to be covered first, therefore these statements are initially not targeted by Evosuite until the if-statement has been covered. If the if-statement has not been

covered, then these goals cannot be covered. The fitness, of a test with respect to these goals is, therefore, always worse than the fitness of the test with respect to goal of covering the if-statement. In the example shown in Figure 5.2 statement 3 cannot be covered before both statements 1 and 2 are covered. If a test case has not covered statement 1, it is not necessary to consider the fitness value of a test with respect to the search goal of covering statement 3. Therefore, these uncoverable goals do not contribute meaningfully to the ranking.

Evosuite first computes a control dependency graph of the program before generating tests and uses information from the control dependency graph to update the current set of search goals. As described earlier, before ranking test cases by their pareto-optimality, DynaMOSA first ensures that tests that are closest to a targeted search goal always survive and are retained in the next generation, even if these tests are not pareto-optimal with respect to the other goals. This makes it more probable for Evosuite to progress towards individual goals, including those that may be difficult to cover. This particular feature is the reason why we build DICE on top of the DynaMOSA strategy.

In Algorithm 3, the dynamic selection of targets can be seen in lines 4 and 9, in which the archive is used to determine which goals have been covered. DynaMOSA uses the control dependencies, D, to determine the initial set of targets in line 4. As the goals are covered, it adds the goals that depend on the covered goals in line 8 to the currently targeted set of goals, C'.

5.3 The DICE Approach

5.3.1 Overview

We show a high-level overview of the approach used by DICE in Figure 5.3. DICE consists of 2 main components: DICE-Tester and DICE-Miner. From a high-level perspective, DICE takes a class under test and an initial test suite as input, producing a FSA model as output. DICE first exercises the test suite, collecting the execution traces. This is followed by three phases:

- **Mining Purity-Aware Temporal Specification.** First, temporal specifications, in the form of LTL temporal properties, are mined from these traces while being aware of method purity.
- **Adversarial Test Generation.** The temporal specifications are fed into DICE-Tester and are converted into search goals for the test generation process. DICE-Tester is adversarial to the temporal specifications

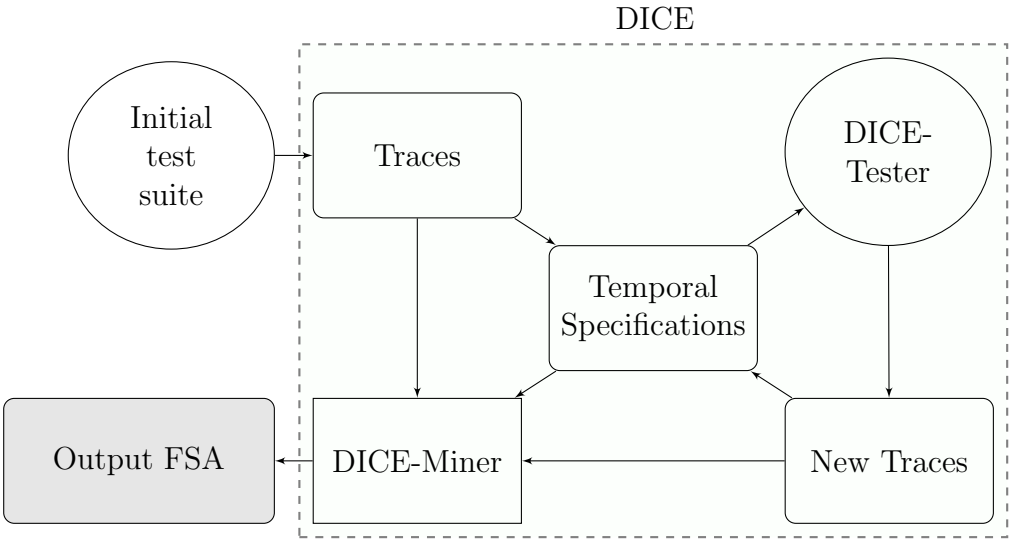


Figure 5.3: High-level overview of DICE

and refines them by invalidating incorrect properties, while generating new test cases and collecting the execution traces of these test cases.

- **FSA Inference.** Finally, the new traces and temporal specifications, with invalid specifications now removed, are input into DICE-Miner, which will infer an FSA. DICE-Miner avoids weaknesses of existing algorithms by using method purity and the temporal specifications to prevent over-generalisation.

5.3.2 Mining Purity-Aware Temporal Specification

The test generation phase of the adversarial specification mining process requires a set of specifications. As such, the first phase of DICE is to mine temporal specifications over the input traces collected from the input test suite. A formalism over constraints ordering events is Linear Temporal Logic (LTL) [327, 202]. Like previous studies in specification mining [142, 250, 85, 86, 428], we use LTL to specify constraints over events. In this work, events are specifically method invocations of a class and the following subset of LTL connectives are used in the property templates:

1. X ϕ means that ϕ has to hold at the ne**X**t state.
2. F ϕ means that ϕ has to hold at some **F**uture state.

3. G ϕ means that ϕ has to hold **G**lobally at all future states.

4. ρ U ϕ is '**U**ntil', which means that ϕ has to hold at some point. ρ has to hold until ϕ holds.

5. ρ W ϕ is '**W**eak until', which means that ρ has to hold until ϕ holds. If ϕ never becomes true, ρ has to hold forever.

Six LTL property templates are commonly used in previous studies. The six LTL property templates are described as follows:

1. AF(a, b): an occurrence of event a must be eventually followed by event b. In LTL, this rule is $G(a \rightarrow XFb)$

2. NF(a, b): an occurrence of event a is never followed by event b. In LTL, this rule is $G(a \rightarrow XG(\neg b))$

3. AP(a, b): an occurrence of event a must be preceded by event b. In LTL, this rule is $\neg aWb$

4. AIF(a, b): an occurrence of event a be immediately followed event b. In LTL, this rule is $G(a \rightarrow b)$

5. NIF(a, b): an occurrence of event a is never immediately followed by event b. In LTL, this rule is $G(a \rightarrow X(\neg b))$

6. AIP(a, b): an occurrence of event a must be immediately preceded by event b. In LTL, this rule is $F(a) \rightarrow (\neg a\ U(b \wedge Xa))$

The last three properties, introduced by Beschastnikh et al. [85] and Le et al. [250], are "immediately" variants of the first three properties and have been shown to be useful for describing FSAs. In this work, we use the LTL property templates from previous work which only consider 2 events. Later, in Section 5.5.4 (Qualitative Evaluation), we note some limitations of considering only 2 events at a time. We also note that the primary objective of this study is to infer automata models, and the LTL specifications are only later used to guide the testing process and the inference of the models.

While we use the same LTL property templates studied by Beschastnikh et al. [85] and Le et al. [250], in our work, we adapt three of them. Recently, Sun et al. [372] pointed out shortcomings of these patterns when using crowdsourcing to identify temporal specifications. AIP(a, b) and AIF(a, b), for example, can never be true since a method can always be invoked between any pair of events a and b. To address these shortcoming and to retain the benefits of these properties in describing temporal constraints, we

observe that side-effect-free method invocations can never affect the state of a software system, and as such, can be abstracted away in the description of the "immediately" variants of the LTL properties. We reformulate these variants to incorporate knowledge of side-effect free methods:

1. AIF(a, b): an occurrence of event a must be immediately followed by an occurrence of event b , ignoring all occurrence of side-effect-free events. In LTL, this rule is $G(a \rightarrow X\ ((p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\ U\ b)$, where p_1, p_2,...,p_n are side-effect free events, known ahead of time.

2. NIF(a, b): an occurrence of event a is never immediately followed by event b, ignoring the occurrence of side-effect-free events. In LTL, this rule is $G(a \rightarrow X(p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\ U\ \neg b)$, where p_1, p_2,...,p_n are side-effect free events, known ahead of time.

3. AIP(a, b): an occurrence of event a must be immediately preceded by event b, ignoring the occurrence of side-effect-free events. In LTL, this rule is $F(a) \rightarrow (\neg a\ U(b \wedge X((p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\ Ua)))$ where p_1, p_2,...,p_n are side-effect free events, known ahead of time.

To identify side-effect free methods of a class, we use a lightweight static analysis [200] and a heuristic based on the method name (we consider names starting with "is-" or "has-" to be getters, which are typically pure). While static analysis is used to partially accomplish this, it is not necessary for our approach. When neither source code nor bytecode is available, a developer can annotate the purity of relevant events by hand.

To mine LTL specifications, we use a solution similar to the linear miner algorithm described by Lemieux et al. [257]. However, as we are only interested in 2-event rules and we restrict ourselves to a few properties, we do not face the challenges that they solve. We use a simple way of iterating over the traces and try to use each trace to falsify the LTL specifications.

In the work by Lemieux et al. [257], support and confidence thresholds are needed. The support counts the number of times while iterating through the traces where a property can be falsified, but is not. The confidence of a property is the ratio of support of a property to the number of times the property can be falsified. When the number of times the property can be falsified is 0, the confidence is defined to be 1.

For our work, as we are interested only in properties that are never falsified, thus in our implementation, we require a confidence of 1.0 for the rules that we mine. Moreover, we only require a support of 1 to admit the temporal property. In other words, as long as a property holds on a trace, and we do not encounter any trace that contradicts the property, we admit the property.

5.3.3 Adversarial Test Generation

The specification mining process adversarially generates test cases against an input specification, aiming to invalidate the specification. To this end, DICE converts the temporal specifications mined from the previous phase into search goals for search-based testing. The objective of this phase is to allow for the discovery of test cases that produce traces that are uncommon and not represented in the initial test suite. To do so, the temporal specifications are converted into fitness goals for test generation. DICE-Tester generates test cases using Evosuite[1] with the addition of the new fitness goals and coverage criteria.

A fitness cost is computed for each fitness goal for each test case that is generated. A single test case may contain multiple object instances of the class of interest and we consider a single trace to be the methods invoked on a single object instance. Therefore, each test may produce multiple traces, one for each object instance in the test case. The fitness of a test, T, with respect to a fitness goal, G, is determined by the trace with the best fitness.

$$Fitness(T, G) = min(Fitness(tr, G)) \quad , \quad tr \in traces(T)$$

We define a new coverage criterion, `TemporalPropertyCounterExample(PropertyType, EventA, EventB)`, based on the LTL properties we have mined. An temporal property is covered if a trace contains a sequence of method invocations that is a counterexample of it. In this formulation, DICE-Tester creates a fitness goal for each temporal property we have mined, guiding Evosuite to produce counterexamples for them. With respect to a single fitness goal, a test is fitter than another if the fitness cost of the test is lower than the other. A goal is covered when the fitness cost is 0, i.e., when a counterexample trace is produced when exercising the test.

When the property is not covered, we aim to guide the test generation process towards a test covering the property. Generally, a trace is scored relative to the number of modifications required to transform the trace to, first, a trace supporting the temporal property, then, to falsifying the temporal property. This will push test generation towards test cases that first support the property, from which they may be mutated towards counterexamples afterwards.

To this end, we grant a better fitness cost when the trace contains one method of a "never" property (NF, NIF) and when the trace contains both

[1]Evosuite version 1.0.6 was used in this study

methods of an "always" property (AP, AIP, AF, AIF). To summarize, for a temporal property AP(A,B) or AIP(A,B), we assign fitness costs to traces such that the following ordering holds:

1. Traces falsifying the temporal property (best)
2. Traces supporting the temporal property
3. Traces containing at least one of the events, A or B, but neither supporting or falsifying the temporal property.
4. Traces where none of A and B are present (worst)

The above ordering guides our design of the fitness functions for all property types. Using the above ordering, the traces are partitioned into 4 subsets for the "always" properties, AP, AIP, AF, and AIF. In our implementation, as Evosuite expects a numerical cost, we evenly partition the range [0, 1] into the 4 subsets. Formally, each trace, tr, is assigned a cost for the search goal targeting a property, p, relating two events, A and B, based on the following conditions:

$$Fitness(tr, p) = \begin{cases} 0.0 & \text{if falsifies(tr, p)} \\ 0.33 & \text{if !falsifies(tr, p) and supports(tr, p)} \\ 0.66 & \text{if !falsifies(tr, p) and !supports(tr, p)} \\ & \quad \text{and (contains(tr, A) or contains(tr, B))} \\ 1.0 & \text{otherwise} \end{cases} .$$

To provide some intuition, we use an example using a property AP(A, B), event B must precede event A. A trace in which event B does not precede event A, e.g. [X, Y, A], will falsify the property. Given a trace [B, X, Y, A], which supports the property, the trace is one transformation away, in which the method call B is removed, away from falsifying the property. For another example, using a trace [B, X, Y]. This trace is one edit from supporting the property. The event A is added in any of the three positions (between B and X, between X and Y, after Y). After this, the modified trace is one edit from falsifying the property AP(A, B). As it took two edits, the trace [B, X, Y] is less fit than the trace [B, X, Y, A] which only requires one edit. Observe that a trace may containing sequence of events that both supports and falsifies a property. A trace [A, B, A] has both the sequence [A], which falsifies AP(A,B), and [B, A], which supports AP(A,B). According to our fitness formulation, this trace will obtain a fitness cost of 0, since it falsifies the property.

A trace without both events A and B, e.g. [X, Y], has the worst fitness cost for the AP(A, B) search goal. This may be contrary to intuition, since a trace without any of the property's events may resemble a trace that has a few edits away from falsifying the property. For example, the trace [X, Y] is only 1 edit from falsifying the property directly, e.g. [X, Y] to [X, Y, A]. There are numerous such traces among the entire test population. An objective of our fitness functions is to focus the search on properties that are hard to falsify. If the property can be easily covered by having such a trace transformed to a counterexample trace, there would be numerous such cases. It is very likely that the search goal can be covered as part of collateral coverage when the entire test population evolves to satisfy other coverage goals. On the other hand, temporal properties that are more difficult to cover will benefit from the focused search.

For example, consider the property AP(`isEmpty:false`, `push:true`) for a hypothetical data structure, which holds if the data structure cannot report that it is not empty unless the push method has been successfully invoked. A counterexample trace can be constructed, e.g. [`<init>`, `pushAll:true`, `isEmpty:false`], by using an alternative method (`pushAll:true`) to insert an item into the data structure. From a trace supporting the event, e.g. [`<init>`, `push:true`, `isEmpty:false`], a single transformation from `push:true` to `pushAll:true` will lead to a trace falsifying the property. On the other hand, given an initial trace [`<init>`, `get:false`, `clearAllElements`] that does not contain either event in the property, if we try to falsify the property by adding a `isEmpty` method invocation, instead of adding the `isEmpty:false` event, the event `isEmpty:true` added to the trace. It is not possible to arbitrarily add an `isEmpty:false` event into any position in a trace. The `pushAll:true` event has to be successfully added first. As a result, a randomly generated trace without the `isEmpty:true` event, e.g. [`<init>`, `get:false`, `clearAllElements`], is much further than a trace supporting the property, e.g. [`<init>`, `push:true`, `isEmpty:false`].

For a temporal property AF(A,B), a trace is scored similarly. Observe that the fitness cost of AF(A,B) can be computed by using the same scoring rules for AP(B, A) and reversing the trace. Likewise, AIF(A,B) can be computed using AIP(B, A) by reversing the trace.

For a NF property, NF(A,B), the same ordering of traces described above applies. A trace supporting the property is a trace where A is present, but B does not appear after A. In the "immediately" variation of NF, NIF, the fitness functions can be computed to differentiate individual traces from one another with higher granularity, while still respecting the ordering of traces. In NIF, the fitness cost reflects how closely positioned the two events in the property are within the trace. A better fitness cost is returned if both events

are included in the trace, and we score the fitness value based on how far apart the method invocations are. This will push test generation towards tests where the events are located nearer to each other. The fitness cost of a trace, tr, with respect to the fitness goal of the property NIF(A,B) is computed as shown in Algorithm 4.

Algorithm 4 takes a single trace, consisting of a sequence of events, as input and returns the fitness score of this trace. The fitness score ranges from 0 to 1, and a score of 0 indicates the maximum fitness value while a score of 1 indicates that the trace has no relevance for this goal. A single pass is made through the sequence of events to look for instances of event A (lines 5-19). A counter (line 4) to measure the distance between a pair of events A and B is updated in this pass. Each time we reach an instance of event A, we record its position and reset the counter (lines 7,8). Once an instance of event A has been found (line 10), we increase the counter for each impure (i.e. not side-effect free) event we pass (line 11-12). Whenever we reach an instance of event B, we update the minimum distance between the last seen instance of A and B if the counter is less than the previous minimum distance between A and B (line 14-16). Finally, the fitness score is computed from the minimum distance. If the distance is 0, then the goal is covered and 0 is output as the fitness score (line 20-22). Otherwise, the ratio of the distance to the length of trace is used as the fitness score (line 23-27).

To conclude, the satisfaction criterion of each *TemporalPropertyCounterExample* depends on the property type. Effectively, if a test contains a trace with a fitness score of 0, then the test is a counterexample. It may be interpreted as follows:

- AF(a,b): a test is a counterexample of AF(a, b) if it contains a trace with an invocation of a that is not (immediately) followed by b.
- NF(a,b): a test is a counterexample of NF(a, b) if it contains a trace with an invocation of a that is (immediately) followed by b.
- AP(a,b): a test is a counterexample of AP(a, b) if it contains a trace with an invocation of b that is not (immediately) preceded by a.

As a consequence of including our new fitness goals into Evosuite, there are a large number of search objectives (one for each temporal property we have mined). As described in Section 5.2, this hampers the ability of Evosuite to search effectively for good test cases that will cover the uncovered goals. The large number of search goals causes the search process to be similar to a random search process, reducing its performance. To manage the large set of goals, we use the DynaMOSA search algorithm as discussed earlier in Section

Algorithm 4: Fitness computation of a trace, tr, with respect to `TemporalPropertyCounterExample(NIF, A, B)`

Input: A trace, *tr*. A fitness goal for a NIF property, $NIF(A, B)$.
Output: Fitness cost of the trace. 0 means that the property is covered

```
1  eventAPosition = -1;
2  i = 0;
3  distance = 999;
4  counter = 0;
5  for event ← tr do
6  |  if event = A then
7  |  |  eventAPosition = i;
8  |  |  counter = 0;
9  |  else
10 |  |  if eventAPosition ≠ −1 then
11 |  |  |  if !isPure(event) then
12 |  |  |  |  counter += 1;
13 |  |  |  end
14 |  |  |  if event = B then
15 |  |  |  |  distance = Min(counter, distance);
16 |  |  |  end
17 |  |  end
18 |  end
19 end
20 if distance = 0 then
21 |  return 0
22 else
23 |  if eventAPosition = -1 then
   |  |  /* Event A does not appear, trace is irrelevant to
   |  |     this goal                                        */
24 |  |  return 1
25 |  else
26 |  |  return distance / (length(tr))
27 |  end
28 end
```

Thus, we use the DynaMOSA search algorithm in DICE-Tester with three modifications. To further boost the efficiency of selecting tests, our first modification is to model the dependencies between the coverage goals beyond structural goals to allow Evosuite to dynamically select goals more effectively. While the DynaMOSA algorithm already models the dependency between structural coverage goals based on control dependencies, it does not model other forms of dependencies, including usage dependencies between methods. It is unable to perform any reasoning about method coverage goals, which DICE-Tester is able to constrain using the LTL properties we have mined previously.

The dependency tree is constructed prior to the start of the search process. At this stage, we assume that every mined LTL properties is true. This dependency tree relates method coverage goals with the `TemporalPropertyCounterExample` coverage goals that we introduced in this work. A method coverage goal, `Method(A)`, is covered when a test case invokes the method A at least once. We add dependencies based on the following rules:

- Given AP(A, B), DICE-Tester adds a dependency where B depends on A. i.e. `Method(A)` should be covered before `Method(B)`
- Given NF(A, B) or NIF(A,B), DICE-Tester adds a dependency where `Method(A)` should be covered before `TemporalPropertyCounterExample(NF, A, B)` or `TemporalPropertyCounterExample(NIF, A, B)`.
- Given AF(A, B) or AIF(A,B), DICE-Tester adds a dependency where `Method(A)`should be covered before `TemporalPropertyCounterExample(AF, A, B)` or `TemporalPropertyCounterExample(AIF, A, B)`.
- Given AP(A, B) or AIP(A,B), DICE-Tester adds a dependency where `Method(B)` should be covered before `TemporalPropertyCounterExample(AP, A, B)` or `TemporalPropertyCounterExample(AIP, A, B)`.

For example, given `NF(StringTokenizer, nextToken)`, for its corresponding goal `TemporalPropertyCounterExample(NF, StringTokenizer, nextToken)` to be satisfied, the method coverage goal, `Method(StringTokenizer)`, must be satisfied first. If this method coverage goal is not satisfied, then it is impossible for the test to produce a counterexample and this goal is always completely uncovered for any test case. Thus, it will not contribute meaningfully to the ranking of tests maintained by DynaMOSA [158]. Constraints about the ordering of methods provided by the set of AP temporal properties also allow us to add dependencies between a pair of method coverage

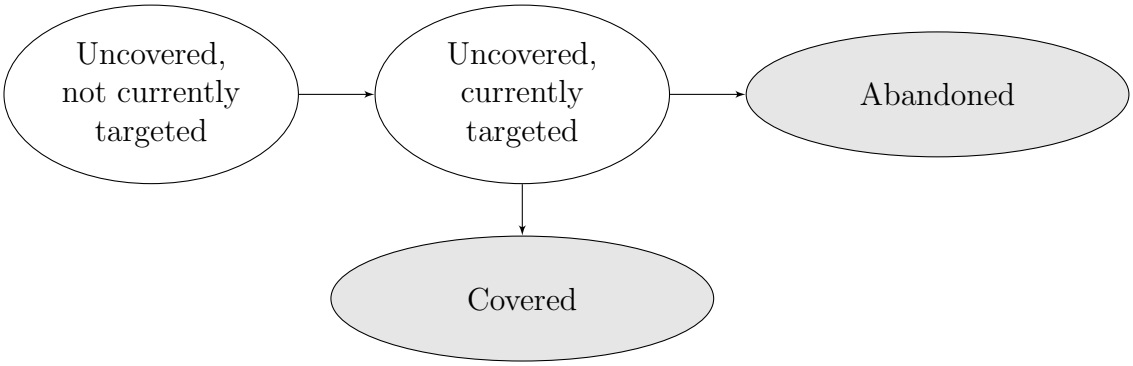


Figure 5.4: Lifecycle of a search goal. At the end of the search process, a goal is either covered or abandoned.

goals. For another example, given `AP(initVerify,update)`, we add a dependency between `Method(update)` and `Method(initVerify)`, requiring an invocation of *initVerify* before Evosuite adds *update* to the set of currently targeted goals. While other coverage goals may return a fitness value between 0 and 1, the fitness of a test with respect to a method coverage goal is binary, either the goal is covered or it is not. Therefore, if the prerequisites are not met, a method coverage goal is always uncovered for every test case.

Our second modification is made for the lifecycle of a search goal. In the DynaMOSA algorithm, goals are in one of 3 states. A goal is either a) covered, b) uncovered but not in the current set of goals, or is c) uncovered but in the current set of goals. In DICE-Tester, we added a new state in this lifecycle. Thus, a goal can be in one of 4 states: covered, uncovered but in the current set of goals, uncovered but not in the current set of goals, or *abandoned* (see Figure 5.4). We refer to a goal as *abandoned* if it is not covered but we have removed it from the current set of goals.

The addition of this state is motivated by the fact that at least some of the mined properties are true. As these properties are true, it is impossible to generate a test case that is a counterexample to it. Consequently, goals representing counterexamples to these properties can never be covered, and this has detrimental effects on the search process. Such goals will continue to contribute to the preference ranking used in Evosuite. This has several implications. Test cases that are closest to and "almost covering" the goal will always remain in our population, although they will not contribute to producing interesting test cases. The search may be weighted towards tests that have a higher chance of covering these goals, even though these goals cannot be covered. This may potentially prevent other test cases (that may lead to a counterexample of another property) from getting added to the

population. Furthermore, these goals have no meaningful contribution to the ranking process when weighing the pareto-optimality of the other test cases. This slows down the search process, wasting time to compute the fitness of tests with respect to these goals.

Hence, we add the abandoned state in the lifecycle of a goal. To enable goals to transit to this state, DICE-Tester tracks the age of each goal. Covered goals and goals that are not in the set of currently targeted goals do not have an age. When an uncovered goal moves into the set of targeted goals, its age is initialized to 0. We increment the age of a goal for each generation where DICE-Tester does not find a test case that covers it. Once the age of a goal has exceeded a threshold, we abandon the goal by removing it from the set of targets. In our experiments, we set this threshold to 100 generations. Once abandoned, goals can never be restored back to the set of targets.

Our third modification is to reset the population of the tests once it has gotten stuck. While the first two modifications allow Evosuite to find more counterexamples, we observed that the search can still get stuck in a local optima. Finally, we bypass this problem by resetting the population of tests, and in effect restarting the search, once 100 generations has passed without DICE-Tester finding a test that is a counterexample to *any* goal. We did not thoroughly empirically evaluate the threshold for abandoning a goal or resetting the population, however, we noticed in our experiments that changing these parameters do not affect the results much, provided that they are not too small.

With these three modifications, DICE-Tester is able to guide test generation towards finding counterexamples to spuriously mined LTL properties, to invalidate them. Temporal properties with at least one counterexample are removed.

Next, we collect execution traces from the output test suite, which are constructed from the test cases that produced counterexamples to the temporal properties. These are later passed as input to the FSA inference process. Typically, specification mining algorithms use traces of correct executions and our approach is not an exception. Therefore, we filter out traces of executions that may not represent a correct usage. If the invocation of a method results in a thrown exception, we ignore the invocation and any further method invocations (as the exceptional invocation may have an effect on the state). Apart from exceptional executions, we also try to detect resource leaks and omit any possible trace that caused them. We keep track of the number of file descriptors that are opened by the process that is executing the tests [2]. Next, we compared the number of file descriptors before and after

[2]We run Evosuite without its sandbox which prevents environmental interactions

the runs, and if we find a mismatch between them, we assume that the tests have triggered a sequence of methods that leaked a file or a socket. When this happens, we remove all traces of tests that led to resource leaks. While this may inadvertently result in correct traces that are incorrectly discarded (consider a scenario where a test instantiated multiple FileOutputStreams, we discard traces from every FileOutputStream as long as one of them caused a leak, even if the rest of the instances represent correct usage of FileOutputStream), this helps us ensure that the traces we have collected do not contain executions of invalid usage of an API or class.

5.3.4 Example of the search process

```
interface DataStructure<T> {

    public boolean add(T item);

    public boolean addAll(Collection<T> items);

    public boolean isEmpty();

    public void clear();

    public T get();

    public Collection<T> getAll();
}
```

Figure 5.5: The API of a hypothetical data-structure.

In this section, we present a synthetic example of the search process. Using a hypothetical data-structure shown in Figure 5.5, we show how a small set of test cases may evolve over a few generations to become counterexamples for a three properties that are not true properties:

- Goal 1: AP(isEmpty:FALSE, add:true),
- Goal 2: AIF(clear, isEmpty:TRUE), and
- Goal 3: AF(clear, isEmpty:TRUE),

We run the DICE process to search for test cases that will falsify the properties. In this simple example, all three properties are falsifiable. As described earlier, we use the dependencies between search goals to consider fewer search goals at a time, omitting goals that cannot be covered yet. At the start of the DICE-Tester process, the following dependencies between

search goals are added:

$$Method(isEmpty : FALSE) \rightarrow AP(isEmpty : FALSE, add : true)$$
$$Method(isEmpty : TRUE) \rightarrow AIF(clear, isEmpty : TRUE),$$
$$AF(clear, isEmpty : TRUE)$$
$$Method(clear) \rightarrow AIF(clear, isEmpty : TRUE), AF(clear, isEmpty : TRUE)$$
$$Method(add : TRUE) \rightarrow AP(isEmpty : FALSE, add : true)$$

The dependencies should be interpreted such that the search goals on the right (the consequent) are added to the currently targeted set of goals once the search goal on the left (the antecedent) has been covered. The test generation process starts by creating a population of randomly generated tests. In this example, we assume that the population size is two, and that each test produces only one trace. First, DICE-Tester generates 2 tests, giving us the following traces as shown below in Figure 5.6:

```
[clear, isEmpty:TRUE, getAll]                    (-, 0.33, 0.33)
[getAll, add:FALSE]                              (-, 1.00, 1.00)
```

Figure 5.6: The initial population. The numbers on the right show their fitness costs for Goals 2 and 3. Goal 1 is not considered yet as neither of the method coverage goals it depends on has been covered.

In Figure 5.6, Goal 1 is not considered among the search goals (listed as '-') as neither of the method coverage goals it depends on has been covered. When the number of goals is large, this helps to simplify the comparison between test cases. When this initial population of test cases is evolved, it produces a set of new test cases as shown in Figure 5.7. Since the method coverage goal of Goal 1 has been covered, all the search goals for the 3 properties are now considered.

```
[clear, getAll, isEmpty:TRUE, getAll]            (1.00, 0.00, 0.33) *
[clear, isEmpty:TRUE, add:TRUE, isEmpty:FALSE]   (0.33, 0.33, 0.33) *
[add:TRUE, clear, isEmpty:TRUE, getAll]          (0.66, 0.33, 0.33)
[getAll, isEmpty:TRUE, add:TRUE, getAll]         (0.66, 0.66, 0.66)
[getAll, add:TRUE, get]                          (0.66, 1.00, 1.00)
```

Figure 5.7: The offspring of the first generation. We order them by their pareto-optimality.

In Figure 5.7, the test cases that are selected to form the next population of tests are indicated with a *. As the first test case has fully covered Goal

2 (falsifying the property that `clear` is always followed by `isEmpty:TRUE`), it is selected. As for the second test case, it ties with the third test case for the best fitness score on Goal 3, but it is the best test case for Goal 1. Therefore, both the first and second test cases are selected as they are the best test cases with respect to the search goals. The remaining test cases are ordered by pareto-optimality. The third test case dominates the fourth test case, and the fourth test case dominates the fifth test case. The first test case is inserted into the archive, as it has fully covered a search goal. If the population size was greater than 2, then the remaining test cases would be added in this order.

Finally, in the next round, we get the following offspring shown in Figure 5.8. As search goal 2 was already fully covered earlier, we no longer need to consider the test cases' fitness with respect to it.

```
[clear, isEmpty:TRUE, addAll:TRUE, isEmpty:FALSE]   (0.00, -, 0.33) *
[clear, clear, getAll]                              (0.66, -, 0.00) *
[clear, getAll, isEmpty:TRUE, add:TRUE, getAll ]    (0.66, -, 0.66)
[clear, isEmpty:TRUE, sEmpty:TRUE, add:TRUE, getAll] (0.66, -, 1.00)
```

Figure 5.8: The offspring of the second generation of test cases. All the test cases are covered after this. Note that Goal 2 was not considered when computing the objective function vectors as it has a corresponding test case in the archive.

Both Goals 1 and 3 have counterexample traces from this set of offspring, and these two test cases are added to the archive. As all the search goals related to the temporal properties are covered, we end the test generation process. In practice, it is typically the case that not all coverage goals can be covered and the test generation process only ends when the search budget is fully consumed. The test cases from the archive are retrieved to form the output test suite, which is run and its execution traces are collected to be used as input to the next step of DICE.

5.3.5 FSA Inference

The final phase of the adversarial specification mining approach is to take the temporal properties and traces produced from the previous phases and infer an FSA model. In DICE, the DICE-Miner algorithm is responsible for this. We add the traces produced by the DICE-Tester to the original set of traces for input to DICE-Miner.

An overview of DICE-Miner is given in Figure 5.9. The DICE-Miner algorithm to infer an FSA-based specification is similar to the k-tails algorithm,

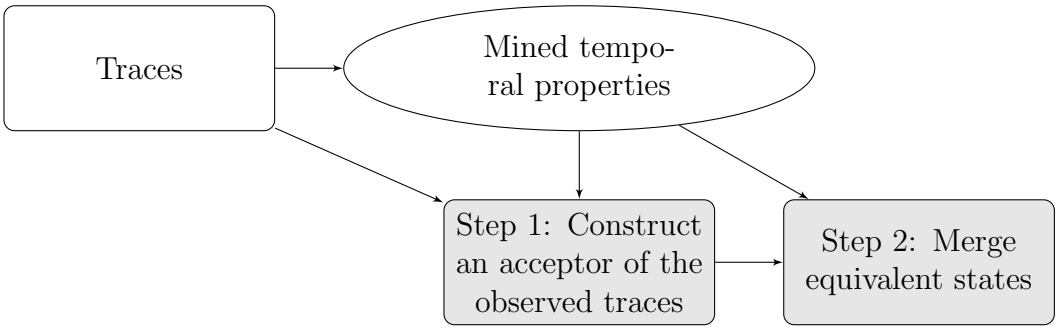


Figure 5.9: High-level overview of DICE-Miner

comprising of two steps. In the first step, we construct an initial automaton based on the input traces and the mined temporal properties. Similar to the PTA (described earlier in Section 5.2), our initial automaton accepts all of the observed input traces. The mined temporal properties are used to prevent erroneous merging of states. In the second step, we merge equivalent states in this automaton, leveraging the mined temporal properties. The mined temporal properties are used to derive the set of enabled methods of each state, which is the equivalence criteria used in DICE-Miner to merge states. We elaborate on this equivalence criteria below in Section 5.3.5.

Regarding the first step of the algorithm, we make the following observations:

- Observation A: Side-effect-free events can be interleaved in any order, and do not change the present state of the software system.
- Observation B: The first step of constructing a PTA makes the assumption that states with the same history of events are equivalent. This assumption may not be true.

We propose a specification mining algorithm with these observations in mind. In light of the first observation, we include information of the purity of each method to allow the model to have greater generalizability. The second observation guided us to prevent the incorrect conflation of states while constructing an initial model that accepts all of the input traces.

With observation A, we model the freedom of side-effects as self-loops in the automaton. We add Constraint A where we ensure that transitions labelled with side-effect free method calls are self-loop. This has the advantage of increasing the model's generalizability, allowing it to accept an equivalent trace with a different permutation of the pure methods. For example, for `StringTokenizer`, observing an input trace [`StringTokenizer`,

`hasMoreElements:true`, `hasMoreTokens:true`] allows the construction of an automaton that will accept a different trace (note a different ordering of the method invocations), [`StringTokenizer`, `hasMoreTokens:true`, `hasMoreElements:true`]. This is the case even without the observation of a trace with this sequence of method calls, since the only difference is that the pure methods *hasMoreTokens* and *hasMoreElements* were invoked in a different order from the same state.

To address observation B, we add Constraint B where we prevent the erroneous conflation of states occurring in the first step of k-tails and its variants. If these states are incorrectly merged in the first step, regardless of the equivalence criteria selected for merging states, this inaccuracy in the initial model will negatively impact the quality of the final FSA produced. This is because the second step does not split up incorrectly merged states from the first step. For an example of Observation B, with the `Iterator` class, given the trace [`Iterator, hasNext:true, next`, `hasNext:false`] and a different trace sharing a prefix, [`Iterator, hasNext:true, next`, `hasNext:true`], the states reached in the two traces after the invocation of **next** is different, yet constructing a PTA will conflate these states as they share the same preceding events [`Iterator`, `hasNext:true`, `next`]. Conflating these states produces an automaton where both `hasNext:false` and `hasNext:true` are incorrectly enabled from the same state.

To address this, we propose to detect incorrectly merged states using the mined temporal properties while constructing an initial automaton accepting all of the observed traces. Whenever adding a new transition to an automaton results in an automaton that may produce traces violating the mined properties, we modify the automaton such that these violations will not occur. Next, we discuss the details of the DICE-Miner algorithm, and show we address both observations within the first step of our algorithm.

First Step

Next, we describe the first step of the DICE-Miner algorithm. In the first step, we pass the example traces into the function *CreateCompatibleAcceptor*, which constructs a Non-deterministic Finite Automaton (NFA). This automaton is build to accept all of the example traces, much like a PTA. However, unlike the construction of a PTA, *CreateCompatibleAcceptor* avoids the creation of states that may accept sequences of events that violates a constraint (we refer to such states as *incompatible* with the constraint). We consider a state to violate a constraint if it causes the automaton to accept a trace violating the constraint. Observe that although the states were constructed based on individually observed traces, and that every trace does not violate the temporal properties mined earlier, it is possible to construct a PTA with states that may accept traces violating the temporal properties.

Algorithm 5: Pseudocode for CreateCompatibleAcceptor. This is a simplified version of the algorithm. In reality, the automaton is non-deterministic and given an event, there may be multiple transitions labeled with the same event.

Input: Input traces, *traces*
Output: A DFA that accepts all traces in *traces*, without creating any states that may violate any constraint

```
1  automaton = emptyAutomaton();
2  for trace ← traces do
3      prefix = [];
4      state = automaton.initialState;
5      for event ← trace do
6          if state.canAccept(event) then
7              nextState = state.accept(event);
8              state = nextState;
9          else
10             if !state.HasIncompatibleTransition(event) then
11                 newState = state.addTransitionToNewState(event);
12                 state = newState;
13             else
14                 stateToModify, suffix =
                    FindAncestorWithoutIncompatibleTransition(state,
                    event);
15                 for suffixEvent ← suffix do
16                     newState = stateToMod-
                        ify.addTransitionToNewState(suffixEvent);
17                     stateToModify = newState;
18                 end
19             end
20         end
21     end
22     return automaton
23 end
```

For example, given two traces, [`Stack`, `addAll`, `remove`, `isEmpty:TRUE`] and [`Stack`, `addAll`, `remove`, `get`] and a temporal property, *NIF(isEmpty=True, get))*, a state with an incoming edge, `isEmpty:TRUE` (a self-loop on the state, as a result of Constraint A and the fact that `isEmpty` is pure), and an outgoing edge, `get`, is *incompatible* with the temporal property as it can accept a trace with the sequence of events [`isEmpty:TRUE`, `get`]. The algorithm to split up a state with incompatible edges is given in Algorithm 5.

CreateCompatibleAcceptor first initializes an empty automaton before iterating over each trace (lines 1-2). For each event in a trace, we first try to accept the event without modifying the NFA (lines 5-8). On reaching an event, e, that cannot be accepted, we modify the NFA to add new states and transitions such that it will accept the events. Before adding a new state, we first ensure that the new transition will not cause the current state to be incompatible with any constraint (line 10). If adding the transition results in an incompatible state (lines 13 - 19), we traverse backwards, looking for an ancestor where we can add transitions corresponding to the events up to e without introducing an incompatible state (line 13). From this ancestor node, we add new transitions and states to represent the events up to e (lines 15-18).

Algorithm 6: Pseudocode of FindAncestorWithoutIncompatible-Transition. Traverses the ancestry of a state to locate a state to branch from.

Input: The current state, *state*, and the event that introduced an incompatible state, *event*

Output: An ancestor state and a sequence of events to add transitions for

```
1 parent = state;
2 suffix = [];
3 label = event;
4 while parent.HasIncompatibleTransition(label) do
5 |   grandParent, label = parent.getIncomingTransition();
6 |   suffix = label :: suffix;
7 |   parent = grandParent;
8 end
9 return parent, suffix
```

To find an ancestor from which it is possible to add a new chain of events such that an event can be added without incompatibility, we use the algorithm *FindAncestorWithoutIncompatibleTransition*. The details of *Find-*

Algorithm 7: Pseudocode of HasIncompatibleTransition. Checks if a transition labeled with the event can be added to the input state.

```
   Input: A state, state, and an event to add, event
   Output: true if adding the event does result in the state becoming
           incompatible
 1 for transition ← state.incomingTransitions do
 2 |   if NIF(transition,event) then
 3 |   |   return true
 4 |   end
 5 end
 6 if NF(state.prefix,event) then
 7 |   return true
 8 else
 9 |   return false
10 end
```

AncestorWithoutIncompatibleTransition are given in Algorithm 6. We initialize the algorithm (lines 1-3) before we iteratively traverse the ancestors of the parent state. At this state of the DICE-Miner algorithm, all states have at most one incoming transition originating from another state, which is obtained using `getIncomingTransition`. As we traverse backwards, we collect the labels on the transitions (line 6). These labels correspond to the events that we will have to add transitions for. The traversal ends once we find an ancestor that we add the sequence of events including e without creating an incompatible state.

Algorithm 7 shows the check for an incompatible state. As we only add new outgoing transitions, it is sufficient to check pairs of the existing incoming transitions with the new event to detect NIF violations (lines 1-5). To check NF violations, we check the prefix of the state against the event (line 6). The prefix of each state is constructed by traversing all transitions on the trail of ancestor states. This includes all self-loops on each ancestor (i.e. the self-loops are traversed first before moving to a child state). It is enough to check for violations of NF and NIF, as the other properties are never violated in the first stage of DICE-Miner.

At the end of the first step, we receive an automaton that contains self-loops in some states, and these are the only cycles in it. The inferred model is a Non-Deterministic Automaton (NFA) as states can have multiple transitions with the same label. We presented simplified versions of the algorithms for ease for explanation. As the automata involved are NFAs, there may be

more than one transition from a state given an event. The model is constructed based on the observed traces. It is sound with respect to these traces, and will accept all of these concrete traces.

Second Step

In the second step, we merge equivalent states in the automaton. To merge two states, a and b, we remove both states from the state machine, then add a new state, c. All the transitions from a and b are added onto state c. If there are multiple transitions with the same label, source, and destination, only one of them is kept and the duplicates are removed. As we already noted earlier, our model is non-deterministic and there may be multiple transitions from a state labelled with the same event.

To determine if two states are equivalent, we draw inspiration from the CONTRACTOR model [128] and define the equivalence of states based on the set of methods that are enabled. Concrete states that have the same enabled methods are merged. As described by de Caso et al. [128], this results in models with states that are intuitively interpreted and are at an abstraction level that developers find convenient. While we reuse the concept of the enabledness of methods in order to group states, our method is still primarily based on the execution traces that are input to DICE. The CONTRACTOR method requires further annotation and the computing of dependencies between the enabledness of all pairs of methods. In our work, we do not use these dependencies and other related concepts described by de Caso et al. [128] to avoid the need to annotate the pre- and post-conditions of every method. Prior work [242] has also shown that the performance of the CONTRACTOR approach is highly dependent on the quality of the pre- and post-conditions, and that it exhausts the running memory when provided with noisy invariants. Instead, we use the temporal properties to aid in determining the enabledness of a method at a particular state. We leave the study of the applicability of the dependencies of method enabledness on DICE and their integration into DICE for future work.

However, unlike de Caso et al. [128] and Krka et al. [242] which creates models from the state invariants of an object, we derive the enabledness of a method from the set of NF and NIF properties. If there are no LTL property that *disables* a method based on its prefix, the known incoming transitions, and the known outgoing transitions, then we consider that this method is enabled. We consider a method, say method A, to be *disabled* on a state from following conditions.

$$disabled(state, A) = \begin{cases} true & \text{if } NF(X, A), X \in prefix(state) \\ true & \text{if } NIF(X, A), X \in incoming(state) \\ true & \text{if } NIF(A, Y), A \text{ is pure}, Y \in outgoing(state) \\ false & \text{otherwise} \end{cases}.$$

If a method on a state leads to the automaton accepting a trace containing a pair of successive events violating the property, then the method is disabled. For example, a state with an incoming transition `add:true` (indicating that an item was successfully added) will have the event `isEmpty:false` disabled (the state can't be empty after a successful addition) by the second condition. As another example, a state with the outgoing transition `remove:true` (indicating that an item can be successfully removed from this state) will have the event `isEmpty:true` disabled (if an item can be successfully removed, it means it can't be empty) by the third condition. Methods can be enabled even if we have not seen the method invoked from a particular state in a concrete trace. Before merging two states, DICE-Miner checks that it does not lead to a violation of known LTL properties. Similar to the work by Lo et al. [275], a pair of states cannot be merged if it results in a state machine that violates any temporal property known to hold on the observed traces.

5.4 Evaluation

To empirically evaluate our tool, we investigate 3 research questions.

- **RQ1: How effective is DICE in inferring FSA models?**
  RQ1 investigates the effectiveness of the adversarial specification mining approach by comparing the FSA models inferred by DICE against models inferred by state-of-the-art specification miners.

- **RQ2: How effective is DICE-Tester?**
  RQ2 investigates the effectiveness of the test generation component, DICE-Tester. This is done by comparing the tool against Evosuite, with DynaMOSA as its search algorithm, as a test generation baseline to answer this question. Instead of using the traces produced by DICE-Tester, we study if using traces produced by Evosuite is enough to mine better specifications. Here, our objective is to evaluate the value of the temporal-property guided adversarial test generation strategy we built on top of Evosuite.

- **RQ3: How effective is DICE-Miner?**
  RQ3 investigates if the specification miner, DICE-Miner, can utilize the traces generated by DICE-Tester effectively. We compare DICE-Miner against DSM as a baseline, passing the same set of traces produced by DICE-Tester as input to both tools.

5.4.1 Experimental setup

Evaluation To investigate these questions, we empirically evaluate the tools by assessing the quality of the models inferred against ground-truth models. Similar to previous studies, we measure precision and recall. This procedure for computing precision and recall has been used in prior studies [251, 242]. As input, a ground-truth model and the model inferred by DICE is provided. From these two models, traces are generated by randomly traversing the edges in the model. The precision of the inferred model is the percentage of traces produced by the inferred model that are accepted by the ground-truth model. The recall of the inferred model is the percentage of traces it accepts among the traces produced by the ground-truth model. In other words, precision is the proportion of traces from the inferred that are correct, and recall is the proportion of correct traces that the inferred model accepts. Finally, the quality of the model is measured using F-measure, computed as follows.

$$\text{F-Measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The 11 ground-truth models publicly released from Le et al.'s evaluation of DSM [252] are used [3]. These evaluation library classes, used for evaluating specification miners in previous studies, represent 100 analysed methods in total, and represent different categories of libraries ranging from data streaming to message exchange. To analyse the classes from JDK, we copied the source files of the corresponding classes from OpenJDK 1.8, to get around a constraint in Evosuite that prevented instrumentation and bytecode-rewriting of classes from some packages provided by the JDK. OpenJDK 1.8 was used as this was the version used in previous studies, and we observe that the choice of a more recent version will not impact the evaluation results as the ground-truth models involved only methods from earlier JDK versions. We also omitted traces from DICE-Tester containing events that are not present in the ground-truth models. This is done to allow direct comparison against the approaches used in previous studies. During our evaluation, we discovered a minor inaccuracy in the ground-truth model of

[3] `https://github.com/lebuitienduy/DSM`

Table 5.1: Precision, Recall, and F measure of DICE and DSM. NFST refers to NumberFormatStringTokenizer.

Class	DICE			DSM		
	P	**R**	**F**	**P**	**R**	**F**
ArrayList	31.4	**27.3**	**29.2**	**44.5**	16.3	23.9
HashMap	100.0	**94.1**	**97.0**	100.0	55.2	71.1
HashSet	**87.4**	**100.0**	**93.3**	74.0	62.4	67.7
Hashtable	84.0	**100.0**	**92.5**	**100.0**	66.6	79.9
LinkedList	100.0	**100.0**	**100.0**	100.0	23.7	38.4
NFST	**87.2**	**89.2**	**88.2**	54.1	70.2	61.1
Signature	100.0	**100.0**	**100.0**	100.0	91.2	95.4
Socket	**87.3**	**67.3**	**76.0**	58.4	62.6	60.4
StackAr	**86.8**	93.9	**89.8**	61.6	**97.1**	75.4
StringTokenizer	**100.0**	100.0	**100.0**	93.6	100.0	96.7
ZipOutputStream	**100.0**	100.0	**100.0**	80.6	100.0	89.3
Average	**87.8**	**88.3**	**87.8**	77.3	67.3	68.4

ZipOutputStream, in which DICE-Tester found counterexamples to. Hence, we corrected the ground-truth model to account for the missing transition. Finally, for each case, we account for randomness by computing the average of the evaluation metrics from 20 runs of the experiment.

5.4.2 RQ1: Effectiveness in inferring FSA models

To determine the effectiveness of our tool for answering RQ1, we compute precision, recall and F-measure of the output FSAs for 11 target library classes. We compare against DSM [252], which uses deep-learning and randomized test generation [306], as a baseline. For a second baseline, we also compare our tool against Tautoko [126], which leverages test generation to complete an initial FSA model. Tautoko takes the specifications inferred by the specification miner, ADABU [127]. Given an initial test suite, it learns an initial model using ADABU and then mutates test cases and executes them again to cover missing transitions in the initial model.

We initialized DICE using the test suite used by DSM in its evaluation, which is generated by Randoop [307]. From the results reported in Table 5.1, DICE improves on the average F-measure of DSM by over 19% (from 68.4 to 87.8), and, for every class, the difference between DICE and DSM is statistically significant – measured using the Wilcoxon signed-rank test. This indicates that DICE was effective in inferring FSA models.

Table 5.2: Precision, Recall, and F-measure of Tautoko and DICE. NFST refers to NumberFormatStringTokenizer.

Class	DICE			Tautoko		
	P	**R**	**F**	**P**	**R**	**F**
ArrayList	31.4	27.3	29.2	-	-	-
HashMap	**100.0**	**94.1**	**97.0**	56.7	25.4	35.1
HashSet	**87.4**	**100.0**	**93.3**	100.0	13.6	23.9
Hashtable	**84.0**	**100.0**	**92.5**	38.8	23.3	29.1
LinkedList	100.0	**100.0**	**100.0**	100.0	20.9	34.6
NFST	87.2	89.2	88.2	**100.0**	**100.0**	**100.0**
Signature	100.0	**100.0**	**100.0**	100.0	23.8	38.4
Socket	**87.3**	**67.3**	**76.0**	84.1	24.4	37.7
StackAr	86.8	**93.9**	89.8	**100.0**	87.0	**92.8**
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	**100.0**	**100.0**	100.0	25.0	40.5
Average	87.8	88.3	87.8	88.0	44.3	53.2

We also investigated the effectiveness of DICE against Tautoko [126], which generates tests based on missing transitions in an initial model. We compare the FSAs mined by DICE against Tautoko's. The publicly available version of the executable artifact was downloaded from Tautoko's website[4] and executed on the same evaluation classes above, using Randoop generated tests as input to Tautoko. For some classes, Tautoko produced models containing methods that were not present in the ground-truth models. We therefore manually modified the models produced by Tautoko such that these methods were omitted, and merged states connected by a transition labelled with removed methods. For Socket, ZipOutputStream, and Signature, we evaluate the models published on Tautoko's homepage due to technical difficulties we encountered trying to run Tautoko on these classes. However, we modify the evaluation criteria as Tautoko does not produce models with transitions labelled with boolean return values of method calls. To compute the evaluation metrics for Tautoko's models, we ignore return values. This should generally lead to higher F-measures. We report the results in Table 5.2.

In one case (ArrayList), Tautoko does not run to completion within 24 hours. For the other classes that Tautoko can mine models for, we observe that Tautoko does not produce models of high F-measures. Apart from StackAr, DICE outperforms Tautoko on the 11 classes. While DICE produces

[4]https://www.st.cs.uni-saarland.de/models/tautoko/

models with an average F-measure of 87.8, Tautoko produces models with an average F-measure of 53.2. If we omit the model of ArrayList, then DICE produces models with an average F-measure of 93.7. We hypothesize that in certain cases, Tautoko's reliance on inspector methods (see Section 5.2.1) meant that it can not identify the right abstract states. For example, ZipOutputStream's state is not characterized by any inspector methods, and as such, Tautoko is unable to mine a good model of it.

The adversarial specification mining process implemented by DICE produces FSA-based models of higher quality, which outperforms existing approaches for inferring FSAs

5.4.3 RQ2: Effectiveness of DICE-Tester

To answer RQ2, we aim to determine if the improvements was a result of our improvements to Evosuite, by studying if Evosuite alone was enough to produce diverse tests that would benefit the specification miner. We use Evosuite (version 1.0.6), with the DynaMOSA [311] search algorithm, as a baseline approach, collecting the traces produced by the final test suite that is the output of Evosuite. We use the default configuration of Evosuite. We do not try to find the optimal configuration for Evosuite as previous studies have indicated that tuning these parameters often do not outperform the default configuration [67]. By default, the population size of test cases is 50 individuals. The default crossover operator is used, which is the single-point crossover with probability of 0.75. The selection of test cases is done using tournament selection, with a tournament size of 10. Tests are mutated with a probability inversely proportional to the number of statements it contains. We use the same test budget of 15 minutes for both DICE-Tester and Evosuite. The traces produced from Evosuite are passed as input to DICE-Miner instead of the traces from DICE-Tester, and we compute the evaluation metrics of the FSA models produced.

The results are shown in Table 5.3. On some classes, DICE-Tester can outperform Evosuite by up to 18.9% in F-measure, and on average, DICE-Tester outperforms Evosuite by about 3.7%. To mitigate the effect of randomness, we run the experiments 20 times and compute if the differences are statistically significant using the Wilcoxon signed-rank test. We find that the differences are statistically significant for 4 out of the 11 classes in the benchmark. This indicates that, on its own, Evosuite is already effective in generating diverse test cases, although DICE-Tester can explore some uncommon usage patterns more effectively. We hypothesize that many of these classes do not exhibit complex usage constraints, therefore, Evosuite already

Table 5.3: Precision, Recall, and F-measure of DICE-Miner when using DICE-Tester and Evosuite. NFST refers to NumberFormatStringTokenizer. * indicate that the difference in F-measures is statistically significant.

Class	DICE-Tester			Evosuite		
	P	**R**	**F**	**P**	**R**	**F**
ArrayList *	31.4	**27.3**	**29.2**	**74.7**	17.0	27.6
HashMap	100.0	94.1	97.0	100.0	94.1	97.0
HashSet *	**87.4**	**100.0**	**93.3**	84.5	65.2	74.3
Hashtable	84.0	**100.0**	**92.5**	**91.0**	93.1	92.0
LinkedList *	100.0	**100.0**	**100.0**	100.0	89.9	94.7
NFST	87.2	89.2	88.2	87.2	89.2	88.2
Signature	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**87.3**	**67.3**	**76.0**	86.4	67.5	75.8
StackAr *	**86.8**	**93.9**	**89.8**	68.9	84.7	76.0
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	100.0	100.0	100.0	100.0	100.0
Average	87.8	**88.3**	**87.8**	**90.4**	81.9	84.1

performs well for these classes. However, to effectively explore non-trivial usage constraints, DICE-Tester can help significantly.

While Evosuite, with the DynaMOSA algorithm, is already able to aid the specification mining process by producing traces that allow DICE-Miner to achieve good performance, DICE-Tester takes it a step further, producing traces that are even better.

5.4.4 RQ3: Effectiveness of DICE-Miner

To answer RQ3, we compare DICE and DSM with both approaches using the same set of traces. We run DSM when provided with the traces from DICE-Tester. We compare the performance of DICE-Miner against DSM. In this study, we do not compare our tool against other specification miners since DSM [252] is the state-of-the-art specification miner and has been demonstrated to outperform multiple approaches such as the traditional k-tails algorithm, SEKT, TEMI and CONTRACTOR++ [252]. The results of using DSM with the additional traces are shown in Table 5.4.

We observe that DSM's performance does not improve with the additional traces. In fact, the additional traces causes the performance of DSM to decrease. As DSM's states are partially determined by the probability of observing each next transition, we hypothesize that it is sensitive to the set

Table 5.4: Precision, Recall, and F-measure of DICE and DSM with the traces produced from DICE-Tester. NFST refers to NumberFormatString-Tokenizer.

Class	DICE			DSM		
	P	**R**	**F**	**P**	**R**	**F**
ArrayList	31.4	**27.3**	**29.2**	**60.4**	16.5	25.9
HashMap	**100.0**	**94.1**	**97.0**	30.8	86.0	45.3
HashSet	**87.4**	**100.0**	**93.3**	50.9	52.7	51.8
Hashtable	84.0	**100.0**	**92.5**	**93.3**	70.2	80.1
LinkedList	100.0	**100.0**	**100.0**	100.0	16.5	25.9
NFST	**87.2**	**89.2**	**88.2**	57.3	81.9	67.4
Signature	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**87.3**	**67.3**	**76.0**	40.7	63.9	49.8
StackAr	**86.8**	93.9	**89.8**	47.2	**100.0**	64.1
StringTokenizer	**100.0**	100.0	**100.0**	75.3	100.0	85.9
ZipOutputStream	**100.0**	**100.0**	**100.0**	79.8	75.4	77.5
Average	**87.8**	**88.3**	**87.8**	64.7	71.4	63.4

of input traces used and it does not handle low-probability, but still valid, transitions robustly. Also noteworthy is that DICE-Miner can achieve a 100% F-measure in 4 of the 11 classes, while DSM achieves a perfect score in only one case. This shows that DICE-Miner is able to utilize diverse traces more effectively than DSM.

DICE-Miner outperforms a state-of-the-art specification mining technique even when the diverse traces produced by DICE-Tester are included in its input.

5.5 Discussion

To further investigate our results, we raised four additional research questions for investigation and qualitatively analysed the models produced by DICE.

- **RQ4: How effective were our adaptations of Evosuite for finding counterexamples?**
  RQ4 studies the effectiveness of DICE-Tester in discovering counterexamples of temporal properties. Instead of measuring the quality of the FSAs output from DICE-Miner, an indirect measurement of how much the test generation benefited the specification mining process, we directly inspect the number of false temporal properties that the

two tools, DICE-Tester and Evosuite (with DynaMOSA), are able to invalidate by finding counterexamples.

- **RQ5: How much did the constraints we introduced help to improve the performance of our specification mining algorithm?**
  RQ5 investigates the two constraints that we added in DICE-Miner motivated by the two observations we made. These observations are described in Section 5.3.5, involving method purity and incompatible transitions on a state. The first constraint ensures that transitions labelled with side-effect free methods are self-loops, and the second constraint prevents the erroneous conflation of states that may accept traces that violate previously-mined temporal properties. We try to drop these constraints and observe their effect on the performance of DICE-Miner.

- **RQ6: How much does the quality of the initial test suite affect DICE?**
  RQ6 varies the quality of the initial test suite by using reduced subsets of it as input to the DICE process. We aim to investigate if the quality of the initial test suite has an effect on the quality of the models inferred by DICE.

- **RQ7: Can the FSAs inferred by DICE be used to support additional testing activities, for example, to perform protocol fuzzing?**
  RQ7 investigates if the FSAs learned by DICE can be used to aid in fuzzing servers of stateful protocols. Effective fuzzing of a server requires the fuzzer to be aware of the specific order of messages to reach certain states. We use state models learned by DICE to initialize a server fuzzer and observe if it helps the fuzzing process. This functions as an evaluation of DICE to determine if its inferred models have practical applicability.

5.5.1 RQ4: Effectiveness in finding counterexamples

To try to further quantify the difference in performance of DICE-Tester and Evosuite+DynaMOSA, we compare the set of temporal properties that were successfully invalidated at the end of the test generation process. We evaluate them against the ground-truth properties annotated by human experts. These ground truth properties were made publicly available by Sun et al. [372]. The human experts annotated each possible temporal property following the template indicating if the property is true. This was done for

Table 5.5: Number of incorrect rules failed to be invalidated when using all possible LTL properties as input

Class	DICE-Tester	Evosuite
HashSet	**51**	233
StackAr	**39**	108
StringTokenizer	**35**	216
Average	**41.7**	185.7

three classes, HashSet, StringTokenizer and StackAr, which we use in our evaluation.

As input to DICE-Tester, we enumerate all possible temporal properties between the methods of the class, and run DICE-Tester. For Evosuite+DynaMOSA, during the test generation process, we collect the traces when executing tests and print to standard output if the trace of the test produced is a counterexample to a temporal property. Note that the set of input temporal properties will include properties that contradict each other (e.g. both `NF(A, B)` and `AF(A, B)` are part of the input). In total, there were 56 true properties for HashSet out of 1014 possible properties, 35 true properties out of 384 possible properties for StringTokenizer and 42 true properties out of 600 possible properties for StackAr.

The results are reported in Table 5.5, and we observe that in each class, DICE-Tester successfully found counterexamples for a vast majority of the incorrect properties. Out of an average of 621.7 incorrect properties, DICE-Tester successfully constructs tests that contradict an average of 580 of them, or over 93% of the them. In contrast, Evosuite does not succeed in invalidating most of the incorrect properties, and there were four times the number of incorrect properties that Evosuite failed to find counterexamples of.

Next, we also investigate the effect of the threshold for resetting the test population, described in Section 5.3.3. On average, the test budget of 15 minutes allows to search to run for 505 generations. With a threshold of 100 generations, we observe an average of 1 reset for each run. We varied the threshold and run DICE on the classes in the benchmark. The evaluation metrics are reported in Table 5.6. Our findings are that having too low a threshold adversely affects the quality of the models learned by DICE. Decreasing the threshold from 100 generations to 50 generation caused a large drop in F-measure from 87.8% to 64.4%, over a 20% decline. On the other hand, increasing the threshold to 150 generations caused a slight decrease in quality, from 87.8% to 86.7%. The results suggest that, at least for this benchmark set of classes, the default value that we choose (100 generations)

Table 5.6: Precision, Recall, and F-measure for DICE varying the threshold for resetting the test population. NFST refers to NumberFormatStringTokenizer.

Class	100	50			150		
	F	**P**	**R**	**F**	**P**	**R**	**F**
ArrayList	**29.2**	72.6	15.7	25.8	78.0	16.1	26.6
HashMap	97.0	100.0	94.1	97.0	100.0	100.0	**100.0**
HashSet	**93.2**	77.9	18.3	30.0	95.9	80.7	87.6
Hashtable	92.5	98.2	100.0	**99.1**	96.6	100.0	98.3
LinkedList	100.0	5.9	44.4	10.4	100.0	100.0	100.0
NFST	**88.2**	87.7	86.4	87.0	86.6	85.6	86.1
Signature	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**76.0**	53.4	66.0	59.0	80.5	66.0	72.5
StackAr	**89.8**	41.7	88.2	56.6	83.2	84.7	84.0
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	**100.0**	28.6	100.0	44.4	72.7	100.0	84.2
Average	**87.8**	69.6	73.9	64.5	91.3	86.7	86.7

is reasonably good.

DICE-Tester outperforms Evosuite in finding counterexamples for incorrect temporal properties, validating the benefits of our modifications for improving the search process.

5.5.2 RQ5: Effectiveness of constraints in inferring FSA

To study if the two constraints described in Section 5.3.5 influenced the performance of DICE-Miner, we ran more experiments, omitting the constraints. Constraint A refers to the constraint that pure methods do not cause a transition to another state, while Constraint B refers to the constraint that a state should be split up if it has incompatible transitions. We use the execution traces from Le et al. [252] and DICE-Tester in these experiments.

The results are presented in Table 5.7. Without both constraints, the average F-measure dropped by about 5.2%. In virtually all classes, we see a decline in the performance of DSM-Miner. We see that using information about method purity is important to achieving good performance. While using Constraint B alone did not provide any improvements without Constraint A, it was necessary to increase the performance to 87.8 from 82.6 (with only Constraint A). This confirms our observations that these two constraints are important for specification mining.

Table 5.7: F-measure of DICE-Miner, without the constraints (A and B) we identified. NFST refers to NumberFormatStringTokenizer.

Class	Both	with A, w/o B	with B, w/o A	None
ArrayList	29.2	28.6	30.4	31.3
HashMap	97.0	97.0	97.8	97.8
HashSet	93.2	88.5	91.8	92.3
Hashtable	92.5	86.3	84.9	87.7
LinkedList	100.0	100.0	70.8	70.8
NFST	88.2	72.7	81.8	81.8
Signature	100.0	100.0	100.0	100.0
Socket	76.0	63.5	67.9	67.8
StackAr	89.8	71.9	65.2	74.5
StringTokenizer	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	100.0	100.0	100.0
Average	87.8	82.6	80.9	82.2

The two constraints that we added for addressing the two observations were helpful for DICE-Miner to achieve its performance.

5.5.3 RQ6: Effect of the quality of initial test suite

To answer RQ6, we investigate how sensitive the DICE process is to the initial input test suite. We performed experiments using different subsets of the initial test suite. We do not run experiments using the test suite originally written by the developers accompanying the classes in the benchmark. We manually analysed the test cases for these classes and found that exercising these tests would only produce a few traces. The functionality of each class is well tested, but the tests typically do not have a high diversity regarding the sequences of method invocations. For example, the test cases for a LinkedList exercise each method of the LinkedList, but do not show how the methods relate to one another. Therefore, we do not expect the evaluation metrics of DICE to change significantly from using a small subset (e.g. 25%) of the initial test suite used as input to the earlier experiments.

The results are shown in Table 5.8. By using 50% of the initial test suite, F-measure drops from 87.8% to 87.2%, and by using 25% of the initial test suite, it drops further to 86.8%. This indicates that the initial quality of the test suite influences the quality of the models inferred by DICE, however, the difference is small. Interestingly, we noticed that F-measure increased for some of the classes in the benchmark. The models for ArrayList and

Table 5.8: Precision, Recall, and F-measure for DICE varying the initial test suite. NFST refers to NumberFormatStringTokenizer.

Class	100%	50%			25%		
	F	**P**	**R**	**F**	**P**	**R**	**F**
ArrayList	**29.2**	31.9	21.1	25.4	28.6	25.3	26.8
HashMap	97.0	100.0	94.1	97.0	100.0	83.1	90.8
HashSet	93.2	92.4	100.0	96.0	94.1	100.0	**97.0**
Hashtable	92.5	98.0	100.0	99.0	96.3	100.0	**98.1**
LinkedList	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NFST	88.2	92.1	89.4	90.7	92.9	87.9	**90.3**
Signature	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**76.0**	84.8	62.5	72.0	88.3	61.7	72.6
StackAr	**89.8**	76.4	81.9	79.0	73.9	84.7	78.9
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Average	**87.8**	88.7	86.3	87.2	88.6	85.7	86.8

HashSet improved in quality by about 1% in F-measure. For these cases, as having fewer initial traces may mean that we infer a larger number of incorrect temporal properties, we hypothesize that these large number of incorrect traces can sometimes lead the search process to collaterally cover more temporal properties and produce informative traces that were useful to the inference process. We answer RQ6 by concluding that the quality of the initial test suite has an impact on the models inferred by DICE, but overall, this impact is small.

The quality of initial test suite has a small effect (1% change in the average F-measure) on the quality of the models inferred by DICE.

5.5.4 Qualitative Evaluation

While the DICE system results in an improved F-measure compared to existing approaches, it is not able to achieve 100% correct finite-state automata for all of the 11 classes. We manually inspected the resulting FSA to try to identify reasons as to why this is the case, and to propose next steps for our work.

One interesting observation is that the interpretation of FSA models may differ between models that have identical F-measures. Apart from boosting the accuracy of models, having knowledge of pure methods will produce models that are qualitatively better for program comprehension. For exam-

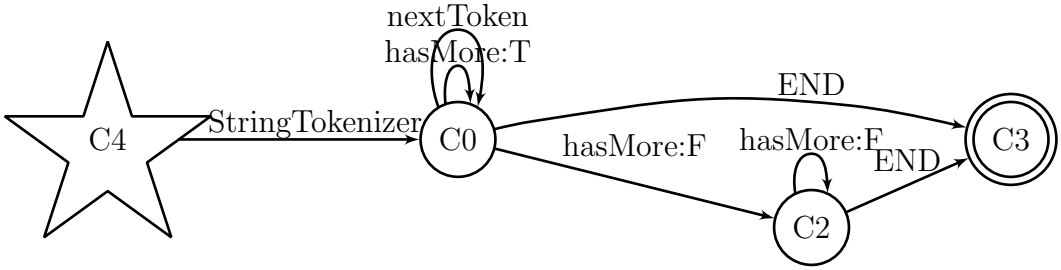


Figure 5.10: Example of a FSA model produced by DSM. hasMore:T is short for hasMoreTokens:TRUE and hasMore:F is short of hasMoreTokens:FALSE

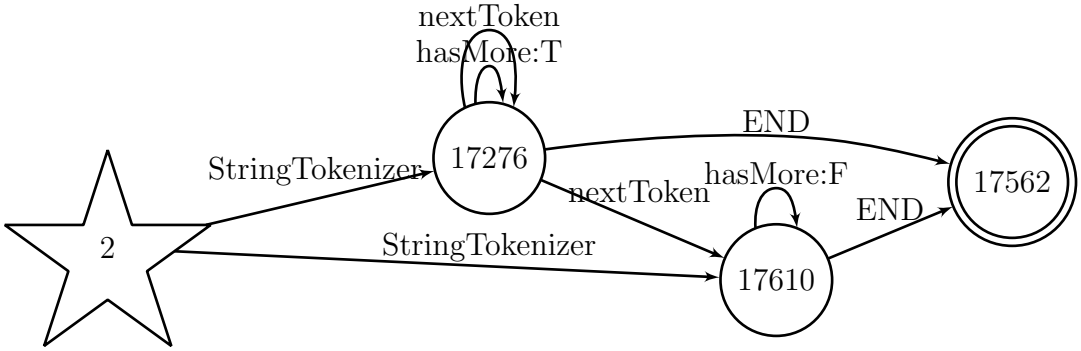


Figure 5.11: Example of a FSA model produced by DICE-Miner. hasMore:T is short for hasMoreTokens:TRUE and hasMore:F is short for hasMoreTokens:FALSE

ple, referring to Figure 5.10, DSM produces a model for StringTokenizer that suggests that `nextToken` is always enabled should a developer never invoke `hasMoreTokens`. In contrast, the model produced by DICE-Miner, as shown in Figure 5.11, cannot be erroneously interpreted in this manner. The invocation of `hasMoreTokens` with different return values are not allowed on the same state and it is clear that `nextToken` may change the state of a StringTokenizer object such that `nextToken` can not be further invoked. Note that both the models produced by DSM and DICE-Miner have an F-measure of 100, showing that there may be qualitative differences between models that are automatically evaluated to be perfectly accurate.

The method `hasMoreTokens` represents an interesting case that may be a next step for DICE-Miner. While it is accurately identified as a side-effect free method heuristically by DICE-Miner, the static analysis we performed did not reveal it as a pure method. The implementation of `hasMoreTokens` is, in fact, impure. A static analysis-based approach therefore treats hasMoreTokens as an impure method. More sophisticated analysis of purity, such as using the notion of observational-purity [80, 299] will help to produce qualitatively better models. Observationally-pure methods refer to the class of methods, in which `hasMoreTokens` belongs to, which have side-effects that cannot be observed outside of the class. From the perspective of learning usage models and specifications about usage of the class, these methods are effectively pure. While our naive name-based heuristic may help to identify some of these methods, it is likely that there are observationally-pure methods in the wild that we cannot detect.

Investigating the poor performance of DICE for some classes, the lack of expressiveness of the six LTL property templates considered is a possible reason for it. LTL properties templates involving more than 2 events and the use of other temporal operators beyond the basic operators may be required. For example, clients of NumberFormatStringTokenizer are able to reset its state using the `reset` method, as such, there are constraints between its methods that can only be expressed through the use of LTL formula involving more than 2 events. The property `NF(hasNextToken():FALSE,nextToken)` is false, contrary to intuition, since invoking `reset` after `hasNextToken:FALSE` may allow `nextToken` to be invoked again. In this case, the three-event property `(hasNextToken():FALSE NF nextToken) U reset` is necessary to accurately represent the temporal constraints between the methods. This formula indicates that `nextToken` cannot follow `hasNextToken():FALSE` until the object instance has been `reset`.

While in this work, we have considered only 2 event LTL formulae, it may be possible to use formulae relating 3 or more events and use them during the testing process or to guide the merging of the states in the automaton.

However, having more complex formulae will come at a cost and there may be a trade-off; gaining some accuracy but slowing down the approach. Including longer rules will lead to an exponential growth in their numbers. Indeed, many studies have focused on mining rules and patterns involving only 2 events [251, 428, 355, 274], and researchers have noted the problem of scalability when mining longer patterns [317].

5.5.5 RQ7: Use of FSA models for fuzzing

Using the inferred FSAs for fuzzing

Next, we evaluate models produced by DICE in its applicability on fuzzing servers of stateful network protocols [325]. In fuzzing, random test inputs are generated automatically in order to find bugs. It is important to discover critical bugs in the implementation of protocols. The Heartbleed vulnerability [5], a security bug in the implementation of the Transport Layer Security in the OpenSSL library, has shown the pervasiveness and the high cost of such bugs [140]. Server fuzzing may help in finding these bugs and several server fuzzers, such as AFLNET [325], have been proposed. Servers are stateful and reaching specific states may require specific sequences of messages between server and client. Without information about the specific messages required, the fuzzer is unlikely to send a sequence of messages that exercises program states deep in the server. The FSA model learned by DICE can be used as a state model to guide the server in reaching states that are difficult to reach otherwise.

AFLNET [325] is a coverage-guided fuzzer that has been shown to outperform other server fuzzers. Like other coverage-guided fuzzers, AFLNET instruments the program to receive feedback if there is an increase in the server's code coverage achieved by an input. This feedback is used by the fuzzer to decide which inputs to mutate, optimising its choice of inputs for increased coverage of the program. Unlike other coverage-guided fuzzers, AFLNET has a state model of the server and detects if inputs lead to unexplored states. In other words, an input is retained for further mutation by AFLNET if it leads to increased coverage or enters an unexplored state. During the fuzzing process, AFLNET constructs a state model of the server using the observed status code sent by the server. By updating the state model during runtime, AFLNET is able to detect bugs in states that only appear in the implementation of the protocol and not in the protocol's official definition. As a tradeoff, AFLNET may spend a significant amount of time early in the fuzzing process using a rudimentary state model and, until the

[5] `https://heartbleed.com/`

state model is refined, is unable to reach deep into the program.

During the fuzzing process, AFLNET selects the next input for fuzzing by selecting a state in the state model to act as the target state. From this target state, two criteria are applied. First, AFLNET picks, with a higher probability, states that are less frequently exercised. Second, states that have been associated with a greater number of inputs that lead to higher coverage or new states are more likely to be selected. Then, an input associated with that state is mutated in a way that ensures that that particular state is still visited by the new input.

A state inference algorithm, such as DICE, can enhance the workings of AFLNET by providing it with the state model before starting the fuzzer. By running DICE on a protocol client, DICE produces a model of the protocol which can be used for server fuzzing. We observe that the API members of a protocol's client tend to have a one-to-one correspondence to the request types defined in a protocol. For instance, an FTP client typically has API members for each request type (e.g. the `user()` method sends a USER request, the `pass()` method sends a PASS request). Consequently, **an automaton specification of a protocol's client API is a model of the protocol from the client's perspective**. The state model is, therefore, useful for a fuzzer to simulate the protocol's client. To target a particular state, one can traverse the edges from the starting state until the target state is reached; the labels on the edges traversed are the requests the client will have to make.

We modify the state-of-the-art server fuzzer, AFLNET, to use the automaton produced from DICE as the state model. Instead of starting the fuzzing process with an empty state model, we modify it to be initialized with the output of DICE. We also make some modifications in AFLNET related to how it uses a state model. Presently, the status code in the server's response is used as an indicator of the present state. On the other hand, the states in DICE's state machine are more granular. We modify AFLNET to traverse the state machine while considering more information than the response code of the server, namely, the sequence of requests made so far. Additionally, when the server responds with a status code indicating an error, we assume that the state has not changed. In contrast, AFLNET transits to an error state when the server replies with server code indicating an error. Overall, with our modifications and initialization with DICE's output, AFLNET can work with a higher granularity of states and is more likely to generate inputs that are accepted, allowing it to generate inputs that go deeper into the program. To distinguish between the two systems, we will refer to our modified version of AFLNET as DICE+AFLNET.

To look at an example of the difference between AFLNET and DICE+AFLNET,

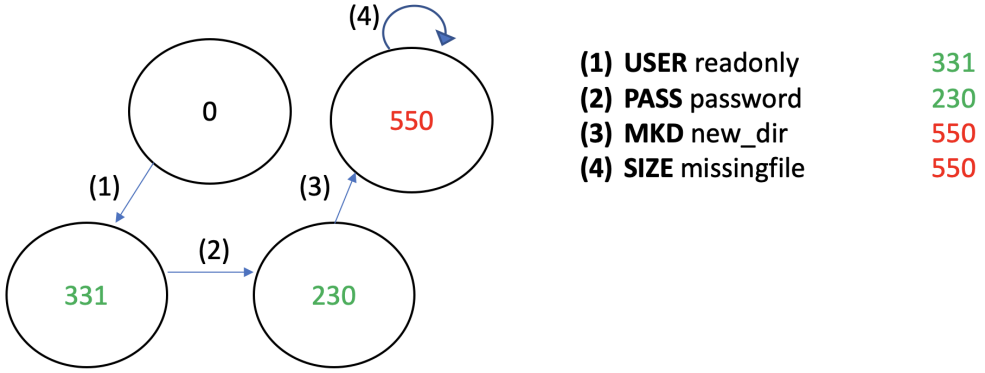


Figure 5.12: State traversal in AFLNET's inferred state machine for the given sequence of request types

we use the following example sequence of requests made by AFLNET: `[USER, PASS, MKD, SIZE]`. In this sequence, the client has successfully logged in with a user. The user then fails to create a new directory, due to insufficient access rights, and attempts to get the size of file that does not exist on the server. Given this sequence of requests, the server responds with the following sequence of response codes: `(331, 230, 550, 550)`, indicating that the login was successful but the creation of the new directory and access of a non-existent file have failed. AFLNET traverses the states as shown in Figure 5.12, in which it first moves to state 331, then to 230, and to 550. On the other hand, DICE+AFLNET traverses the states as shown in Figure 5.13. Notice that DICE+AFLNET remains in the same state after the failed requests. Compared to the transition to state 550 used in AFLNET, we suggest that this more accurately captures the semantics of a failed request in the two protocols that we study. As this may not be true of many network protocols, we will provide configuration options allowing the user of the fuzzer to specify the semantics of a failed request with respect to the state model. Overall, this enables AFLNET to work with the state models from DICE, which are more granular.

Experimental Results. To determine if the models learned by DICE can aid server fuzzers, we evaluate DICE+AFLNET on two protocols that were studied in the evaluation of AFLNET [325], FTP and RTSP. We reuse the same FTP[6] and RTSP[7] servers that were fuzzed by Pham et al. [325]. We run the modified version of AFLNet that takes a state machine from DICE as input. For each of the two protocols, we performed a search on GitHub to find

[6] `https://github.com/hfiref0x/LightFTP`
[7] `http://www.live555.com/mediaServer/`

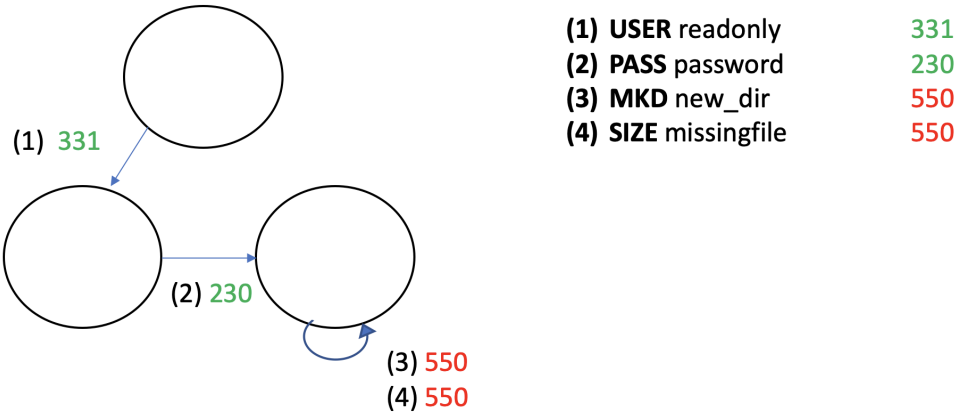


Figure 5.13: State traversal in DICE's inferred state machine for the same sequence of request types as Figure 5.12

open-source clients[8][9] of the protocol. We selected one client for each protocol and ran DICE on it. This process is semi-automatic. Some human effort was required to annotate which API methods directly represent a request from the client. We modified the source of the clients to enable DICE to run effectively on it. Originally, the clients have different API usage patterns in which the developers have to check the integer return value of the method call (`int returnCode = ftpClient.user(username)`), or invoke another method to retrieve the response server code (`if (parseServerResponse() != 200)`). We modified the clients to throw exceptions on requests where the server responds with status codes signalling failure. For each of the two clients, the modification of the client took less than 15 minutes. This was done for DICE to detect a failing method call. DICE was run on each client for another 15 minutes.

The modification of the client and annotation of the methods took less than 15 minutes for each protocol. In total, less than 30 minutes is required to make the necessary modifications and to run DICE. For a fair comparison, we grant AFLNET an additional hour of fuzzing to account for the additional effort and time to run DICE. We fuzz each protocol for 24 hours using DICE+AFLNET, and 25 hours using the original version of AFLNET. We compare the fuzzers by the average line and branch coverage of the resulting inputs. Apart from the additional step of running the automata inference

[8] `https://github.com/apache/commons-net`
[9] `https://github.com/mutaphore/RTSP-Client-Server`

Table 5.9: Line and Branch Coverage achieved on each protocol server by the AFLNET and DICE+AFLNET. Numbers in parenthesis indicate the proportion of total lines/branches covered.

Protocol	**AFLNET**		**DICE+AFLNET**	
	Lines	**Branches**	**Lines**	**Branches**
FTP	644 (57%)	311 (40%)	**777 (69%)**	**400 (50%)**
RTSP	2453 (11%)	1216 (7%)	**2470 (11%)**	**1234 (7%)**

Table 5.10: Line and Branch Coverage achieved on each protocol server when using the state models from DSM and DICE. Numbers in parenthesis indicate the proportion of total lines/branches covered.

Protocol	**DSM+AFLNET**		**DICE+AFLNET**	
	Lines	**Branches**	**Lines**	**Branches**
FTP	703 (62%)	341 (43%)	**777 (69%)**	**400 (50%)**
RTSP	2448 (11%)	1201 (7%)	**2470 (11%)**	**1234 (7%)**

algorithms and the different running time, we follow the same fuzzing procedure as prior work. As the fuzzing process is stochatic, we mitigate the effect or randomness by running 20 independent experiments for each fuzzer and report the average coverage obtained.

The average coverage achieved by the fuzzers are shown in Table 6.6. On the RTSP server, there was only a slight increase in coverage (of 17 lines and 18 branches). As Pham et al. [325] observed, the RTSP server has fewer states to explore and the number of messages required to exercise code deeper into the server is lower than FTP. In other words, for such protocols where the state model is simpler, AFLNET benefits less from having a state model initialized before it begins fuzzing. On the other hand, on the FTP server, the use of the state machine from DICE significantly improves both line and branch coverage. The average line coverage increased from 57% to 69% and branch coverage increased from 40% to 50%.

Next, we investigate if AFLNET achieves the same increase in performance when we use the automata inferred by DSM [252] instead of DICE. The experimental results are shown in Table 5.10. On the RTSP server, the difference between the use of models from DSM and DICE is insignificant. However, on the FTP server, DICE achieves a significantly higher line and branch coverage. We also observe that the coverage on the FTP server increased over the baseline AFLNET fuzzer; it increases from 57% to 62% line coverage, suggesting that the modifications we made to initialize the fuzzing with an initial state machine were useful.

Based on the improvements on fuzzing the FTP server, we answer RQ7 by concluding that there is evidence that the models learned by DICE have practical applicability to a downstream application. DICE provides an informative state model that the fuzzer can use to guide the mutation of inputs, and may effectively boost the performance of a stateful server fuzzer. In the future, we will further evaluate DICE on more protocols.

The FSA models learned by DICE have practical downstream application and can be used for initializing a fuzzer with a state model. It improves the coverage achieved by a fuzzer on the servers of a two stateful protocols.

5.5.6 Threats to Validity

Threats to internal validity. In our studies, we have tried to reuse most of existing implementation whenever possible. While it may be possible that there are bugs in the code we have written, we have checked them multiple times to reduce threats to internal validity. We have evaluated DICE in different aspects, such as evaluating that each component in DICE outperforms strong baselines (DSM and Tautoko for DICE, Evosuite for DICE-Tester, DSM for DICE-Miner). We have performed a deep analysis, including a qualitative analysis, for a deeper understanding of DICE.

Threats to construct validity. In our experiments, we used common evaluation metrics that were used in previous studies. The evaluation process for computing these metrics has been used in previous studies and is well-understood.

Threats to external validity. While it may be possible that our findings do not generalize to other library classes and APIs, we have considered the 11 classes evaluated in previous studies on specification mining. These classes are diverse, coming from both the Java standard library and other third-party libraries. Furthermore, these are classes used in real-world software and are from a range of different domains. We emphasize that our approach cannot capture every possible constraint of an API or an object class, and it may not be possible to propose search goals for every constraint. The models DICE learned may not always be a realistic representation of every program. DICE is currently limited to mining finite-state automata, corresponding to regular languages. Still, on a benchmark created by prior studies, we have shown that the automata mined by DICE are more accurate specifications than those mined by prior approaches. Although DICE cannot mine specifications represented by a context-free language, we note many specifications mined in the literature are regular languages and researchers have found uses for these specifications, such as analysing and finding secu-

rity flaws in bank cards [50] and TLS [129], or modelling Android applications [335]. Moreover, as we have shown in Section 5.5.5, DICE can still learn models that are useful on a downstream task. We leave the mining of other types of specifications, such as those equivalent to context-free languages, as future work.

5.6 Related Work

Usage models have been incorporated in test case generation previously by Fraser and Zeller [159]. However, they use usage models to improve the readability of test cases by guiding test generation to resemble test cases written by humans, and reducing the amount of nonsensical test cases generated. In contrast, our work aims to generate test cases that are correct, but do not resemble common usage patterns that tend to be exercised by existing tests.

For the generation of test inputs, various approaches first learn the probability distribution of observed test inputs. While most techniques generate test inputs that resemble the learned distribution to produce synthetically correct inputs, the Skyfire [400] approach learns a probability distribution to generate test inputs that do not resemble the examples it has seen, applying heuristics such as favouring low probability rules. Pavese et al. [316] proposed inverting the probabilities in grammar-based test generation to explore uncommon test inputs. Our work shares a similar goal with these two studies, aiming to explore uncommon behavior in testing.

Many studies have studied diversity in test generation [287, 149, 112, 150]. For example, researchers have studied the diversity of test inputs and outputs [57]. Shin et al. [367] have proposed the use of diversity for mutation testing. Our work differs from these studies as we focus on the diversity of execution traces produced by tests, instead of modelling the diversity of the inputs or outputs of each method. Moreover, we use test generation only to produce more diverse traces, supporting our objective of learning accurate FSA specifications.

There are other studies on the diversity of software traces, for example, coverage information collected from test execution [258, 150]. Typically, these studies propose metrics over traces to measure the similarity of tests to maximize fault detection capability [149]. Our goal in this study is different, aiming to diversify traces for mining more accurate FSA specification models of a software system.

There are similarities between DICE and model checking approaches that leverage counterexamples. One example is CEGAR [120], that performs counterexample-guided abstraction refinement during model checking. In

CEGAR, counterexamples are used to split the abstract state as a counterexample indicates that there is some behaviour in the abstract model that is not present in the concrete version. The abstract state is split such that it no longer admits the counterexample.

Adaptive model checking [171] is akin to black-box model checking [318], where there is no initial model of the system. In adaptive model checking, an inaccurate model is updated as it is used to verify a software system. Inaccuracies in the model may be due to differences caused by updates to the system. Like other Angluin-style automata learners, counterexamples are used to incrementally improve the model. These model checking techniques use the Vasilevskii-Chow algorithm [390, 118] for conformance testing to check if the model is equivalent to the system. DICE is similar as it tests the system against specifications, but differs in that it never checks or verifies conformance between the model and software system, which can be expensive and impractical with a cost exponential to the size of the automaton. DICE avoids this cost, only performing search-based testing to search for traces that falsify individual LTL properties.

5.7 Conclusion and Future Work

To conclude, we proposed a new approach of adversarial specification mining and prototyped a tool, DICE (**Di**versity through **C**ounter-**E**xamples), for mining specifications. DICE systematically diversifies execution traces and addresses shortcomings in current specification mining algorithms. By adversarially guiding test generation towards finding counterexamples of the specification, our approach produces diverse traces that represent uncommon but correct usage of the program. To do so, we introduce new fitness goals representing counterexamples to temporal specifications expressed in LTL properties, address shortcomings in the LTL property templates used in previous studies, and use search-based testing to produce diverse traces. To take advantage of the diverse traces and the temporal properties, we propose a new specification mining algorithm that utilizes knowledge of method purity and use the temporal specifications to prevent erroneous merges to infer Finite-State Automata models with improved precision and recall. Finally, in our empirical evaluation, our approach significantly outperforms DSM, the current state-of-the-art specification miner, and Tautoko, which generates tests for specification mining. DICE produces models with an average F-measure of 87.8, while the current state-of-the-art approach, DSM, produces models with an average F-measure of 68.4. and Tautoko produces models with an average F-measure of 53.2 in our experiments. Furthermore, our

experiments suggest that the performance of DSM does not always improve when provided with more data. The artifact website of DICE can be found at https://kanghj.github.io/DICE

While we focus on generating uncommon sequences of method invocations in this study, we hope to explore the integration of methods that diversify test inputs [149, 112] to improve DICE's ability to generate uncommon test cases in future. We also hope to investigate more expressive LTL property types and evaluate DICE with other specifications beyond those that were studied in prior work. We will also study the tradeoffs of including longer temporal properties in future. Another possible direction is to explore more complex properties using temporal properties that were hard for DICE-Tester to falsify. The difficulty in falsifying them may indicate that the properties hold, or that there are more complex relationships between the events in the property. Users of DICE may also find these properties useful. We will also study other ways to improve the effectiveness of DICE-Tester. To that end, we hope to explore the use of techniques such as Swarm Testing [172], which may help to further increase the diversity of tests.

Chapter 6

(Dynamic Analysis + API) Active Learning-based Input Selection for Fuzzing Deep Learning Libraries

6.1 Overview

The use of deep learning is now prevalent. It affects many aspects of our lives, including in safety and security-critical domains such as self-driving vehicles [357, 457]. Consequently, there have been increasing concerns about vulnerabilities in deep learning systems, which can have a severe impact. While many studies have focused on testing and uncovering weaknesses of deep learning *models*, the development of approaches that mitigate the risks of vulnerabilities in deep learning *libraries*, such as TensorFlow and PyTorch, is equally crucial. These vulnerabilities may lead to errors that corrupt memory contents or crash the software system, which can be abused for denial-of-service attacks on applications using deep learning libraries.

Challenges. Fuzzing the deep learning libraries poses challenges related to the selection of suitable inputs. The first challenge is that the space of inputs is large and many generated inputs do not belong to the domain of **semantically-valid inputs**. By randomly selecting inputs from the large input space, the vast majority of inputs would be rejected by the library's input validation checks, and therefore, fail to adequately test the library's behaviors. A second challenge is related to the **redundancy of inputs**. Given the observation of a test outcome (e.g., an exception thrown when invoked with a particular input), an appropriate strategy should be employed

to pick an input that tests a different behavior from the already observed test outcome. Otherwise, the same library behavior would be tested again, leading to redundancies in fuzzing. Ideally, a fuzzer triggers a wide range of test behaviors.

For most of the deep learning libraries' APIs, the input constraints are unknown [421]. Without knowledge of the input constraints, randomly generated inputs can be supplied. However, unlike other programs e.g. UNIX utilities [286, 285] that take sequences of bytes as input, libraries accept inputs that are highly structured [308, 76]. Likewise, for TensorFlow and PyTorch, randomly generated inputs are unlikely to be structurally valid (e.g. a tensor) or semantically valid (i.e., passing the libraries' input validation).

Existing approaches. Existing works propose methods of partially addressing the first challenge of selecting inputs satisfying the function's input constraints, i.e., generating valid inputs. As off-the-shelf fuzzers do not encode knowledge of these constraints and cannot generate a high proportion of semantically valid inputs, Xie et al. [421] proposed `DocTer`, which infers the input constraints from API documentation. `FreeFuzz` [415] mines valid inputs of functions from open source code and resources. `DeepRel` [132], building on `FreeFuzz`, identifies pairs of similar functions using their documentation to share the mined valid inputs. `DocTer` and `DeepRel` rely on API documentation, which may not always be available or well-maintained. Therefore, they may not be able to cover all functions in the libraries' APIs [421]. Moreover, not every function would be invoked in open-source code. This motivates new techniques of *input constraint* inference *sans* documentation and high-quality sample usages.

Existing approaches do not address the second challenge of high input redundancy. They apply random mutations to change the type, value, etc. [415, 132] of a valid input or randomly select inputs based on the input constraints [421]. There are numerous possible inputs, with the majority of them triggering the same behaviors and provides no new information. This motivates methods of distinguishing inputs and systematically selecting them for invoking the library.

Our approach. In this study, we propose an approach (embodied in a tool), SkipFuzz, that infers a model of the input constraints at the same time as fuzzing the deep learning library. SkipFuzz does not require existing specifications or the collection of a wide range of seed inputs, as was done in prior work. Instead, it learns the input constraints of the API functions through fuzzing. Once inferred, the input constraints allow the generation of valid inputs. To do so, SkipFuzz employs *active learning*, which learns by interactively querying an oracle. In SkipFuzz, the test executor takes the role of the oracle by constructing and executing test cases invoking the

library based on the queries. The *test outcomes* (e.g., if the input is valid, invalid, or a crashing input) are provided back to the active learner. For successful inference of the input constraints, the active learner queries the test executor with a wide range of inputs that satisfy/violate different possible input constraints. This enables fuzzing with less redundancy.

SkipFuzz leverages findings from prior studies [210, 211, 205, 421]. SkipFuzz employs a set of input *properties*, by which test inputs are distinguished. The design of the input properties is based on the input constraints and root causes of bugs identified in these studies. SkipFuzz seeks to reduce redundancy by assuming that inputs with the same properties trigger the same behaviors; if the input does (not) trigger a vulnerability, then other inputs with the same properties will also (not) trigger it. The input properties differentiate inputs by their structure, shapes, values, corresponding to possible input constraints. Inputs satisfying the same properties are grouped into the same *category*. Selecting inputs from different categories allows for more input diversity and is more likely to provide new information about the input constraints.

The active learner aims to identify a hypothesis of the input constraints that is *consistent* with the observed outcomes. In the active learning literature [101], a consistent hypothesis is one where the behavior of the program under the hypothesis matches that of the actual program. Given the execution history indicating if each input was valid or invalid, an ideal hypothesis is a set of categories that contain the valid inputs but exclude the invalid inputs. We quantitatively assess the consistency between a given hypothesis and the observed test outcomes using *precision*, the proportion of observed valid inputs under the hypothesis, and *recall*, the proportion of valid inputs under the hypothesis out of all observed valid inputs. Once a hypothesis is found to be adequate, SkipFuzz generates only inputs satisfying the hypothesized input constraint, allowing for a high proportion of valid inputs to be generated.

In our experiments, SkipFuzz detects crashes in 108 functions in TensorFlow and 58 functions in PyTorch. After analyzing and removing crashes with similar root causes, the new crashes have been reported to the developers. 23 TensorFlow vulnerabilities and 6 PyTorch bug reports have been confirmed or fixed. SkipFuzz can trigger up to 65% of the crashes found by the prior approaches, `DocTer` and `DeepRel`. In a deeper analysis, we find that SkipFuzz has a greater input and output diversity, which contributes to its capability in generating crashing inputs. SkipFuzz is able to generate valid inputs for 37% of TensorFlow and PyTorch's API, while prior approaches only generate valid inputs for up to 30% of the API. When the active learner succeeds in inferring an input constraint, SkipFuzz is able to generate valid

inputs over 70% of the time, indicating that the input constraint was inferred reasonably well. This validates that active learning is effective in inferring the input constraints. Overall, SkipFuzz generates more crashing inputs than existing approaches.

6.2 Background

Deep learning libraries, such as TensorFlow and PyTorch, are employed by deep learning systems. Library vulnerabilities widen the attack surface of the software systems that depend on them [426]. These vulnerabilities may, for example, allow denial-of-service attacks on software systems using them [49].

Architecture. The core functionality of the deep learning libraries is implemented in their kernels, which are written in a low-level language such as C/C++. Applications using the libraries access their functionality through their Python API. The libraries perform validation on their inputs before accessing the core library code.

Input domain of deep learning libraries. Among others, the input domain of deep learning libraries includes tensors and matrices. These inputs are complex; a tensor may be sparse or dense (corresponding to the format that the tensors are encoded), may be ragged (tensors with variable length), has a shape (dimensions of the matrix/tensor) and rank (number of linearly independent columns). Functions in the libraries' APIs may impose constraints on its inputs, for example, requiring tensors of a specific type (e.g. float) and size (e.g. a 3x3 matrix). Xie et al. [421] investigated TensorFlow's input constraints and categorized them by their **structure** (e.g., a list), **type** (e.g., tensor containing 'float' values), **shape** (e.g., a 2-d tensor), and valid **values** (e.g., non-negative integers)

Bugs in deep learning libraries. Previous studies [211, 205, 210] have empirically analyzed bugs in deep learning programs. Jia et al. [210] reported that common root causes of bugs within TensorFlow include **type confusion** (incorrect assumptions about a variable type), **dimension mismatches** (inadequate checks of a variable's shape), and unhandled **corner cases** (usually related to incorrect handling of a specific variable's value, e.g. unhandled division by zero errors). The overlap between the root causes of bugs and input constraints suggests that distinguishing inputs by these properties would help in both finding bugs and inferring input constraints.

Testing deep learning libraries. Several recent works [323, 412, 178, 403] mutate deep learning models for testing deep learning libraries. Subsequently, the experiments of `FreeFuzz` [415] showed that API-level testing of deep learning libraries is more effective. `FreeFuzz` is seeded with inputs

Table 6.1: A glossary of terms used in the Active Learning literature and this chapter.

Active Learning: An algorithm that learns by interactively querying an oracle.
Consistency: The extent to which executions under the inferred hypothesis matches the actual program
Input constraints: The validation checks performed by the library on its inputs
Input properties: Predicates which describe inputs
Input categories: Conjunction of input properties.
Hypothesis: A model of the input constraints as inferred by SkipFuzz. A disjunction of properties associated with a set of input categories.

from publicly available code, models, and library test cases.

To effectively test the deep learning libraries, the inputs selected by the fuzzer should be semantically valid, i.e. they should satisfy the input validation checks of the API function. Otherwise, the core functionality of the libraries would not be tested. To address this, `DocTer` [421] was proposed to exploit the libraries' consistently structured documentation to extract input constraints. Still, as the API documentation is incomplete, manual annotation is required for part of the API and `DocTer` achieves a valid input generation rate of only 33%.

`DeepRel` [132] and `FreeFuzz` [415] use seed inputs collected from publicly available resources (e.g., publicly available deep learning models, documentation, developer test suites). `FreeFuzz` invokes functions in the API for which a valid invocation was observed from the resources. Building on `FreeFuzz`, `DeepRel` generates valid inputs for some functions without seed inputs by using the *similarity* of pairs of functions to transfer inputs from test cases of similar API functions to other functions without seed inputs. The similarity of functions is determined based on the function signature and documentation, which may limit it to well-documented functions. The existing approaches do not have a method of systematically selecting inputs to reduce redundancy.

Table 6.2: Examples of input property templates. X refers to the input. C and C1 refer to constant values, which can be replaced with concrete values to instantiate a property. SkipFuzz uses a total of 92 property templates, which can be viewed on the artifact website [48].

Property group	**Example**	**Description**
Type	isinstance(X, type)	the type of the input (e.g. a list)
Structure	X.dtype = type	the type (e.g. int) of elements in a tensor/matrix
Value	X < C	ranges of values
	all(X > C)	ranges of values of elements in a tensor/data structure
	X[C] = C1	value of a specific element
Shape	len(X) < C	length/size of a data structure
	X.shape.rank > C	rank of a matrix
	X.shape[C] == C2	size of a specific dimension

6.3 Preliminaries

6.3.1 Active Learning

We apply active learning for input constraint inference. To infer and refine a model of a function's input constraints, our approach generates inputs that provide new information when they are used to invoke a function. Table 6.1 presents a glossary of terms used in the active learning phase of SkipFuzz.

In active learning [63, 64], a learner sends queries to an oracle who responds with some feedback (e.g., the ground truth label of a given data instance). When active learning is employed for inferring a model of a program, a *hypothesis* is a possible model. A hypothesis is *consistent* if the behavior expected from the model matches the actual behavior of the program.

In this chapter, active learning is done while fuzzing the deep learning libraries. Our study combines active learning with fuzzing to learn the input constraints of an API. Fuzzing is used to learn a model of the input constraints, which are, in turn, used to improve fuzzing by enabling the generation of semantically valid inputs.

Input constraint inference. Our approach, SkipFuzz, aims to infer accurate models of the *input constraints* of the functions in the deep learning libraries' API. Input constraints refer to the conditions on the inputs that are expected to be fulfilled for the function to be successfully invoked. Code

```
# Input 1
# property: type(X) == RaggedTensor
input1 = tf.ragged.constant([[1,1,1,1], [2]])

# Input 2
# property: X.dtype = [('qint32', '<i4')]
input2 = tf.constant(np.zeros((1,1,1,1),
  [('qint32', '<i4')]))
```

Figure 6.1: Example of inputs constructed using `tf.constant` differing in their properties

in the library typically performs validation checks on the inputs, ensuring that the constraints are satisfied before executing the core functionality of the library. SkipFuzz refines a *hypothesis* of the input constraints of the API functions as *queries* are made to the test executor. The test executor answers the queries by checking if an input with properties corresponding to the query satisfies the input constraint (i.e., if it is *valid*, *invalid*, *crash*) determined by observing if the function was invoked without error (*valid*), rejects the input through an exception (*invalid*), or crashes the program. A crashing input is one that causes the library to terminate in an unclean manner (e.g. segmentation faults), which leads to a denial-of-service.

6.3.2 Input properties

SkipFuzz characterizes inputs to the deep learning libraries by *input properties*. The properties are used by SkipFuzz to distinguish inputs from one another. Some examples of the properties are given in Table 6.2. The properties were designed based on previous empirical studies of deep learning libraries, which found that the common root causes of bugs are **type confusion**, **dimension mismatches**, and unhandled **corner cases**. The root causes motivate properties related to an input's *type*, *shape*, and *value*, respectively.

Two example inputs are shown in Figure 6.1. These inputs, which are constructed using `tf.constant`, are both `Tensors`. However, they both satisfy at least one property that is not satisfied by the other. As they do not share the same properties, they are more likely to trigger different behaviors when passed to the same function.

6.3.3 Input categories

As each input can satisfy multiple input properties, SkipFuzz characterizes each input with the properties that it satisfies. As a pre-processing step of fuzzing, SkipFuzz groups inputs that satisfy the same properties. An *input category* is a conjunction of input properties and is associated with the inputs that satisfy the conjunction of properties.

As all inputs in a category satisfy the same conjunction of properties, they all satisfy the input constraints corresponding to the properties. For example, an input of a category with the property `X.shape.rank == 4` will pass the validation checks of a function requiring an input of rank 4.

Definition 1. *Two inputs, x and y,* ***belong to the same input category****, C, if every property that x satisfies matches a property that y satisfies, and vice versa.*

A category contains the inputs that satisfy the same properties We assume that the true input constraints of the function corresponding to a set of input categories, i.e., a collection of properties describing valid inputs. For example, all inputs in the input category associated with {`X is not None`, `X.shape = (2,2)` } are tensors of the same shape and will satisfy input constraints requiring tensors of this shape. The execution of multiple test cases selecting inputs from different categories provides information about the function's true input constraints. As such, the input categories allow the systematic selection of inputs during fuzzing.

Definition 2. *An input category, C_1, is* ***weaker*** *than an input category, C_2, if the set of inputs associated with C_1 is a superset of the set of inputs associated with C_2.*

SkipFuzz orders the categories by their strengths. One category, C_1, is stronger than another if it has input properties that are associated with inputs that are a subset of the inputs associated with the other category, C_2. Intuitively, if the input constraints of a function match the properties of an input category, c_1, then we assume that inputs from a stronger category, c_2, would be valid. Inputs from the stronger category will observe the same input properties of the weaker category, and will satisfy the corresponding input constraints.

For example, given a first category associated with the set {`X is not None`, `X.shape = (2,2)` }, and a second category associated with {`X is not None`, `X.shape = (2,2)`, $\forall$ `x` $\in$ `X (x > 0)` }, the first category is weaker than the second category (as the first category has fewer properties to satisfy).

If the first category is already a match for the actual input constraints, then we expect that the inputs from the second category would be valid.

SkipFuzz maintains a mapping of the input categories to the inputs satisfying the associated properties. This enables it to quickly sample the inputs during fuzzing. The input categories also allow for the sampling of fewer redundant inputs. An input sampled from an input category will satisfy all properties associated with the input category. To obtain some evidence that inputs from a category satisfy the actual input constraints of a function, SkipFuzz observes the outcome of invoking the function with inputs sampled from the category.

Definition 3. *A **hypothesis** is a disjunction of properties associated with a set of input categories.*

SkipFuzz models the input constraints of a function as a set of input categories. The active learner infers and refines its *hypothesis* of the true input constraints expressed as a disjunction of the properties associated with the input categories. A disjunction of input categories is used because the functions in TensorFlow and PyTorch allow for a union of types, a common feature of dynamically typed languages, e.g., Python. For example, a hypothesis can be constructed by the selection of two input categories, one associated with the set {`X is not None`, `X.type = list`, `len(X) = 4` }, and another with the set {`X is not None`, `type(X) == Tensor)`, `X.dtype == tf.int64` }. The hypothesis captures a different set of properties depending on the input's type. Once SkipFuzz considers a hypothesis of the input constraints to be adequate, SkipFuzz then generates test cases using inputs expected to be valid according to the inferred input constraints.

6.3.4 Motivating Example

Figure 6.2 shows an example of a test case generated for the API function, `tf.placeholder_with_default`. To generate semantically valid inputs that satisfy the function's input constraints, the inputs require the right type and satisfy other constraints on their shape and values. If `shape` is a list, there are other constraints such as the type or range of values of its elements. If provided an input that does not meet these constraints, the library signals that the input is *invalid* by throwing an appropriate exception.

To generate valid inputs, SkipFuzz has to discover the input constraint by invoking the function multiple times with different values of `shape` and observing the result of each invocation. A successful invocation indicates that the input satisfies the input constraints, and an unsuccessful invocation indicates otherwise. SkipFuzz forms a hypothesis regarding the constraints

```
import tensorflow as tf

# generate inputs
input1 = tf.constant([1, 2, 3])
shape = [4]

# invoke the target API function with
# the generated inputs
tf.placeholder_with_default(input1, shape)
```

Figure 6.2: Example of a test input generated for `placeholder_with_default`. Inputs for each argument (e.g. `shape`) are generated. A valid input of `shape` can be a `tf.TensorShape` **or** a `list` of `int`

of `shape`. As previously described, SkipFuzz expresses a hypothesis as a disjunction of properties so that it can capture input constraints that are a union of multiple constraints. The true constraints of the `shape` parameter permits inputs typed `list` or `tf.TensorShape`. SkipFuzz has to express one set of properties if provided a `list` and another set of constraints if provided a `tf.TensorShape`.

Once the input constraints are successfully inferred, SkipFuzz generates inputs that are *valid*, i.e., invoking the function without error, by sampling inputs from the input categories in the hypothesis. This allows SkipFuzz to generate inputs that pass the input validation checks and test the core functionality of the library. Testing the libraries with a diverse range of inputs is key to finding crashes. In Figure 6.2, if `shape` is a quantized tensor, then the library's kernel code does not correctly access its memory contents and will trigger a segmentation fault. In other words, a quantized `shape` is a crashing input. Finding a crashing input poses a challenge as the space of inputs is large and there are only a few crashing inputs. Many inputs are redundant as they share the same properties. For example, all inputs with the same wrong shape will fail the same validation check on the input shape and not reach the core functionality of the library. To this end, SkipFuzz does not get stuck with inputs that fail the same validation checks as using them does not provide SkipFuzz with new information. Instead, SkipFuzz skips past the inputs in the same category to inputs from other categories, invoking the function with more informative inputs.

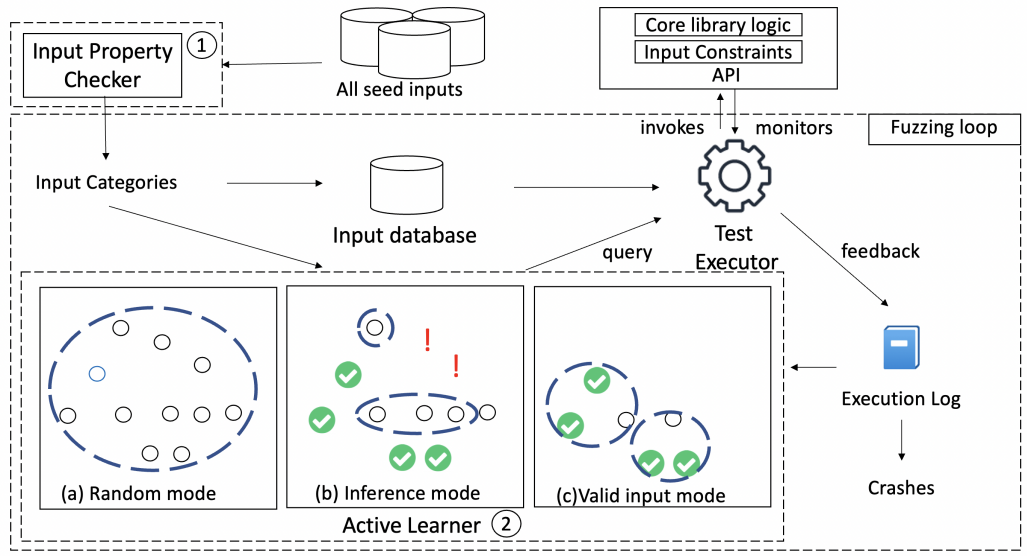


Figure 6.3: Overview of SkipFuzz. Step (1): Seed inputs (e.g. collected from the library's test suite) are grouped into input categories. Step (2): As the library is fuzzed, the active learner has 3 modes that determine what queries are posed to the test executor. It selects input categories from the input space. Initially, in its (a) *random generation* mode, it randomly selects input categories (denoted as circles). Next, in its (b) *inference* mode, it selects input categories to refine its hypothesis of the input constraints, which are formed based on the execution log which indicates the outcomes (check marks denote *valid* inputs and exclamation marks (!) denote *invalid* inputs) of prior queries. Finally, once a hypothesized input constraint is accepted, in its (c) *valid input generation* mode, it selects only inputs satisfying the hypothesis. Each blue, dashed ellipse shows the narrowing space of categories considered in each mode.

6.4 SkipFuzz

6.4.1 Overview

Figure 6.3 shows the overview of SkipFuzz. In the first step ((1) in Figure 6.3), SkipFuzz collects inputs from the execution of the library's test suite and associates them with properties that they satisfy (Section 6.4.2). Then, each input is grouped into input categories with other inputs satisfying the same properties. These inputs form the input space considered by SkipFuzz. In the second step ((2) in Figure 6.3), SkipFuzz fuzzes the deep learning

libraries. This involves the generation of test cases by selecting inputs to use as arguments in invoking the API functions. The selection of inputs involves an active learning algorithm that infers the input constraints of a target API function. The active learner constructs queries to check if an input category is a member of the input domain, i.e., its inputs are valid. The test executor has the role of the oracle; to respond to the query, it invokes the library with appropriate inputs sampled from the queried category, checking if they satisfy the actual input constraints (i.e., the function invocation does not lead to an exception or crash).

The test executor constructs test cases by sampling inputs associated with the target input categories. As it constructs and executes a test program, the invocation of the library is monitored for crashes (errors in the C++ code of the libraries that may be exploited by an attacker, e.g., segmentation faults) and exceptions thrown by the library are caught. The observation (i.e., query and the outcome of the test execution, *valid*, *invalid*, or *crash*) is written to the execution log. Considering these observations, the active learner refines its hypothesis and constructs more queries.

During fuzzing, SkipFuzz employs active learning to learn the input constraints of the API function. The fuzzing loop involves an active learner and a test executor. The active learner maintains a hypothesis of the input constraints of a given API function. To check the hypothesis, it passes queries to the test executor. Each query is one input category. On receiving the query, the test executor samples an input that satisfies the input category and constructs a Python program that invokes a function from the library's API. Each constructed Python program consists of code that generates the inputs (e.g., the variable, `shape`, in Figure 6.2) using program fragments (e.g., invocation of `tf.constant`) collected from the developer test suite. After the inputs are selected, they are passed as arguments to the API function under test (e.g., `tf.placeholder_with_default`).

When fuzzing each target function, there are three phases (described in detail in Section 6.4.3). Initially, as there is no history to support a hypothesis, SkipFuzz's randomly selects input categories from the entire input space ((a) in Figure 6.3). Afterward, the active learner begins to pose queries to the test executor for input constraint inference (described in Section 6.4.4). These queries are selected based on the hypotheses ((b) in Figure 6.3). Each query corresponds to one input category. Finally, once the hypothesis is determined to be adequately consistent, then SkipFuzz selects only inputs that satisfy the input constraints indicated by the hypothesis ((c) in Figure 6.3).

SkipFuzz maintains a list of crashing test cases. When SkipFuzz is terminated, the crashes are the output of SkipFuzz and can be inspected.

6.4.2 Step 1: Input property checking and input category construction

SkipFuzz requires seed inputs before it begins categorizing them. In our experiments, we use the developer test suite, which is readily available from the deep learning libraries' repositories, and execute them to obtain seed inputs. Before the execution of the test cases, SkipFuzz instruments the API functions. As the test cases are executed, the inputs passed as arguments to the functions of the APIs are traced. Whenever a library test case invokes the API function (either directly or transitively), the API function sequences (e.g. ⟨ `tf.constant`, `tf.ragged.constant` ⟩) for generating the argument inputs are recorded. This enables SkipFuzz to reconstruct the inputs used in the developer test suite.

SkipFuzz enumerates the possible properties for each obtained input, checking if the input satisfies the input properties. After associating the satisfied properties with every input, SkipFuzz groups them into input categories. The input categories are fixed at this time, to be later used during fuzzing. A mapping from categories to their inputs is maintained by SkipFuzz for efficient sampling of the inputs.

Reducing input redundancy. SkipFuzz leverages the input categories to reduce redundancy. As inputs from the same categories share the same properties, they will satisfy the same constraints corresponding to these properties. By selecting inputs from different categories, SkipFuzz aims to not construct multiple test cases with similar inputs that will fail the same validation checks, as it does not gain information necessary for refining its hypotheses. By avoiding the use of inputs that are similar to previously selected inputs, each test case is more likely to provide new information about the true input constraints. Hence, using inputs from different categories leads to the use of fewer redundant inputs.

6.4.3 Step 2: Active Learning-driven fuzzing

In the second step, SkipFuzz begins fuzzing the deep learning libraries. This is done through three phases.

(a) Random inputs generation. SkipFuzz begins generating test cases for each target function by selecting inputs from random categories. This phase ends once SkipFuzz successfully generates a test case with valid inputs (i.e., the function executes without error using the inputs).

(b) Input constraint inference. Once a valid input has been identified, SkipFuzz is able to form hypotheses of the input constraints (later described in Section 6.4.4). SkipFuzz tests the hypothesis that is most consistent with

Algorithm 8: The fuzzing loop of SkipFuzz which involves the active learner posing queries to the test executor.

```
   function fuzz(input_categories):
1  history = []
2  while not done do
3  |   selected_category ← active_learner(input_categories,history)
4  |   if selected_category == null then
5  |   |   random_category = sample(input_categories)
6  |   |   input ← sample(random_category)
7  |   else
8  |   |   input ← sample(selected_category)
9  |   end
10 |   tc ← construct_test_case(input)
11 |   outcome ← execute(tc)
12 |   update(history, selected_category, outcome)
13 end
```

the observations by selecting queries based on the hypothesis. It selects input categories from which inputs should be valid according to the hypothesis, as well as categories from which invalid inputs should be produced. Through interacting with the test executor, the active learner refines the hypothesis.

The active learner forms hypotheses of the correct input constraint, assessing them by quantitative measures of consistency. These measures are computed using the number of observed valid and invalid inputs that correctly and incorrectly satisfy the hypothesized input constraints.

(c) Valid input generation. Once SkipFuzz considers a hypothesis adequately consistent, SkipFuzz begins to construct test cases with inputs that are valid according to the hypothesized input constraints. This is done by sampling inputs from the input categories that are part of or are stronger than the hypothesis.

The procedure for selecting one argument given an API function is given in Algorithm 8. Initially, SkipFuzz begins the fuzzing campaign with purely random inputs as the active learner is not able to construct queries without previously observed executions (lines 4–6). After one valid input is observed, the active learner begins to pose queries to the test executor, which constructs test cases based on the queries (line 3). When SkipFuzz has entered its valid input generation mode, the active learning only poses queries to guide the selection of inputs that are expected to be valid according to the hypothesis. Given the input category in a query, the fuzzer selects a random input that

is associated with the input category (line 8). With the selected input, a test case is constructed and then executed to invoke the library (lines 10–11). The outcome of the test execution is written to the execution log (line 12), *history*, which is used in the next iteration by the active learner to pose a new query.

6.4.4 Input constraint inference

The key novelty of SkipFuzz is that it learns the input constraints while fuzzing the API function (in (b) of step (2) in Figure 6.3). Through the interaction of the active learner with a test executor, the active learner records the test outcomes in the execution log. These observations are used to form and refine hypotheses of the input constraints, and for the active learner to pose queries to the test executor.

Selecting queries based on a hypothesis. We refer again to Table 6.1, the glossary of terms used in the active learning phase of SkipFuzz. The active learner in SkipFuzz poses *queries* to the test executor to check if its *hypothesis* of the actual input constraints indeed match the true input constraints. If the hypothesis is a match, then inputs satisfying the hypothesis should be accepted by the library while inputs that do not satisfy the hypothesis should be rejected by the library. Hence, we expect that inputs from the input categories of the hypothesis should lead to *valid* outcomes. Conversely, inputs that are missing at least one property in a category of the hypothesis should be rejected. We expect that these queries should result in *invalid* outcomes. If these queries lead to valid outcomes, then it implies that the hypothesis is stronger (see Definition 2 in Section 6.3.3) than the true input constraints.

As such, for one hypothesis, the active learner constructs several queries by selecting input categories with respect to the input categories that compose the hypothesis. One set of queries checks that inputs satisfying the hypothesis also indeed satisfy the actual input constraints (i.e., the test constructed will be executed without error). Another set of queries checks if the inputs that do not satisfy the hypothesis also do not satisfy the input constraints (i.e., the test constructed results in an error when executed).

At the beginning of the fuzzing campaign, SkipFuzz selects random inputs. As the fuzzing proceeds, the active learner begins to pose queries to the test executor. The active learner considers the execution log to select input categories to form a hypothesis, and selects input categories as queries. It optimizes for the confirmation of possible hypotheses of the input constraints of the API function.

As the test executor component evaluates a test case, the query (i.e.,

choice of input category) and test outcome are written in the execution log. If a test case results in an exception thrown by the deep learning library, then the input is *invalid*. If the library invocation succeeds without any exceptions, then the input is *valid*. If the test case crashes the deep learning library, then the input is a *crash*ing input.

Measuring consistency. At any given time, there may be multiple hypotheses that can be considered by the active learner. The active learner selects the hypothesis that is the most consistent with the observations in the execution log. To do so, it quantitatively measures the number of valid and invalid inputs that are consistent with the hypothesis. Given a perfectly consistent hypothesis, all valid inputs will be included in an input category in the hypothesis. Conversely, all invalid inputs should not be a member of the hypothesis.

As it may not be possible to infer a perfectly consistent hypothesis, we compute quantitative measures of a hypothesis' *consistency*. Each hypothesis proposed by the active learner is assessed on its consistency with regard to the observations in the execution log; valid inputs should satisfy the input constraints in the hypothesis and invalid inputs should not. Within SkipFuzz, we do not expect that the hypothesis will perfectly match the input constraints. As such, SkipFuzz assesses each hypothesis and selects one that is the most consistent with the observed executions. A good hypothesis includes input constraints that cover a large part, if not all, of the valid observations. It should also not incorrectly cover invalid inputs. SkipFuzz uses precision and recall to assess the quality of a hypothesis. Out of `all` inputs selected, given that `covered(valid, hypothesis)` represents the number of valid inputs that fall within the hypothesis, and `covered(all, hypothesis)` represents the number of inputs, both valid and invalid, that fall within the hypothesis. The precision, P, and recall, R, are computed as follows:

$$P = \frac{covered(valid, hypothesis)}{covered(all, hypothesis)}$$

$$R = \frac{covered(valid, hypothesis)}{|valid|}$$

Precision measures the proportion of valid inputs that fall within the hypothesized input constraints out of all the observed inputs. Recall measures the proportion of valid inputs that fall within the hypothesized input constraints out of all the observed valid inputs. Together, the two metrics measure the adequacy of the hypothesized input constraint. A hypothesis is adequately consistent if the precision and recall exceed a threshold set at the start of the fuzzing campaign.

6.5 Implementation

In the previous section, we have discussed the key ideas behind SkipFuzz. Here, we discuss the implementation details.

Building the input database. SkipFuzz is implemented as a Python program that takes the API and the developer test suite as its input. The list of functions in the API is obtained. We obtain the input values used in the library test suite as the seed inputs for SkipFuzz, the Python library code is instrumented to track the invocation of every function call to record their argument inputs. The functions to construct the inputs, the returned values of their invocations, and the input properties satisfied by the inputs are stored in the database. Inputs are generated by fetching and invoking the functions.

Crash Oracle. As our research objective in this study is to assess the ability of SkipFuzz to explore the input space, we only monitor the deep learning libraries for crashing inputs. SkipFuzz is implemented with a crash oracle. The test executor constructs and executes a test program on a different process. Then, the test process is monitored for crashes. Inputs that crash the library are written to the execution log. These crashes are later investigated manually to identify unique crashes before we report them to TensorFlow and PyTorch.

The crash oracle detects weaknesses considered as security vulnerabilities [49] (e.g. segmentation faults) that cause the running process to terminate in an unclean way. Other methods of detecting vulnerabilities may be implemented in SkipFuzz in the future, but in our experiments, we focused on uncovering crashes in the libraries that may be exploited for denial-of-service attacks.

Active Learning. The active learner takes the execution log as input and produces a series of queries to be posed to the test executor. The queries are constructed based on the subset of input categories in the hypothesis. The construction of a hypothesis and the selection of queries are obtained through the execution of a logic program. Using a logic program allows us to declaratively express the desired characteristics of a hypothesis and optimize the selection of input categories against a criteria. The active learner selects an appropriate hypothesis while maximizing the number of valid inputs that match the hypothesis, minimizing the number of invalid inputs that are incorrectly matched by the hypothesized input constraint, and favouring simpler hypothesis by minimizing the number of input categories used in the hypothesis. In this way, SkipFuzz assesses each hypothesis on its consistency with the observed test outcomes.

SkipFuzz accepts a hypothesized input constraint considering if its precision and recall exceed a threshold. In our experiments, we set a low threshold for both precision and recall at 0.25. This enables the input constraints to be inferred for a large proportion of the API. As our goal is to fuzz the API thoroughly, we find allowing the fuzzer to focus on a broad region of inputs that include the valid domain of inputs of the functions is more beneficial than precisely identifying the valid domain of inputs. We empirically find that the low thresholds do not adversely impact the proportion of valid inputs selected by SkipFuzz when using the hypothesized input constraints to generate valid inputs. This is because the logic program already optimizes the selection of hypotheses for a high level of consistency.

Interleaving of target functions. The active learner SkipFuzz employs `clingo` [168] to execute the logic programs used by SkipFuzz to select the next set of inputs. Logic programs take a significant amount of time to be executed to produce their output. To allow time for the logic program to be executed, SkipFuzz interleaves the construction of test cases for different API functions, coming back to the same function only after completing a test case for each of the other test cases. This provides ample time for the logic program to be run before the same API function is tested again.

6.6 Evaluation

6.6.1 Research Questions

Our experiments aim to provide answers to the following research questions. We investigate SkipFuzz capability in finding crashing inputs (RQ1). Next, we analyze the input generation ability of SkipFuzz (RQ2 – RQ4).

RQ1. Does SkipFuzz produce crashing inputs?

This question concerns the ability of SkipFuzz in triggering crashes, which is our primary objective. We count the number of new crashes that have not been previously reported, which we then reported to the library developers for validation. We also compare the ability of each approach in triggering the set of crashes found by at least one approach.

RQ2. Does SkipFuzz sample diverse inputs?

Active learning should enable SkipFuzz to reduce redundancy during fuzzing by selecting a wide range of input categories. We investigate if SkipFuzz was able to do so. We compare SkipFuzz against the baselines and compare the inputs generated to fuzz the functions known to crash.

RQ3. Does SkipFuzz sample valid inputs?

SkipFuzz is expected to generate a larger proportion of valid inputs. We investigate if inputs selected to satisfy the inferred input constraints are indeed valid inputs.

RQ4. Which components of SkipFuzz contribute to its ability to find crashing inputs?

SkipFuzz aims to have a less redundant selection of inputs and generate a higher proportion of valid inputs. We perform an ablation study to determine how the components of SkipFuzz contribute to it.

6.6.2 Experimental Setup

Baselines. We compare SkipFuzz against the state-of-the-art deep learning library fuzzers targeting the libraries' API, `DeepRel` [132] and `DocTer` [421]. We run the tools from their replication packages and analyze the list of bugs reported.

`DeepRel` builds on top of `FreeFuzz` [415], using the same strategy of generating inputs for each API function. `DeepRel` and `FreeFuzz` collect inputs for use from open-source code on GitHub, publicly available models, and the library test suite. Compared to `FreeFuzz` and `DeepRel`, SkipFuzz uses only inputs from the libraries' test suites while `DeepRel` and `FreeFuzz` use seed inputs collected from open source resources. As `DeepRel` and `FreeFuzz` uses the same input generation strategy and differ only in the number of API functions they cover, we only compare SkipFuzz against `DeepRel`.

`DocTer` extracts input constraints from the library documentation. Then, it generates inputs to invoke the libraries considering the extracted input constraints.

Environment. We run experiments on TensorFlow 2.7.0 and PyTorch 1.10, the same version of the libraries used by the most recent study [132]. We collect a list of all API functions of TensorFlow and PyTorch. It is used in our initial experiments, where we attempt to run the approaches on every function. Subsequently, we focus our analysis on the ability of the fuzzers to trigger the crashes found by the approaches. Using `DocTer`, `DeepRel`, and SkipFuzz, there are crashing inputs to 231 functions in TensorFlow and 95 functions in PyTorch.

We configured and ran the fuzzers for up to 48 hours. In the prior experiments of the baseline fuzzers [421, 415, 132], the tools were allowed up to 1,000 [415, 132] or 2,000 [421] executions for each function. To generate 1,000 test cases, we executed `DeepRel` and it took 172 hours and 43 hours to complete generating test cases for TensorFlow and PyTorch. `DocTer` took 16 hours for TensorFlow and 25 hours on PyTorch. Therefore, to use the same

Table 6.3: The number of unique new vulnerabilities of TensorFlow reported in this study and prior studies.

Approach	# new vulnerabilities
`DocTer`	1
`FreeFuzz`	7
`DeepRel`	1
SkipFuzz	23

budget for a fair comparison, we tweaked the number of test cases generated by the baseline fuzzers to fit in 48 hours and reran the fuzzers.

Our experiments on executed on a machine with an Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, 205G, Tesla P100. While our fuzzer does not directly use the GPU, some functions in the library may use the GPU.

Evaluation Metrics.

We use the following metrics to assess SkipFuzz:

- **# of detected crashes.** The primary goal of SkipFuzz is to generate inputs to crash TensorFlow and PyTorch.
- **Input property coverage.** We report the number of unique input properties that have been satisfied by at least one input during fuzzing. To reduce input redundancy, a high input diversity is desirable. To measure the diversity of inputs, we count the total number of unique input properties observed to be satisfied at least once.
- **API coverage.** We report the number of functions that valid inputs were successfully generated for. This metric was previously used in the evaluation of `DeepRel` [132]. Related to this metric, we also report the proportion of generated valid inputs.

6.6.3 Experimental Results

RQ1. Vulnerabilities detected

Existing crashes. We perform a thorough analysis of the capability of the approaches in detecting existing crashes. In this analysis, we consider all crashes found by the approaches. To perform this analysis, we consider that a vulnerability was not detected if its corresponding API function is not

covered by the tool or if the tool does not report the bug although test cases were generated for the function.

In total, SkipFuzz detects a total of 168 crashing functions, 108 in TensorFlow version 2.7.0 and 58 in PyTorch version 1.10. From the 108 TensorFlow functions, we grouped related crashes and reported 43 vulnerabilities. From the 58 PyTorch functions, we reported 10 vulnerabilities. After corresponding with the developers of TensorFlow and PyTorch, they confirmed that 23 of the TensorFlow vulnerabilities and 5 of the PyTorch vulnerabilities were previously unknown. The remaining crashes are confirmed as vulnerabilities too, but they were already known by the developers (although the fix was not released yet).

We received 23 CVEs from these reports. Next, we analyze the extent to which SkipFuzz, `DocTer`, and `DeepRel` are able to detect the same vulnerabilities.

`DocTer` found 163 crashing functions. Of the 108 vulnerable TensorFlow and 58 vulnerable PyTorch functions found by SkipFuzz, `DocTer` was able to successfully generate crashing inputs to 6 of the 108 vulnerable functions in TensorFlow. and 12 of the 58 vulnerable functions in PyTorch. Overall, `DocTer` detects just 18 of the 166 vulnerable functions detected by SkipFuzz.

On the other hand, when executed on the versions of libraries before these crashes were fixed, SkipFuzz is able to detect 52 (84%) out of the 62 crashing functions in TensorFlow detected by `DocTer`. On PyTorch, SkipFuzz is able to detect 7 (23%) out of the 31 crashing functions detected by `DocTer`. Overall, SkipFuzz detects 59 (63%) out of 93 crashing functions detected by `DocTer`.

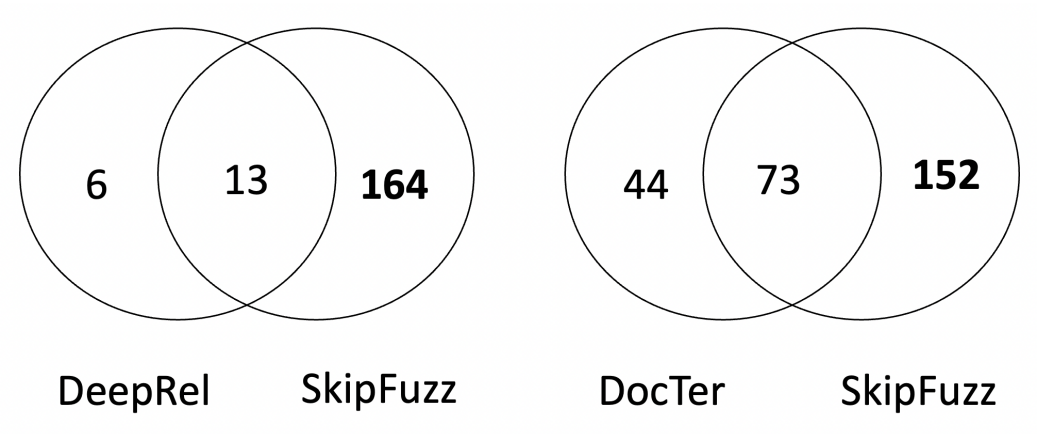


Figure 6.4: Crashes in the deep learning libraries found by `DeepRel`, `DocTer`, and SkipFuzz.

Next, we compare SkipFuzz against `DeepRel` and `FreeFuzz`. `DeepRel` and `FreeFuzz` was able to detect crashes for only 9 of the 108 TensorFlow functions and 7 of the 59 PyTorch functions. The original experiments done to evaluate `FreeFuzz` [415] and `DeepRel` [415] resulted in 39 bug reports on TensorFlow, of which 10 involved crashes, and 72 bug reports on PyTorch, of which 7 involved crashes. When executed on versions of the libraries before the crashes were fixed, SkipFuzz is able to detect 8 (80%) of the 10 crashes on TensorFlow and 3 (43%) of the 7 crashes on PyTorch. Figure 6.4 shows two Venn diagrams of the functions that each approach is able to generate crashing inputs to.

New vulnerabilities in TensorFlow. Table 6.3 shows the number of new crashes found in the experiments of `DocTer`, `FreeFuzz`, and `DeepRel`. On TensorFlow, we determine if a crash is new by going through the list of TensorFlow vulnerability reports and comparing the referenced bug reports against the bug reports referenced by the replication packages of the prior approaches. `DocTer` [421] found 1 newly discovered vulnerability. `FreeFuzz` [415] and `DeepRel` [132] found a total of 8 crashes. In our experiments, SkipFuzz detects 33 new vulnerabilities. 23 of them have been confirmed by TensorFlow developers to be new vulnerabilities, with 23 CVE IDs assigned. We do not perform this analysis for the crashing inputs to PyTorch as its developers do not assign CVEs to potential security weaknesses.

There has been significant effort in detecting TensorFlow vulnerabilities. Apart from the baseline approaches discussed, it is a fuzz target in the OSS-Fuzz project [47]. OSS-Fuzz has found over 30,000 bugs in open source projects, including 6 security bugs in TensorFlow[1]. Evidently, finding new vulnerabilities is not trivial.

RQ2. Reducing input redundancy.

To investigate the factors contributing to SkipFuzz's performance, we study the inputs used in fuzzing. We evaluate the reduction in redundancy by measuring input diversity.

We analyze the coverage of input properties by the inputs generated by the approaches. A higher coverage of input properties indicates a greater diversity of inputs. This suggests a low amount of redundancy in input generation. Conversely, a low coverage may indicate that similar inputs was generated over and over again, which implies a high level of redundancy as the inputs may be failing the same validation checks, or triggering the same library behavior. We investigate the number of properties that were satisfied

[1]Issues tagged "Bug-Security" on `https://bugs.chromium.org/p/oss-fuzz/issues/list?sort=-opened&can=1&q=proj:TensorFlow`

Table 6.4: Coverage of input properties

Approach	% input properties covered
`DocTer`	15%
`DeepRel`	16%
SkipFuzz	**31%**

Table 6.5: The distribution of outcomes running the test cases produced by `DeepRel`, SkipFuzz, and `DocTer`. SkipFuzz produces test cases with more diverse outcomes, indicating that it is better at uncovering corner cases. IAE refers to `InvalidArgumentError`

Exception type	DeepRel	DocTer	SkipFuzz
None (no errors)	77%	13%	24%
IAE	15%	<1%	2%
`ValueError`	6%	41%	7%
`TypeError`	1%	46%	19%
Other errors	<1%	<1%	48%

by an input passed to TensorFlow's and PyTorch's API as a proportion of all input properties observed in the experiments. In this analysis, we focus on the test cases generated to target the crashing functions to investigate the reason for SkipFuzz's stronger ability to generate crashing inputs.

Input diversity. Table 6.4 shows the experimental results. The inputs used by SkipFuzz in its test cases cover two times more input properties than the inputs used in the test cases generated by `DeepRel` and `DocTer`. While the inputs selected by `DeepRel` and `DocTer` cover only 16% and 15% of the possible input properties, SkipFuzz achieves an average of 37% property coverage. This suggests that the diversity of the inputs contributed to the stronger performance of SkipFuzz.

Output diversity. We further investigate if the increased input diversity leads to more diverse library behaviors. To do so, we analyze the result of each generated test case by investigating the number of occurrences of each type of input constraint that was not satisfied. We count the number of times each type of exception is thrown. Note that we do not consider crashes among these outcomes (crashes are rare occurrences leading to the termination of the running process).

The proportion of each result type (e.g., a successful run without errors, or a particular exception type) is shown in Table 6.5. SkipFuzz achieves a distribution with diverse test outcomes while `DeepRel` has a greater proportion of successful executions of TensorFlow. We stress that a high proportion

of valid inputs is usually desirable but maybe achieved at the cost of failing to explore uncovered behaviors, for example, if a fuzzer uses the same valid input over and over again. Our experimental result suggests that SkipFuzz achieves a high diversity of outcomes; SkipFuzz triggers up to 30 types of exceptions, ranging from `RecursionOverflow`, `UnicodeDecodeError`, and `ResourceExhaustedError` (categorized as "Other errors" in Table 6.5), while `DeepRel` triggers only 7 types of exceptions. `DocTer` only triggers 4 types of exceptions, with the vast majority of them `InvalidArgumentError` and `ValueError`. The experimental results validate our finding that SkipFuzz successfully triggers a greater number of different behaviors (and corner cases) compared to `DeepRel` and `DocTer`.

RQ3. Generating valid inputs

SkipFuzz performs input constraint inference. We study if the inferred input constraints are precise enough for producing inputs satisfying the actual input constraints.

`DocTer` generates just valid inputs 13% of the time, underperforming SkipFuzz which produces valid inputs 24% of the time. This validates our initial intuition for using active learning. The better performance of SkipFuzz in generating valid inputs indicates that active learning may be more successful in inferring input constraints than `DocTer`'s use of the API documentation, which may be incomplete [421].

`DeepRel` uses `FreeFuzz` as its test generator, and therefore, will produce the same output as FreeFuzz. Their input generation strategy is to mutate the seed inputs. As seed inputs are always semantically valid inputs, the vast majority of inputs generated by `FreeFuzz` and `DeepRel` are valid. However, as discussed in the previous section, a large proportion of valid inputs may imply that the fuzzer is using similar inputs repeatedly, leading to redundancies. Indeed, based on Table 6.5, `DeepRel` has a lower diversity of inputs, which may have led to a lower chance of generating crashing inputs (Table 6.3).

API Coverage. Table 6.6 shows the API Coverage obtained by SkipFuzz and the baseline tools. SkipFuzz successfully invokes 37% of the functions in the API. In contrast, the strongest baseline, `DeepRel`, generates valid inputs for 30% of the API functions. The results indicate that SkipFuzz is able to generate valid inputs for a greater proportion of the API than existing techniques.

Table 6.7 shows the proportion of valid inputs generated. We compare SkipFuzz against `DocTer` as well as a simple baseline that randomly selects inputs used in the libraries' test suite. While SkipFuzz generates valid

Table 6.6: Coverage of the functions in the API. We consider an API function covered if the tool generates valid inputs. The numbers in parenthesis indicate the proportion of the API that SkipFuzz accepts the hypothesized input constraints for.

Approach	API Coverage	# of functions covered
`DocTer`	12%	956
`DeepRel`	30%	1902
SkipFuzz	37%	2362

Table 6.7: Proportion of valid inputs generated

Approach	% of valid test cases
Random selection of inputs	1%
`DocTer`	13%
SkipFuzz	24%
`DeepRel`	77%
SkipFuzz (valid input mode)	**80%**

inputs 24% of the time considering all three input generation modes, SkipFuzz produces valid inputs 80% of the time in its valid input generation mode (after inferring the input constraints). This is higher than the proportion of valid inputs generated by both `DocTer` and `DeepRel`. Overall, this demonstrates the benefit of the active selection approach for input constraint inference.

RQ4. Ablation analysis

For a deeper analysis, we perform an ablation study on SkipFuzz. SkipFuzz^{-} refers to a version of SkipFuzz where inputs are sampled from the input categories, but there is no active learner posing queries and no inference of the input constraints (removing (2) in Figure 6.3). SkipFuzz^{--} refer to a version of SkipFuzz where inputs are selected randomly (removing both (1) and (2) in Figure 6.3).

Table 6.8 shows the experimental results of the ablation analysis. Without using active learning to infer input constraints, the number of crashes found by SkipFuzz^{-} drops from 168 to 122, a 26% decline. Without using active learning, SkipFuzz^{-} does not drive the test executor toward valid inputs. While it is able to cover a higher proportion of properties (93%), the majority of the inputs (99%) are invalid.

Without the input properties, SkipFuzz^{--} selects inputs entirely at ran-

Table 6.8: Ablation analysis of the components in SkipFuzz. % valid is the proportion of valid inputs that are generated. SkipFuzz^{-} removes active learning. SkipFuzz^{--} removes the use of input properties and active learning.

Approach	Property coverage	% valid	# crashes
SkipFuzz	31%	**24%**	**168**
SkipFuzz^{-}	**93%**	1%	112
SkipFuzz^{--}	84%	1%	52

dom. The number of detected crashes substantially drops to just 52, a third of the original number of crashing inputs found. The majority of inputs selected are invalid; only 1% of them are valid. It spends most of its test budget using inputs that invokes the library with errors.

The experimental results indicates that higher input diversity alone is not enough. Having a valid input proportion that is too low hinders the ability to find crashing inputs. Overall, our experimental results suggest that the input properties are essential to SkipFuzz and that active learning substantially boosts the effectiveness of SkipFuzz.

6.7 Discussion and Limitations

Our experiments demonstrate that SkipFuzz outperforms the existing fuzzers in generating crashing inputs to TensorFlow and PyTorch. Our analysis suggests that SkipFuzz is effective due to the combination of both the higher diversity of inputs and the higher proportion of valid inputs. These improvements stem from the effectiveness of active learning in input constraint inference.

Effectiveness of active learning. Active learning is effective in our task as we encoded the domain knowledge of deep learning libraries in the input properties. This allows SkipFuzz to successfully infer the input constraints. Had the input properties not correctly encoded the input constraints, a hypothesis would not express meaningful properties. Once SkipFuzz infers the input constraints, the majority of inputs generated are valid. This is an improvement compared to the prior approach of extracting constraints from documentation.

Limitations. Next, we discuss some limitations of SkipFuzz. The active learner poses queries that are answered through the invocation of the library. This is a form of dynamic program analysis. it, therefore, inherits the

limitations of dynamic analysis; the observed behaviors are an underapproximation of the actual behaviors of a program. Consequently, the model of the input constraints hypothesized by SkipFuzz may not capture some properties of the true input constraints of the library. We leave the investigation of other methods of input constraint inference for future work.

6.8 Related Work

Fuzzing deep learning models and systems. Researchers have proposed approaches to assess the security of deep learning systems. Existing approaches fuzzes either deep learning models [425, 177, 138] or larger systems that use deep learning [424, 183, 141, 72, 371, 457]. Other approaches use static analysis [245, 268]. Some studies reveal that software deploying deep learning does not secure their models well; the weights of models can be stolen by querying the model repeatedly [217, 208]. SkipFuzz fuzzes deep learning libraries rather than individual models or systems that use deep learning.

Fuzzing deep learning libraries. Several approaches have been proposed for testing deep learning libraries. Several approaches detect bugs through metamorphic and differential testing [323, 403, 412, 178, 425]. These approaches check for different behaviors when the same behavior is expected, e.g. a similar function invoked with the same inputs on TensorFlow and PyTorch. Another approach targets precision errors in TensorFlow [451]. Crucially, these previous studies overlook the systematic selection of inputs for minimizing redundancy.

ExAIS [359] uses specifications of the deep learning layers for fuzzing. As it requires expert analysis and manual writing of specifications, its scalability is limited. The closest approaches to SkipFuzz are `DocTer` [421] and `DeepRel` [132], which have been discussed and used in our experiments.

Fuzzing other libraries. Recent research has also proposed to fuzz libraries. Some approaches aim to generating valid inputs for libraries in specific languages, e.g. Rust [213, 376]. Some studies propose approaches for constructing fuzz drivers [76, 206, 443], e.g. library calls to prepare the complex inputs required to invoke the library. SkipFuzz has a similar goal of generating valid inputs but does so through active learning to infer the input constraints.

Selecting inputs. Several studies [337, 399, 309, 459, 114] propose methods of selecting good inputs for fuzzing. Some methods optimize for code coverage [337, 399, 309, 90] or filtering out inputs predicted not to reach a target program location [459]. Unlike these approaches, SkipFuzz selects

inputs that may glean more information about the input constraints.

Input validation. SkipFuzz addresses the problem of generating inputs that pass input validation checks through input constraint inference. Several approaches [123, 267, 185, 319, 408] use static analysis to address the problem. DriFuzz [366] proposes a method of generating high-quality initial seed inputs. Different from these approaches, SkipFuzz uses active learning to learn the input constraints to generate valid inputs.

Active Learning. Our approach uses active learning [101, 361, 63, 64], which queries an oracle and learns from its feedback. In classification tasks, active learning is used to query for labels of informative data instance when labeling every instance is too difficult [361]. Recent work uses active learning to learn models of programs, and then regenerate programs using the models to remove undesired behaviors [389, 365]. SkipFuzz uses active learning to learn models of input constraints.

6.9 Summary

In this study, we address the problem of generating crashing inputs to deep learning libraries. Our approach, SkipFuzz, uses active learning to infer the input constraints of the libraries' API during fuzzing. SkipFuzz has two advantages over existing approaches. Firstly, SkipFuzz infers the input constraints without the use of documented specifications. Secondly, its use of active learning guides the selection of a diverse set of inputs during fuzzing. These advantages address the challenge of generating semantically-valid inputs as well as the challenge of reducing input redundancy, which is only partially addressed and overlooked by the previous studies, respectively. Our experiments show that addressing both challenges is crucial. 23 CVEs have been assigned to vulnerabilities found by SkipFuzz. The source code of SkipFuzz can be found at `https://github.com/skipfuzz/skipfuzz`.

Chapter 7

(System of Interacting Components) IoTBox: Sandbox Mining to Prevent Interaction Threats in IoT Systems

7.1 Overview

Internet of Things (IoT) systems, such as smart homes, are becoming popular and widespread. There are security risks at each level of granularity in an IoT environment. At the application-level, researchers have studied the security implications of IoT platforms that allow users to install apps that allow devices to interact with one another. A smart home comprises a collection of devices and apps. These apps control and connect different devices together, bringing many benefits and convenience to users, but this comes at the price of security risks. The increased attack surface has led to new types of attack, such as those that introduce physical risks. For example, an app can be used to configure a smart home such that the windows will be opened if a temperature sensor in a room measures a reading above a user-specified temperature. A malicious app can spoof fake temperature readings to trigger the opening of the window, potentially allowing a break-in [136]. Other apps can open a door when no one is at home, disable a smoke detector, and induce seizures through the rapid strobing of lights [347]. Hence, it is important to understand the security risks of apps used in a smart home and defend against malicious behaviors.

One source of complexity is that apps can interact with one another through the devices they control and are triggered by. Malicious behavior

can arise through the interaction of multiple apps, which may interact to produce unintended results in the physical environment. This motivates the need to study a smart home as a system of interacting apps and devices, rather than asserting that the behaviors of the individual apps in the smart home are safe.

Existing work has focused on using model checking or monitoring smart homes to ensure that joint behaviors of the apps do not violate safety properties [105, 301, 106, 53, 406]. Some of these techniques extract models of the apps through static analysis [105, 301, 53], checking them against predefined safety properties that were written by hand. For example, techniques may enforce a property that the door is never unlocked while the occupants of a smart home are away, and another property may be that the lights in the house are never automatically switched off while the occupants are home.

A problem is that these properties may not anticipate all legitimate uses of the apps. Indeed, the normal executions of some apps deliberately violate these safety properties and users may install these apps knowingly. A security policy that prohibits any switches from turning on when the users are not home will prevent the use of the legitimate app, *VacationLightingDirector* [42], which simulates occupancy in a house by occasionally switching on the lights when the user is on vacation. These techniques cannot distinguish between user-intended violations of the properties from real safety problems. Moreover, as new devices and new apps are developed and introduced into the market, there will be new forms of incorrect behavior involving new apps and devices. Existing techniques cannot defend against new modes of attacks involving new channels or devices, for which safety properties have not been written yet. This motivates the need to automatically identify relevant security policies.

In this work, we propose that we can begin addressing the above mentioned limitations through the technique of *sandbox mining* [209], inspired by the work by Jamrozik et al. mining sandboxes for Android apps [209]. We propose IoTBox, which automatically mines a sandbox for a smart home. Rather than writing out safety properties by hand, IoTBox encodes the current behaviors of a smart home and protects the user against unexpected behaviors. The possible behaviors of a smart home are encapsulated in the sandbox, which detects changes in behavior. Changes in behaviors may be caused by the introduction of malicious behavior in an app, unexpected bugs due to a new interaction between apps, or the removal of behavior that the user depended on. After the sandbox is mined, it excludes behavior that was not previously captured during analysis. If an app is replaced or updated with malicious code, the sandbox prohibits any behavior that violates its rules and reports it. The user of the apps can investigate and decide if the

new behavior should be permitted. If a change is benign and acceptable to the user, the sandbox can be relearned on the updated smart home again, and will defend the smart home system against new threats.

However, unlike mining sandboxes for Android [209, 252, 79], the possible behaviors in a smart home cannot be as easily explored using test case generation. Unlike traditional software systems, actions invoked within the smart home may take several seconds before they execute. The state space, composed of every app located in the smart home, is enormous compared to individual Android apps. Yet, it is essential to comprehensively explore the possible behaviors of the smart home to learn accurate rules.

Another challenge is that malicious behavior may disable an action instead of invoking new actions. For example, an adversary may disable existing behavior that locks a door when the user is asleep, leaving it unlocked for an adversary. Consequently, a sandbox for IoT should detect changes in behavior that causes missing actions.

In this study, we overcome the above challenges by leveraging the precision of the formal models proposed in existing studies [53]. Specifically, IoTBox uses a formal model of a smart home [53] to identify a complete execution context for any automated action with the help of a model checker. During monitoring of the environment, IoTBox then uses these execution contexts to identify actions that it expects the smart home to automatically run. If there is a mismatch between the expected actions and actions in reality, then IoTBox detects that there is a behavioral change and alarms the user.

7.2 Background

In this section, we present relevant background. IoTBox builds on top of both prior studies formalizing an IoT system [53], and techniques to mine sandboxes [209, 252, 398].

7.2.1 Smart Home Platforms

We focus on the apps in the Samsung SmartThings [30] and IFTTT [19] platforms. In a typical smart home platform, physical devices have a corresponding virtual representation on the platform. The state of each device is a mapping of *attribute*s to *values*. The state of a device can be modified through *actions*, such as switching on the lights (toggling *switch.off* to *switch.on*). The set of attributes and actions of a device is determined upon

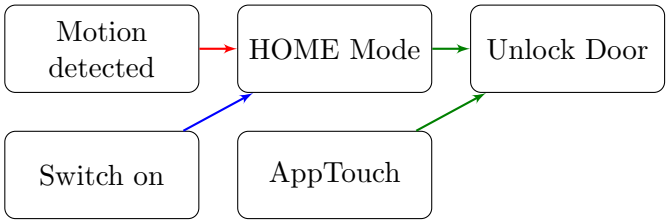


Figure 7.1: This smart home transits to *HOME* mode on detecting motion (App1) or if a light is switched on (App2). The door is unlocked when the mode has changed or through a user interaction with the SmartThings app (App 3).

registration of the physical device to the smart home platform, where it is granted a *capability* (e.g. a lock, a switch, or a thermostat).

Once registered, apps can access these devices by specifying the required capability. Within the apps written in Groovy, each device can then be accessed as an object. Device attributes are read through the attributes on the object, and the actions on the device (e.g. *door.unlock()*, *alarm.off()*) are invoked through method calls. Installed apps can interact with one another through various means. For example, an app can change the mode of the smart home to *Home*, indicating that the user is home, and a second app unlocks the door when it detects that there is a mode change.

Arbitrary apps can be installed by the user on the smart home platform. Each app has a set of capabilities it requires, in which the user binds existing devices to. Apps are written following an event-handling paradigm. Usually, app waits for events from sensor devices and trigger new actions through actuator devices. Apps can interact through devices (e.g. one app toggles a switch at a particular time of the day, and another app is triggered by the switch to turn on all the lamps in the house) or through physical channels (e.g. one app triggers a lamp in a room, which causes another app to pick up sensor readings from an illuminance sensor).

We show an example of app interactions spanning multiple apps in Figure 7.1. A first app switches the smart home to *HOME* mode after a motion sensor detects motion, a second app switches the smart home to *HOME* mode if a light is switched on, and a third app unlocks the door if it detects a transition to *HOME* mode or if the user unlocks the door through the SmartThings apps. This transitively creates a link between a motion sensor event and unlocking the door.

7.2.2 Formal model of a smart home

In this study, we use the formal model of a smart home proposed by Alhanahnah et al. [53]. They provide a tool, IoTCOM [53], to analyse apps written for the SmartThings and IFTTT platforms. IoTCOM provides a parser that translates these apps to Alloy [207] models. A smart home consists of a set of devices and apps. Each device has one or more capabilities, attributes, and at any time, a value associated with each attribute. Smart apps can connect these devices, such as invoking an actuator (e.g. unlocking a door) given a reading from a sensor (e.g. a motion detector sensing motion). A smart app is a collection of rules. Rules are tuples of a **Trigger** $\times$ a set of **Conditions** $\times$ a set of **Commands**.

Triggers represent the conditions in which the app is activated. These are often events from the smart home sensors, such as a door opening. A trigger comprises a device capability, an attribute associated with the capability, and the value of the attribute. Each rule has at most one trigger.

Conditions represent the logical predicates on the state of other devices/smart home. These predicates guard the invocation of a rule's commands. For example, after a rule is triggered by an event ("smoke detected"), a rule may have other conditions ("the door is locked"), before it executes a command ("unlock the door"). A trigger comprises a capability, an attribute associated with the capability, and the value of the attribute. Each rule may have any number of conditions.

Commands represent the actions taken by a rule, including device actuations that change the physical state of the smart home. Each command comprises a device capability, an attribute, and a value. A rule may have one or more commands.

The SmartThings platform also allow for state variables that persist over different executions of an app. Each state variable is encoded as a device capability in IoTCOM, allowing for analysis of behaviors that depend on these variables.

Another consideration is communication between devices and apps through physical channels. IoTCOM [53] includes the physical channels in its model of the smart home, and models them as a mapping of capabilities to a physical channel. Each channel can link together an actuator device (e.g. a valve) and a sensor device (e.g. a water sensor).

IoTCOM [53] precisely represents the smart home in Alloy, and uses the Alloy Analyser to assert that safety properties hold on the smart home. First, IoTCOM converts the apps in the smart home to Alloy models. Their joint behaviors are represented as a *behavioral rule graph*, which captures the behaviors of apps, linking together the triggers, conditions, and commands

of relevant rules. IoTCOM then asserts that the apps do not violate any safety property. In our work, IoTBox can be built on top of any formal model, however, as prior studies faced scalability issues [105, 212, 301], we use IoTCOM's precise model of a smart home.

7.2.3 Mining sandboxes

Traditionally, malicious programs have been executed in sandboxes, which blocks access to resources that have security concerns. Researchers [209, 79] have suggested that entirely blocking/allowing access to a certain resource may be too coarsed-grained, and have proposed to identify more granular conditions of accessing each resource, restricting access if the conditions are unmet. To do so, techniques have been proposed to automate the mining of rules for a sandbox. Jamrozik et al. [209] suggest mining associations between GUI elements and sensitive API access through the generation of Android GUI tests. This allows the identification of rules permitting access to sensitive resources, such as the camera, only if the user is performing specific actions on the app.

There are two phases to mining sandboxes, the exploration phase and the sandboxing phase. In the first phase, the behaviors of an app are explored and encoded into the sandbox. In the sandboxing phase, new behaviors that were not seen during the first phase are prohibited. If the app requires a new behavior, the sandbox should prohibit it or defer the request for the approval by a human user.

For example, if a new version of an app is released, a user can install and run it in the sandbox. As the sandbox detects previously unseen behavior (e.g. reading a file), it alerts the user. Then, the user assesses the situation, and if the user determines if the new behavior is desirable, its execution is permitted. Otherwise, the execution of the potentially dangerous behavior is stopped.

For these techniques to work effectively, it is necessary to sufficiently explore the app. If a normal behavior is not accessed during the exploration, it will be produce false alarms during the sandboxing phase. Existing techniques [252, 209, 398, 79] rely on test generation to explore the app. These studies suggest that test generation can be effective for exploring behaviors in individual Android applications and Linux containers.

Key to mining sandboxes with test case generation is the *test complement exclusion* [438] method. There is no guarantee that behaviors that have not been observed will not occur in future. Test complement exclusion turns this limitation into a guarantee by using the sandbox to allow only behaviors seen as the sandbox is mined. Therefore, if no malicious behavior was observed,

then no malicious behavior can execute.

Existing techniques differ in the context considered to determine if a given resource should be accessible. If the rule allowing a resource access is too coarse-grained, then it may fail to detect malicious behaviors, while if it is too granular, then it may stop benign behaviors with inconsequential differences from executing. In the work of Wan et al. [398], only the set of system calls are tracked and their contexts are ignored. In Jamrozik et al's work [209], the execution context is the GUI element that the user interacted with before an Android API call. In Le et al's work [252], the execution context is the sequence of other API calls before the given API call.

7.3 IoTBox

Our primary contribution is the proposal of IoTBox, a technique that mines a sandbox for a smart home. Key to our approach is to consider only causal information between events in a smart home. First, IoTBox connects events and actions to determine all possible paths leading to any action. If there is some unexplained difference between IoTBox's expectations and the actions in the real world, then it suggests that there is a behavior in the smart home that differs from the rules that IoTBox has learned. There are two phases to IoTBox, much like other techniques mining sandboxes [209, 398, 252, 79], the exploration phase and the sandboxing phase.

In the exploration phase, IoTBox thoroughly explores of the behaviors of the software system to determine the execution context of possible actions in the smart home. We consider an action's execution context as the set of execution paths causing the action's execution. IoTBox's refines the execution context of a given action, finding all possible causes of the action, by utilizing the Alloy Analyzer to identify all paths linked to the action, including non-trivial interactions across multiple apps and timed events. This creates two guarantees. Firstly, only events that are linked to the given action are identified, and secondly, all such events are found. In contrast, existing work on mining sandboxes rely on test case generation, which may fail to comprehensively explore the entire search space of behaviors (In Section 7.5, we discuss the tradeoffs of this choice). Abstracting the identified behaviors as rules, IoTBox uses these rules in the sandboxing phase to judge if there is any missing or disallowed action given recently observed events.

7.3.1 Exploration phase

To mine a sandbox, we use the Alloy Analyzer, a model checker, to thoroughly explore the behaviors of a bundle of apps encoded in the behavioral rule models, introduced by Alhanahnah et al. [53]. As described before in Section 7.2, these models are encoded in Alloy. The exploration phase can be viewed as answering the question *"What are all possible paths that lead to a given action?"*.

After we have produced formal models of all apps in the smart home, our first step is to identify all actions that the apps may execute. This is done by traversing all rules and picking out all actions that may be executed. Our next step is to find all execution paths to lead to any given action. In the example in Figure 7.1, one such path is ("Motion Detected" → "Home Mode" → "Unlock Door"), This entails finding out all events, *events*, that can trigger each action, a. First, we identify an event, *event*, that will trigger it (i.e., by simply using the triggers and conditions of the rules that the action is a command of). Next, we construct an Alloy assertion that checks that all events on a path leading to the action is either *event*, or is preceded by it. The following Linear Temporal Logic [163] fragment describes that any occurrence of a must be on an execution path triggered by *event*:

$$\neg a \; W \; event$$

event has a device capability, an attribute, and a value associated with the attribute. This is used to initialize *events*, which initially contains just *event*. For example using Figure 7.1, given a trigger (*location*, *location_mode*, *HOME*), which matches an event on a motion sensor, we initialize the set, *events*, to contain a single event, (*location*, *location_mode*, *HOME*). An example of the assertion in Alloy is shown in Figure 7.2. It asserts that for all rules that unlocks the door, there is no direct or transitive predecessor that is not on the chain of events triggered by the transition to *HOME* mode.

We run the Alloy Analyser to check the assertion. If the assertion fails, the Alloy Analyser will generate a counterexample of an event, different from *event*, that transitively triggers the action. This new event, *event*2 (one of "motion detected", "switch on", or "App Touch" in our example), is added to *events*, and we modify the assertion to look for counterexamples of paths which are triggered by events not in *events*. Only *event*2, which is only a single event that precedes the other events on the path, is added. Obtaining the possible paths from *event*2 is done afterwards. There may be multiple paths that are triggered by *event*2 that result in a.

```
assert {
  no r : IoTApp.rules, action : r.commands {
    action.attribute =lock
    action.value    =unlock
    (some predecessor : r.*(~connected),
    action' : predecessor.triggers
     {
      not {(
      {
          action'.attribute =location_mode
          action'.value =HOME

      }) or
      (some predecessor' : predecessor.*(~connected),
        action'' : predecessor'.triggers
         {
              predecessor ≠r
              action''.attribute =location_mode
              action''.value =HOME
         }
      )}
    })
  }
}
```

Figure 7.2: Example of an Alloy assertion. The Alloy Analyser produces a counterexample, a path that triggers the unlocking of a door without initially triggered by a transition to *HOME* Mode.

$$\neg a \, W \, (event \vee event2)$$

Again, we execute the Alloy Analyser to check this assertion. If it fails, we once again use the counterexample it finds to modify the assertion. This process continues until the Alloy Analyser fails to find a counterexample, and that all chains of events that lead to a have been accounted for. At the end of this process, this assertion doubles as an interpretable security policy. The policy declares all possible events that can lead to a, and checks that other events do not transitively trigger a. This process produces the following assertion with n different triggering events:

$$\neg a \, W \, (\vee_1^n event_i)$$

An advantage of this technique is that it identifies only events that have a causal relationship with a based on the behavioral rule graph. Only events that are a root cause of the execution of a are identified and included in *events*. This avoids the problem of spurious associations had we applied data mining techniques on the large number of events.

Next, we traverse the behavioral rule graph, beginning with the members of *events*. We find all paths that lead to a. A path is a series of events and does not contain loops. From all paths that start with a member of *events*, we identify all subpaths that end with a, and in turn, these subpaths are paths. This gives us the set of paths, *paths*.

$$execution\_context(a) = paths$$

We treat *paths* as the execution context of a. For an invocation of a, at least one path within the execution context must be satisfied; all conditions and their predicates along the path must be satisfied, and all triggers in this path must have been triggered. $satisfied(path)$ is an implementation detail that will be described in Section 7.3.2.

If an execution context of an action is satisfied, then an actuation of the action is expected. With this assumption, IoTBox looks for mismatches between the expected and actual actions in the smart home. Let us define two predicates: $expected(a)$ is true when the execution context of a has been satisfied. $actual(a)$ is true when the action a has been observed in the IoT environment. Based on these two predicates, the following gives a formal definition of IoTBox's assumption:

$$satisfied(execution\_context(a)) \leftrightarrow$$
$$\exists path(path \in execution\_context(a) \wedge satisfied(path))$$
$$expected(a) \leftrightarrow satisfied(execution\_context(a))$$

By the end of this phase, we have constructed for any given action, a, an execution context that comprises the set of all paths that lead to the triggering of a. The execution context allows for both the detection of new causes of an action and missing actions.

Given an action a that has taken place, IoTBox considers it as a disallowed action if its execution was not expected.

$$disallowed(a) \leftrightarrow \neg expected(a) \wedge actual(a)$$

Conversely, IoTBox considers an action to be missing if its execution was expected but is not observed in reality.

$$missing(a) \leftrightarrow expected(a) \wedge \neg actual(a)$$

7.3.2 Sandboxing phase

The objective of the sandboxing phase is to detect if there is a change in the behaviors of the smart home. This is done through monitoring executions at runtime. While the exploration phase was done through static analysis, it is insufficient to use static analysis to prevent malicious behaviors; malicious behaviors can be invoked through dynamic language features, such as call-by-reflection. Within IoTBox, the sandboxing phase answers the question: *"For all automated actions, are there changes in the paths that can trigger it?"*

If there is an unexpected action, then it implies that there is a new path that IoTBox is unaware of. If there is a missing action, then it implies that some paths has been removed.

When deployed, IoTBox communicates with the app before the execution of each action and whenever an event takes place. This requires the instrumentation of the smart apps to send events to IoTBox. We write a Groovy program transformer that modifies the Groovy smart apps at each event handler and at call sites of any action. The modified app calls out to a third-party server to either update IoTBox of new events or to request for permission to run an action. When an event handler runs, it logs the event to IoTBox. Before an action is invoked, the app waits for permission from IoTBox. Malicious behavior may be invoked through dynamic program features such as call-by-reflection, and we add guards to locations using Groovy's *GString* feature to perform dynamic method invocations. The action, resolved at runtime, waits for permission from IoTBox before execution, similar to statically invoked actions.

The traces of events and actions previously taken in the smart home prior to the action are used by IoTBox to make a decision to allow or reject an action. As events occur in a smart home, IoTBox updates its model of expected actions based on the execution contexts of the actions. With every new event that is reported, IoTBox determines if there is any missing event. IoTBox warns the user if an action is rejected or if there is a missing action.

We did not experience significantly increased latencies when we deployed our tool on the SmartThings Simulator. Our findings are similar with previous studies, which found that even after instrumentation, the latency for an action to execute is largely caused by communication between the IoT platform's server and the physical device [106].

IoTBox uses the most recent executed events to make decisions, much like DSM, the state-of-the-art technique to mine sandboxes [252]. Using these events, it makes a best-effort guess if each *path* in an action's execution context is satisfied. IoTBox is conservative about the triggers on the path,

requiring that they must all be present before allowing an action, but liberal in checking the conditions; an action is denied only if there is evidence that a condition is not satisfied. Concretely, we propose the following algorithm to determine if *satisfied*(*path*) holds for a given *path*.

In Algorithm 9, we iterate over the sequence of events in the traces, using `tracePointer` (initialized in Line 1, incremented in Line 16), and over the triggers on the path (Line 3). Each trigger has conditions associated with it, coming from the same rule (Line 4). Between the events that match subsequent triggers, if an event matches a condition on its device and attribute, but with a different value, then it causes the valuation of the condition to be false. If the event matches on the value, then it causes the valuation to be true. We track the conditions' valuation using `isConditionNegated` (initialized in Line 2), which maps a condition to true if an event caused the condition to have a negative valuation (set in either Line 10 or 12). Before the next trigger is matched, if a condition's valuation is false, then the path cannot be satisfied (Lines 18-20). `any(isConditionNegated, conditions)` returns true if any condition maps to `true`. A path is not satisfied if there is a missing trigger (Line 21) or if there is evidence that a required condition is false (Line 18). Otherwise, the path is satisfied (Line 25).

This algorithm is best-effort and may not always be accurate. A smart home is stateful and local information from the most recent traces may not be enough to make the right decisions; it is not always possible to determine the truth value of a condition. For example, a condition that the home is in *HOME* mode cannot always be determined from the most recent traces, as *HOME* mode may have been set hours or even days ago[1]. While it may be possible to simply store the all updates to state of the smart home, including state changes from multiple days ago, we surmise this poses a privacy risk if this monitoring service is ever compromised [439, 152]. As such, we present an approach that use a minimal amount of information from recent traces to make best-effort decisions.

7.4 Empirical Evaluation

We are interested in answering 2 research questions:

- **RQ1: How frequently do handcrafted security policies lead to false positives?**

  Existing approaches detect security issues from the **joint** behavior of multiples apps. We claim that their handcrafted policies may produce

[1]This is particularly true during a pandemic

Algorithm 9: Algorithm to determine if a given *path* is satisfied (i.e., *satisfied*(*path*)) based on the events in *trace*.

Input: A sequence of events, *trace*.
Input: A path, *path*.
Output: *satisfied*(*path*), either true or false

```
1  tracePtr = 0, event = null
2  isConditionNegated = {}
3  for trigger ← triggersOf(path) do
4      conds = conditionsAssociatedWith(trigger, path)
5      while event != trigger && tracePtr <trace.len do
6          event = trace[tracePtr]
7          for cond ← conds do
8              if cond.device = event.device && cond.attribute =
                event.attribute then
9                  if cond.value! = event.value then
10                     isConditionNegated[cond] = true
11                 else
12                     isConditionNegated[cond] = false
13                 end
14             end
15         end
16         tracePtr++
17     end
18     if any(isConditionNegated, conds) then
19         return false
20     end
21     if tracePtr == trace.len && event != trigger then
22         return false
23     end
24 end
25 return true
```

many false positives. In this research question, we investigate if **individual** benign apps violate the security policies. As users are likely to understand and reason about individual apps, we assume that, if installed, their behaviors are intentionally introduced into a smart home.

- **RQ2: How effective is IoTBox?**

  In this research question, we investigate the effectiveness of IoTBox against DSM, previously proposed for Android apps, and a simple strawman sandbox.

7.4.1 RQ1: How frequently do handcrafted security policies lead to false positives?

This research question investigates the prevalence of false positives from the use of handcrafted security policies from prior work. We test a random sample of 500 apps from public repositories containing existing apps [21, 81]. For each app, we check the model produced by IoTCOM against the 36 policies used in IoTCOM [53]. These policies are similar to the policies used in other studies [106, 301, 105]. As we only test individual apps, all violations are not caused by interactions between apps, and are likely to be from behaviors that were intended if a user installed the app.

Findings

We find that out of the 500 apps, 326 of them (65%) violate at least one policy, with a total of 572 violations. On average, there is 1 violation per app. If deployed in a real setting, there is a high chance that a violation of the security policies is a false alarm. While the security policies were designed for and can help to catch dangerous behaviors, they cannot be used out of the box.

One cause of the numerous false alarms is that the policies are too broadly specified, causing IoTCOM to detect violations even for legitimate uses. For each app, we describe the policy violated (based on the policies used in IoTCOM [53]), highlight the reason for the violation, and if the violation of the policy is intended. False alarms can come from either surprising uses of devices or from IoTCOM's overapproximation of execution paths that will not be taken in reality. While we discuss only 4 violations (out of 326) in Table 7.1, we expect that, in the wild, there are many situations where IoT devices are used in unexpected ways. Many of these uses will violate security policies that do not anticipate surprising uses. This problem was also observed by Celik et al. [105]. For example, they reported anecdotes

Table 7.1: Behaviors of individual apps that violate the handcrafted security policies. The parenthesis indicate the policy identifier in the IoTCOM paper. Policies prefixed with P indicates a safety property, while T indicates a general coordination threat.

Policy Violated	**Reason for violation**
The heater should not be turned off when the temperature is low (P.7)	Intended; the app (use_outdoor_temp_to _turn_on_off_a_switch) turns off a switch (which may be the heater) based on the temperature outside, and not indoors.
Lights are not switched off if someone is at home (P.9)	Intended; the app (IlluminatedResponseto--UnexpectedVisitors [20]) toggles the lights on and off for illuminating someone snooping about the house (e.g. a burglar), switching the lights off if the burglar has left.
No rules with conflicting actions but same triggers/conditions (T7)	Intended; the app (LockDoorafter--Xminutes [26]) sets a variable for internal bookkeeping (to track that the door was automatically opened), but sets it to a different value (that it has been closed) after some time.
Location mode should be set to HOME when someone is at home (P.15)	Intended; while the user is home, the app (MotionModeChange [27]) may set the mode to NIGHT.

of users using flood sensors that produce alerts when water levels are low (contrary to its expected use in detecting floods) for reminders to water their plants.

In Software Engineering literature, researchers have pointed out that users rarely use tools that produce many false positives, inhibiting its usage [153, 239, 215]. Users would need to apply significant effort to understand and tweak each security policy to their smart home. This finding motivates more research into automating this process.

7.4.2 RQ2: How effective is IoTBox?

This research question investigates the effectiveness of IoTBox. We compare the number of bundles of apps for which IoTBox successfully identifies malicious changes in behavior. In all cases, we compare IoTBox against two baseline techniques, a strawman sandbox and DSM [252]. All techniques take traces of events as input. First, for each bundle, we produced events based on the formal models to simulate the smart home, triggering an average of 1208 actions. Next, to confirm our findings, we generate tests on the SmartThings Simulator and collect traces from the apps' executions, triggering 555 actions on average.

Experimental Setting

We use the flawed apps that were studied previously. We use the bundles of apps from IoTMAL, used in the evaluation of previous studies (Bundle 1-6) [105, 53]. Furthermore, we proposed new bundles of apps (Bundles 7-16), constructed with individual, flawed apps proposed in prior studies [105] and combining them with other benign apps. We study the same apps used in the evaluation of IoTCOM [53]. Bundle 17 is the example from Figure 7.1, and we constructed a malicious variant by removing a transition to *HOME* mode.

In each bundle, we locate the app with malicious behavior and create a variant of the bundle by removing the unsafe logic in this app. To evaluate the ability of IoTBox to detect missing behavior, we further constructed variants of several bundles by removing a piece of behavior required for it to function correctly. We pass the benign bundles of apps as input to IoTBox, which explores their behaviors and constructs rules for the execution of each action. Thus, in this work, a benign bundle of apps is the modified bundle of apps without malicious behavior. A malicious bundle of apps is either 1) the original bundle containing an app with a malicious behavior, or 2) a bundle of apps modified from the a benign bundle to remove a necessary

behavior. A total of 17 benign bundles of apps were explored, and 20 variants of these bundles of apps with either additional malicious or missing necessary behavior were constructed.

Next, we statically produce execution traces of the bundles of apps. The executions of the benign bundles of apps are used for learning the sandbox for both the strawman sandbox and DSM. The trace are produced from the models to allow for a fair comparison of existing techniques and IoTBox, as only the parts of the IoT app captured by the formal models are used to produce traces. To produce events from the formal models, we identify two types of event for each trigger and predicate; one event will set the predicate to true, and another will falsify the predicate. We track the state of the environment as a mapping of the attributes of every device located in the environment to one value. We model time using a counter, which increases for every event. At each time step, a random event will be selected for execution. As an event is executed, the state of the environment is modified. Every time an event is executed, we iterate over all the rules in the formal model and determine for each rule, if its trigger matches the event, and if its conditions have been met by the modified state. If so, then an event that matches the command will be executed, either immediately or after some time.

The execution traces from the malicious bundles of apps are input to all techniques. If a technique rejects more than 1% of actions, then we consider that it is able to detect the malicious behavior in the bundle.

In simulating the apps' executions, we constructed a challenging experimental setup. We produced a large number of events while simulating the possibility of race conditions between the apps. The triggering of apps may be delayed to reflect real-world network conditions. Events from the execution of an app may be interleaved between executions of other apps. Thus, the threshold of 1% permits some degree of false positives caused by these challenging conditions. Without this threshold, DSM will find two times more false positives. In practice, we expect that these challenging conditions will not frequently occur. Also, we believe that users will find reasonably rare false alarms to be acceptable. We manually inspected the statically generated traces to verify if the traces are plausible based on the source code. Of 50 randomly sampled traces, none were infeasible.

The strawman sandbox detects if there is any action executed by the smart home that was not seen during the exploration phase. Any previously unseen action is rejected. DSM [252] takes the sequence of events and actions that occurs before an action as input. DSM uses an automaton model, inferred through a Recurrent Neural Network [252]. If the events are rejected by this model, the action is prohibited.

We use IoTBox as described in Section 7.3. Rules of the sandbox are

first learned through the exploration of the modified bundles of apps without any malicious behavior. Next, we test the sandbox against both the original bundle of apps containing the malicious behavior, and the modified bundles without malicious behavior. We count the number of malicious bundles of apps that were correctly identified (**T**rue **P**ositives), the number of benign bundle of apps identified as malicious (**F**alse **P**ositives), and the number of malicious bundle of apps identified as benign (**F**alse **N**egatives).

We evaluated the effectiveness of the tools using Precision, Recall, and F1. Precision and Recall are computed as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{FP+FP}} \text{ Recall} = \frac{\text{TP}}{\text{FP+FN}}$$

A precision of 100% indicates that the absence of false positives, while a recall of 100% indicates that every malicious behavior was caught. Finally, F1 is the harmonic mean of precision and recall. These metrics are widely used in the literature of both mining sandboxes [252, 79], as well as other related domains such as the detection of malware [342, 69].

Experimental Results from Statically Produced Traces

Table 7.2 summarizes our results on the static traces, for 16 out of 20 bundles, IoTBox detected a change in behavior that should be disallowed. In contrast, DSM and the strawman detected only 13 and 8 bundles with changed behavior.

We evaluated the techniques on false alarms and found that DSM produced false positives on 3 bundles. Both the strawman and IoTBox do not produce false positives.

As we produced over a thousand traces, we rule out a lack of training data as a reason for DSM's relative ineffectiveness. We hypothesize that its poor performance is caused by the tight coupling of its decisions to the local context, requiring that all necessary information to make the right decision appear in the most recent traces. In fact, the relevant events to make the right decision can occur far apart from one another. While Le et al. [252] suggests that DSM's use of Recurrent Neural Networks may help in capturing long-term dependencies between events, still, an IoT system is stateful and the relevant state changes may have occurred before the collection of the most recent traces. Ultimately, DSM is fooled by spurious patterns in the most recent traces. On the other hand, IoTBox encodes domain knowledge of IoT apps, only loosely enforcing conditions to check if they are satisfied.

Only IoTBox can detect changes in behavior that result in missing actions (indicated with (M) in Table 7.2). Both DSM and the strawman do not detect

Table 7.2: Bundles with malicious behavior that were detected. T indicates that the malicious behavior was detected, F otherwise. FP indicates that the benign bundle was classified as malicious. (M) indicates bundles where the malicious change is a removal of existing behavior.

Bundle	**Strawman**	**DSM**	**IoTBox**
1. Bundle 1	F	F	**T**
2. Bundle 2	T	T	T
3. Bundle 3	T	T, FP	T
4. Bundle 4	T	T, FP	T
5. Bundle 5	T	T	T
6. Bundle 6	T	T	T
7. MaliciousBatteryMonitor [151, 212]	T	T	**T**
7 (M)	F	F	F
8. ID1BrightenMyPath	F	T	**T**
9. ID2SecuritySystem	F	T	**T**
10. ID3SmokeAlarm	F	T	F
11. ID4PowerAllowance	F	F,FP	F
12. ID5FakeAlarm	F	T	**T**
12 (M)	F	F	**T**
13. ID6TurnOnSwitchNotHome	T	T	T
13 (M)	F	F	**T**
14. ID7ConflictTimeandPresence	F	**T**	F
15. ID8LocationSubscribeFailure (M)	F	F	**T**
16. ID9DisableVacationMode	T	T	T
17. Figure 7.1 (M)	F	F	T
# True Positives	8	13	**16**
# False Positives	**0**	3	**0**
Recall	40%	70%	**80%**
Precision	**100%**	75%	**100%**
F1	57%	72%	**88%**

missing actions. On the other hand, IoTBox detects 4 out of 5 cases with missing actions.

Both the strawman sandbox and IoTBox have 100% precision. Overall, IoTBox performs best, in terms of Recall and F1 score. Even if we omit the cases with missing actions, then IoTBox has a Recall of 60% and an F1 of 75%. Overall, we believe that there is sufficient evidence to suggest that IoTBox is more effective than existing sandbox mining techniques.

Experimental Results on Traces from Test Case Generation

Next, we investigate the effectiveness of IoTBox on traces produced from test case generation. This approach uses the SmartThings simulator [32]. We instrument the apps, adding log statements to them.This allows us to collect information about the state of the smart home without modification to the platform that the apps runs on. Our objective is to elicit enough traces to evaluate IoTBox to determine if our findings are likely to hold in practice.

Our test generation strategy identifies relevant event types by detecting which devices are present in the smart home. After that, it randomly generate events of these types through the simulated devices in the SmartThings IDE. As some apps are time-sensitive, we transform the Groovy programs, replacing time-related functions with mocks. Our test generator simulates the passing of time. Instead of using the current real-world time, the app fetches the mocked time from our server. Time monotonically increases with the number of executed events; every time an event is executed, the test generator advances time by a random number of minutes. For functions scheduled to run after some time (i.e., using *runIn*), we set its waiting duration to between 1 to 3 minutes, giving them a high chance of executing within the experimentation duration.

Execution traces are collected from the logs. Each trace is a sequence of events and actions executed by the apps. For each bundle, we generate tests for over an hour. The same execution contexts are used, unmodified, from the previous experiments.

Due to the limitations of the SmartThings simulator, we restrict our analysis to only a few bundles of apps. The simulator cannot simulate physical channels of interactions (e.g. an app switching on a lamp may trigger an app reading from a light sensor), and does not support simulating every device type, a known limitation of test generation on the SmartThings simulator [212]. Therefore, we cannot create an accurate simulation of several bundles (Bundles 4-7, 12-14) in our evaluation dataset. We also omit Bundles where IoTBox was ineffective on the static traces (Bundles 10, 11).

Our results are shown in Table 7.3. IoTBox detects the malicious changes

Table 7.3: Effectiveness of IoTBox on traces from test generation. Not all bundles can run on the SmartThings Simulator as they use devices unsupported by the Simulator.

Bundle	Malicious Behavior Detected
1. Bundle 1	T
2. Bundle 2	T
3. Bundle 3	T
8. ID1BrightenMyPath	T
9. ID2SecuritySystem	T
16. ID9DisableVacationMode	T
17. Figure 7.1 (M)	T

in behaviors on all seven bundles of apps. As before, there are no false positives. Overall, these results agrees with the evaluation results using the statically produced traces in Table 7.2, and supports our findings that IoTBox is effective in mining sandboxes for a smart home.

7.5 Discussion

7.5.1 Risk of encoding malicious behavior in the sandbox

In any technique mining sandboxes, there is a risk that the mined rules encode malicious behavior that were already present in the smart home [209, 398]. If so, the sandboxing phase does not prevent its execution as it is an expected behavior. This is one source of false negatives [209], as the malicious behavior would be considered benign. On the other hand, this implies that *the malicious behavior is already explicitly described in the security policy mined during the exploration phase* and can be checked by a human user.

We analyse the quality of the rules produced by IoTBox. The rules mined by IoTBox are simple and precise, expressed as an Alloy assertion. Furthermore, IoTBox can visualize the paths from any unexpected trigger to a given action. This lends the rules to inspection by human users of an IoT system.

We investigate what a malicious behavior encoded in the rules may look like. We use the running example involving the unlocking of doors introduced in Figure 7.1, and now, we point out it was crafted with a flaw in mind. While it correctly unlocks the door when the smart home transits to *HOME* mode, in fact, it unlocks the door in any mode change, including mode changes

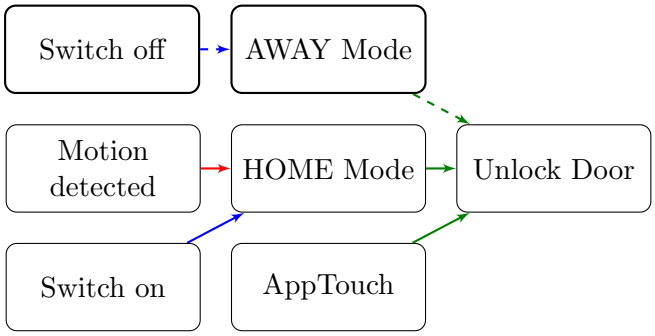


Figure 7.3: An undesired path (from the interaction of App2 and App3 in dotted lines) to unlock the door is already in the smart home before the exploration phase. Undesired paths may not be easily noticed. IoTBox helps detect such paths through its interpretable rules.

away from *HOME* (e.g. to *AWAY*) [41]. This behavior is unexpected to our hypothetical user who did not carefully inspect the app.

In Figure 7.3, we show a more complete version of the graph from Figure 7.1 (in Section 7.2) with this surprising behavior. An excerpt of the security policy mined by IoTBox is shown in Figure 7.4. Both the Alloy assertion and accompanying code comments, generated by IoTBox, makes it simple to determine the events that led to the smart home automatically unlocking the app. Also, IoTBox can present all paths that lead to a given action from a given event, providing a visualization identical to the graph shown in Figure 7.3. In this case, our hypothetical user may inspect the security policy, and will be surprised that toggling a particular switch will always trigger the unlocking of the door. Thus, the user will be able to identify existing undesired behavior with the help of IoTBox.

IoTBox allows malicious behaviors to be executed provided that they were encoded in the execution context. While this seems to be a limitation, this is, in fact, a strength of sandbox mining. Writers of malicious apps are forced to "disclose-or-die" [209], as any malicious behavior must be made explicitly detectable for analysis. Otherwise, the sandbox will prevent its execution. If so, a user of an IoT environment can detect the malicious behavior through inspection of the assertion and the visualization of paths leading to an action, both produced by IoTBox, to locate the undesired behavior.

```
assert {
  // if the lock is automatically unlocked,
  // it is caused by ...
  no r : IoTApp.rules, action : r.commands {
    action.attribute =lock
    action.value    =unlock
    (some predecessor : r.*(~connected),
       action' : predecessor.triggers {
      not {
        ...// omitted code
        // the switch turning on OR off
        action'.attribute =switch
        (action'.value =switch_on
           or
        action'.value =switch_off)
      }
    )
  }
}
```

Figure 7.4: Excerpt of the Alloy assertion revealing possible triggers.

7.5.2 Limitations and Tradeoffs

The primary difference between IoTBox and existing techniques that mine sandboxes is its use of models produced from static analysis, instead of using test generation. Fundamentally, the previous techniques use *test complement exclusion* [438]. This relies on the incompleteness of test case generation; test generation cannot explore all possible behaviors. If only normal behaviors were observed from testing during the exploration phase, then it guarantees that the sandbox allows only normal behaviors to run. IoTBox relies on a variant of this guarantee; only behaviors captured in the formal model can be executed. Conversely, *potentially dangerous behaviors that were not analysed are disallowed from running.*

We do not learn from normal executions, but include all behaviors that were abstracted into an app's model. By doing so, we gain the advantage of fewer false alarms as more behaviors are covered compared to test generation. On the other hand, static analysis usually overapproximates possible behaviors, introducing the possibility of including malicious behavior in a model. This is mitigated in IoTBox through non-opaque rules that can be interpreted by human users.

7.5.3 Threats to Validity

We mitigate threats to internal validity by relying on the models and tool used by other researchers [53], with a formalism of an IoT app (abstracted into triggers, conditions, and commands) that is similar to that of other

studies [212, 105, 106, 301]. In some cases, we find that the Alloy models produced by IoTCOM are not directly usable (e.g. as they do not compile), so we modified them. Our models are publicly available [22].

To minimize threats to construct validity, we have used the evaluation metrics from previous studies [252, 79]. We evaluated the risk of false negatives, similar to Jamrozik et al. [209]. Moreover, we studied the same flawed IoT apps from a previous study [53], and many of these apps have also been studied in other works [212, 105, 106, 104]. We studied 20 bundles of apps, similar to prior studies. Jamrozik et al. [209] and Le et al. [249] studied 13 and 25 Android apps, and Wan et al. [398] studied 8 Linux containers.

A threat to external validity is that our experiments focuses only on the SmartThings and IFTTT platforms. Other smart home platforms include Apple's HomeKit [10], Google Home [14], Zapier [45], and Home Assistant [17]. However, as SmartThings support more devices than competing platforms [30] and IFTTT has over 11 milion users, we expect our findings to generalize.

7.6 Related Work

Mining sandboxes. Compared to existing studies on mining sandboxes [209, 252, 79, 398], IoTBox does not use test case generation due to the difficulties of generating comprehensive test cases for an entire IoT system. Instead, IoTBox explores a formal model of the smart home, identifying the execution context of an action through the counterexamples found by the Alloy Analyser [207]. This has the advantage of identifying only events that have a causal relationship with a given action through analysis of the behavioral rule graph.

Related to mining sandboxes, Acar et al. [51] suggests that the automatic generation of security policies may help to address the permission comprehension problem on Android. Provos [333] proposed learning policies for system calls in UNIX systems. IoTBox is the first approach to account for context that span multiple apps in a smart home.

Threats in an IoT system. Researchers have studied threats at the application-level on IoT platforms [151, 407, 105, 106, 406, 53, 301, 116, 136, 386, 373]. ContexIOT [212] considers the context of an app in isolation, missing out threats that span multiple apps. Some studies have shown the prevalence of incorrect behavior in IoT applications [386, 53, 373, 81]. Compared to IoTMon [136], Soteria [105], IoTGuard [106], HOMEGUARD [116], IoTSan [301], IoTCOM [53], iRuler [406], our work shares the goal of identifying malicious behavior from the interaction of apps, but different from

these studies, does not require predefined policies of potentially dangerous interactions and has the objective of inferring rules that will help to detect behavioral changes. Moreover, these studies do not detect missing actions caused by malicious behavior.

IoTBox and ProvThings [407] share similarities as both studies trace the origins of events, traversing graphs of apps connected by how they trigger and influence the execution of one another. However, they have different goals. IoTBox mines rules for a sandbox to detect behavioral changes while ProvThings collects information for debugging.

Other aspects of IoT systems have been studied [56, 152, 65, 453, 110]. Researchers studied the impact of platform compromise [152], and suggested the need to minimize potential damage from an adversary that has compromised an IoT platform. Various security aspects, including the misuse of IoT devices for botnets [29, 65] and firmware security [453, 110], have been studied but differs from our work in their goals.

Other researchers have shown that developers face difficulties trying to interpret behaviors in IoT apps [93, 446, 198, 439]. IoTBox may be helpful as a debugging aid, as it can reveal unexpected changes in behavior given a behavioral change. Researchers have suggested the need to simplify the configuration of privacy policies [440]. There are many concerns, including the invasion of privacy of other users in the smart home [439]. Our work may find application in these areas, in helping to interpret the automation in a smart home. Indeed, other IoT researchers have pointed out the need for tools to allow users to identify the effects of enabling a new app, to identify unforeseen consequences [184].

7.7 Summary

In this work, we show that many handcrafted security policies for smart home produce numerous false alarms and suggest the need to automate the specialization of these policies for a smart home. We propose IoTBox to mine sandboxes of IoT systems, which detects change in behaviors in IoT systems. A sandbox with rules mined from an existing smart home will produce few false positives from unexpected usages of apps and devices. IoTBox produces rules that can be inspected and visualized by a human user. We develop IoTBox, which identifies the complete execution context of any action. This produces rules that we can enforce during deployment, detecting both disallowed and missing actions.

IoTBox can comprehensive explore the behaviors of a smart home and can precisely identify rules to enforce. We empirically evaluated IoTBox on

app bundles containing flawed behavior that were studied in prior work. We find that it can effectively detect changes that introduce malicious behavior, without producing false positives. A replication package is available [22].

Chapter 8

(System of Interacting Components) Test Mimicry to Assess the Exploitability of Library Vulnerabilities

8.1 Overview

Software engineering projects often depend on open-source software libraries [314, 244, 409]. As vulnerabilities in a project's library dependencies may be exploited by attackers of the project, developers have to understand their project's dependencies and update them whenever library vulnerabilities are discovered. For example, the recent Log4Shell vulnerability, which affected millions of devices, required client developers to update their applications to use the latest version of the log4j library quickly [44, 15]. Moreover, as the clients of a library include other libraries, a vulnerability in one library would transitively propagate throughout the ecosystem of libraries and their clients.

After library vulnerabilities have been fixed and publicly disclosed, client developers are advised to update their dependencies to use the new, non-vulnerable versions of libraries. However, studies have shown that client developers are reluctant to update their dependencies. Many alerts regarding vulnerable dependencies are false alarms [314], and developers may be wary of breaking changes from library updates. This leaves client projects open to exploitation of vulnerabilities in library dependencies [244, 130, 288, 409].

To address this problem, there have been proposed techniques that assess the *reachability* of the vulnerable code (e.g. function) from the client

project, allowing developers and security researchers to better assess a vulnerability's exploitability from the client project. Existing techniques use call graph analysis to determine if the vulnerable code is called from the client project [328, 156, 203]. As control-flow within functions is not considered, existing techniques produce false alarms [156]. Fundamentally, these tools are limited since they check only if a vulnerable function may be *called*, but do not determine if the client projects are able to construct the inputs that trigger the vulnerability [156, 328, 203].

Recently, Iannone et al. [203] proposed SIEGE, a tool that automatically generates test cases demonstrating the exploitability of library vulnerabilities for client projects [203]. Given a description of the vulnerability, which is manually determined from a vulnerability-fixing commit, SIEGE targets the coverage of a single line of library code indicated. SIEGE generates test cases of the client projects that transitively execute the line of code, thus, providing evidence that the library vulnerability can be reached from the client project. While SIEGE could confirm the exploitability of some vulnerabilities, it is limited by its inability to overcome the intrinsic complexity of exploiting vulnerabilities; specific domain knowledge is required for triggering many vulnerabilities [203].

Recreating the triggering conditions of a vulnerability may be challenging because of the domain knowledge required. Take CVE-2019-12402 [13] in Apache Tika as an example: an attacker can trigger a denial-of-service attack by providing a zip file quine[1], a specially-crafted zip file that is unzipped to produce itself as output. It is extremely hard to build a zip file quine with random mutations (as in fuzzing) even with some guidance.

To overcome the challenging requirement of extensive domain knowledge, we propose to leverage test cases of the vulnerable, open-source libraries, particularly the test cases that accompany the vulnerability fixes. Specifically, we propose a new framework, which we term *Test Mimicry*, depicted in Figure 8.1. Expert domain knowledge and the conditions (e.g., specific inputs, program state) of triggering a vulnerability are captured in the test cases written by the domain experts. Rather than blindly generating test cases, we generate test cases of the client project that invokes the vulnerable method with the same arguments as the library's vulnerability-witnessing test cases. Rather than designing oracles to detect if a vulnerability has been triggered, we detect the replication of the program state reached in the vulnerability-witnessing test case.

Concretely, our framework uses the vulnerability-witnessing test case from the library's code, denoted as $\mathcal{L}^{\mathcal{T}}$, to generate a test case, denoted

[1]https://www.cvedetails.com/cve/CVE-2019-10094/

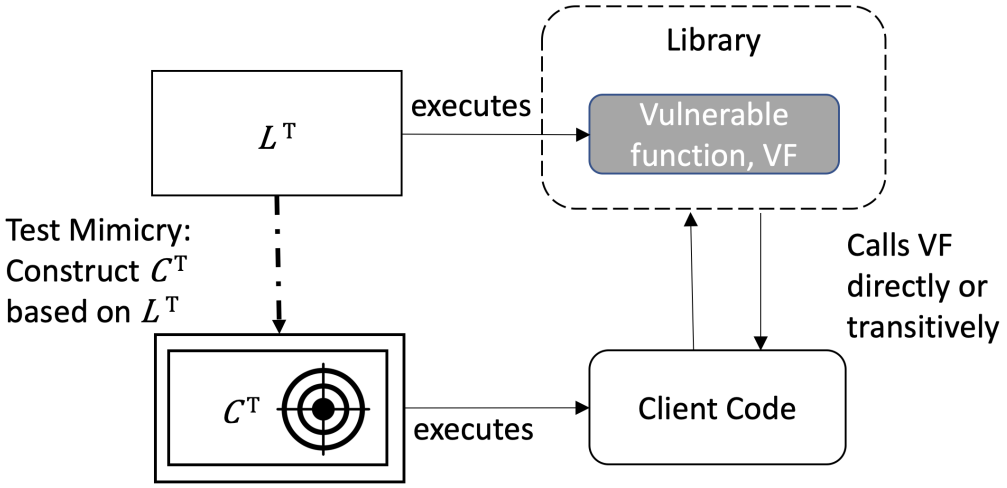


Figure 8.1: Test mimicry: Constructing a test case, $\mathcal{C}^{\mathcal{T}}$, for the library's client that demonstrates the same library vulnerability as witnessed by a library's test case, $\mathcal{L}^{\mathcal{T}}$.

as $\mathcal{C}^{\mathcal{T}}$, for code in a client project that *mimics* $\mathcal{L}^{\mathcal{T}}$. $\mathcal{C}^{\mathcal{T}}$ should test the client program such that the library exhibits the same behavior when it was tested with $\mathcal{L}^{\mathcal{T}}$. If the same program state reached by $\mathcal{L}^{\mathcal{T}}$ can be reproduced, then the vulnerability can be triggered from the client program. If so, then we have evidence that the vulnerability is exploitable from the client project.

To this end, we implement a tool, TRANSFER. TRANSFER targets client projects of open-source Java libraries. Given a library function, *VF*, associated with a known vulnerability, TRANSFER executes the library test case, $\mathcal{L}^{\mathcal{T}}$, that demonstrates how the vulnerability can be triggered. After identifying the program state relevant to the vulnerability, TRANSFER extracts the *triggering conditions*, σ, which are satisfied by reaching the same program state from a generated test case. TRANSFER uses an evolutionary algorithm to generate test cases, directing it to transitively invoke *VF* through the client program and favors test cases closer to satisfying σ. Finally, TRANSFER outputs a test case if it is sufficiently close to satisfying σ.

We evaluate TRANSFER by analyzing 22 real library vulnerabilities from a wide range of domains (e.g. JSON parsing, file compression) and types of vulnerabilities (e.g. XXE injections, unhandled exceptions). Analyzing 64 real client programs obtained from GitHub, we find that a library vulnerability can be exploited from 42 of them. While Siege produces exploits for 5 client programs, TRANSFER generates exploits for 23 client

programs.

8.2 Background and Motivation

Our work builds on prior studies on software composition analysis and search-based test generation. In this section, we describe them.

8.2.1 Software Composition Analysis

Software composition analysis is a domain related to the identification and replacement of vulnerable dependencies of software projects [156, 328]. Many solutions enumerate through a project's dependencies to detect potentially vulnerable libraries (i.e., looking up the versions of the project's dependencies against a database of known vulnerable library versions). From the source code of the project, a call graph is constructed to determine if a vulnerable library function is reachable. These analyses produce false positives as static call graphs may contain calls that do not occur at runtime [174, 348]. Hence, existing approaches [156, 328] complement the static analysis with dynamic analysis by running the client projects' test cases to construct call graphs, which reduces the number of false positives. These techniques are limited by the test coverage of the client projects, which may be low [236, 235]. Fundamentally, call graph-based approaches are limited as they do not consider control-flow or check that the inputs for triggering the vulnerability can be passed from client programs [156, 328, 203].

Siege [203] is a search-based test generator that produces exploits on client programs that executes vulnerable library code. These exploits are in the form of test cases that reveal how the library vulnerabilities can be exploited from the client programs. Exploits produced by Siege and our tool, TRANSFER, act as evidence of the exploitability of the library vulnerability from the client projects.

8.2.2 Search-based test generation

Search-based techniques have been proposed for generating test cases to satisfy a specified search criterion. EvoSuite generates test cases that achieve high coverage for Java programs [158]. Siege [203] assesses the exploitability of library vulnerabilities from client programs generating a test case for the client program, guided by the criterion of executing a vulnerable line of library code given as an input vulnerability description. Algorithm 10 shows a simplified version of a genetic algorithm for generating test cases. Starting

Algorithm 10: Simplified version of a genetic algorithm for generating test cases

Inputs: **1.** *goals*, the search goals
2. M, the population size
3. *search_budget*, amount of time to run
Output: *tests*, test cases

```
1  P_0 = construct_random_population(M)
2  k = 0
3  fitness = compute_fitness(pop, goals)
4  best_fitness = max(fitness)
5  while best_fitness < 1 and time spent <search_budget do
6  |   k = k + 1
7  |   // offsprings through mutations and crossover
8  |   offsprings = generate_offspring(P_{k-1})
9  |   // evaluate fitness and pick top M tests
10 |   P_k = P_{k-1} ∪ offsprings
11 |   fitness = compute_fitness(P_k, goals)
12 |   P_k = select_top_M(P_k, fitness)
13 end
14 return P_k
15
```

with a population of randomly generated test cases, a fitness value for each test case is computed with respect to the search goals (e.g., total code coverage for EvoSuite, how close the test is to covering the vulnerable line of code for Siege). While the search budget has not been exhausted and the optimal fitness value has not been reached, the genetic algorithm produces a new generation of test cases through mutations and crossovers on the previous generation of the test cases. The top M test cases are selected based on their fitness values to populate the next generation. As such, test cases with poor fitness are removed from the population.

A challenge faced by search-based test generators is the reproduction of complex behaviors. It is difficult for unguided, random test case generation to produce test cases that invoke complex behaviors. To generate test cases that invoke more complex behaviors, many techniques have been proposed. One such method is seeding [346], which uses existing knowledge about the program to help solve the search process. For example, string and numerical literals within the program are extracted into a constants pool. Seeding increases the likelihood of the test generator using these values, allowing it to

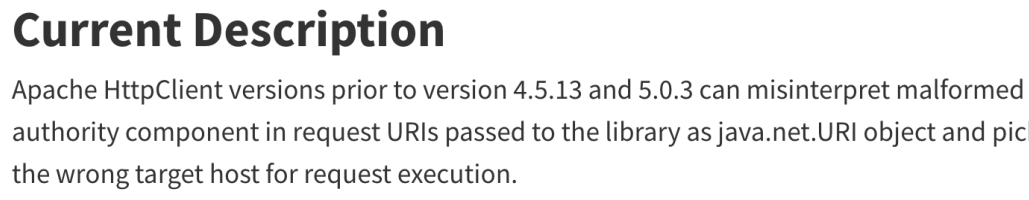


Figure 8.2: The information of a vulnerability given in NVD. The high-level description of the vulnerability makes assessing its possible impact difficult.

pass difficult conditional checks. A related technique is *test carving* [145, 346]. Test carving was proposed to convert larger system tests to a set of smaller unit tests for the same project. The technique extracts parts of the program state reached in the system tests as they are executed, capturing potentially reusable objects that can be recreated when constructing new test cases. The object states that comprise the program state may be difficult to recreate, and carving allows the construction of new test cases that starts from the same program state.

In this work, we build TRANSFER using the infrastructure and tooling provided by EvoSuite. Due to its maturity [55, 134], TRANSFER uses EvoSuite's implementation of the genetic algorithm, including the crossover and mutation operators. Unlike EvoSuite, TRANSFER seeks to reproduce, from client programs, the behavior demonstrated by the vulnerability-witnessing test case. TRANSFER's search criteria are dynamically determined from execution of the vulnerability-witnessing test case. TRANSFER targets the same vulnerable library function executed by the test case, producing new test cases that satisfies the triggering conditions extracted from its carved program state.

8.2.3 Motivating Example

From an alert about a new vulnerability in a library, e.g. CVE-2020-13956, a developer of a project (which may itself be another library) using the vulnerable library (e.g. Apache HttpComponents) is unsure if the vulnerability can be exploited. Figure 8.2 shows the vulnerability's high-level description, indicating that the vulnerable behavior occurs when a "misinterpreted authority component in request URIs" is input to the library. Without better understanding the vulnerability's exploitability and knowing that many alerts about vulnerable dependencies are false alarms [315], the developer may not prioritize the library update, leaving the library vulnerability in the project open to exploitation.

On the other hand, **with the test case generated by TRANSFER**,

```
Config config = new Config();
BasicClassicHttpRequest httpRequest =
  new BasicClassicHttpRequest("/", null,
  "http://blah@goggle.com:80@google.com/");
HttpClient httpClient = new HttpClient(config);
try {
  httpClient.execute(httpRequest);
   fail ("Expecting IOException");
} catch(IOException e) {
  verifyException("CloseableHttpClient", e);
}
```

Figure 8.3: Snippet of a test case generated by TRANSFER, with changes made for conciseness. The test case demonstrates how the client program can transitively invoke *VF* with inputs that triggers the vulnerability.

the developer has evidence of the exploitability of the vulnerability from the client project. As seen in Figure 8.3, the test case shows the client project class (`HttpClient`) and function (`execute`) that transitively calls the vulnerable function, how the function from the client program may be invoked to trigger the vulnerability (i.e. the construction of multiple classes from the client project and a concrete example of a malformed URL triggering the vulnerability, *http://blah@goggle.com:80@google.com*). While SIEGE struggles to construct a malformed URL, TRANSFER uses the example of the malformed URL from the library's test case. Providing evidence of the exploitability of a vulnerable may motivate developers to update their library dependencies.

Given the growing prevalence of library vulnerabilities, their widespread impact, and the importance of detecting them, reducing the difficulty of generating exploits for even a proportion of vulnerabilities would already be helpful. Moreover, writing test cases for bug fixes is considered a good software development practice [83]. Still, we assess the feasibility of using test cases from libraries. We looked for vulnerabilities reported in vulnerability databases [38, 39]. From the entries between March 2017 and March 2021, we sampled 780 entries containing a reference to a GitHub commit fixing the vulnerability. From the 780 entries, we identified 233 vulnerabilities from libraries. Of the 233 vulnerabilities, 121 (over 51%) of the commits include a test case, showing that a majority of vulnerabilities are fixed with test cases. Furthermore, our analysis underestimates the true number of vulnerabilities that can be revealed through a test case, as test cases may be introduced

through a different commit from the vulnerability fix. This indicates that test mimicry is likely to work for a large number of vulnerabilities.

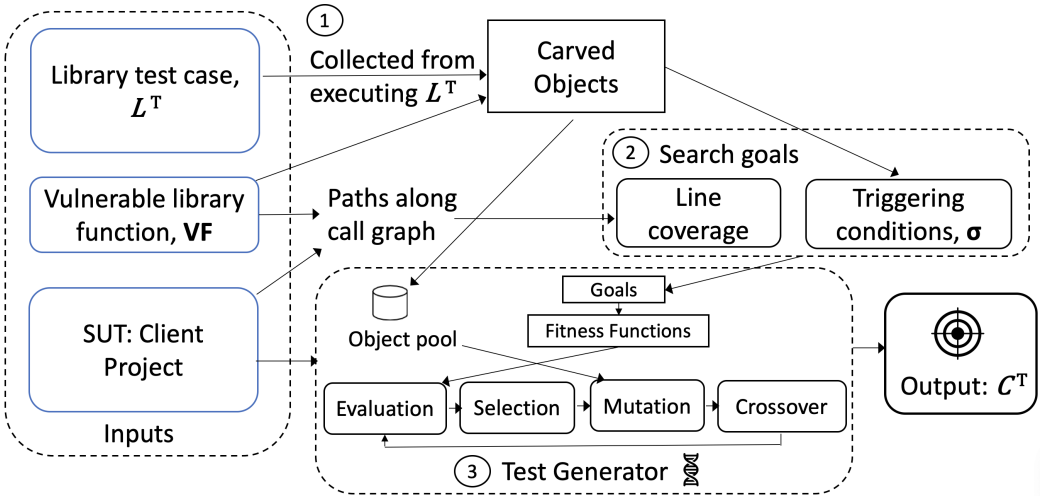


Figure 8.4: Overview of TRANSFER. From the vulnerability-witnessing test case from the library, TRANSFER produces an exploit for the client project – the Software Under Test (SUT).

8.3 Test Mimicry

In this section, we describe the details of our tool, TRANSFER.

8.3.1 Objectives and Problem Formulation

From a known vulnerability in a library and a function associated with the vulnerability, the *vulnerable function*, VF, a client program is a program from a different project using the library. Given a test case, $\mathcal{L}^{\mathcal{T}}$, that witnesses the vulnerability, our goal is to assess if the library vulnerability can be exploited in the client program by deriving a test case $\mathcal{C}^{\mathcal{T}}$ for the client program that mimics $\mathcal{L}^{\mathcal{T}}$. We refer to $\mathcal{C}^{\mathcal{T}}$ as the *exploit* (from the client program) and $\mathcal{L}^{\mathcal{T}}$ as the *vulnerability-witnessing test case* (from the library code). We consider that a library vulnerability can be **exploited** from the client project if $\mathcal{C}^{\mathcal{T}}$ can be generated. $\mathcal{C}^{\mathcal{T}}$ is a test case invoking public functions of the client program and transitively executes VF to exhibit the same behavior witnessed by $\mathcal{L}^{\mathcal{T}}$.

8.3.2 Approach

Figure 8.4 presents an overview of our tool, TRANSFER. TRANSFER takes as input 1) the vulnerability-witnessing test case, $\mathcal{L}^{\mathcal{T}}$, from an open-source library, 2) the vulnerable library function, *VF*, and 3) the code of the client project that uses the vulnerable library. After instrumenting and executing $\mathcal{L}^{\mathcal{T}}$, TRANSFER extracts the *triggering conditions*, σ, from the carved program state. TRANSFER generates a test case (that executes the client program) that satisfies σ if it finds one within the search budget.

Execution of $\mathcal{L}^{\mathcal{T}}$

Test mimicry relies on information from $\mathcal{L}^{\mathcal{T}}$. TRANSFER instruments the library code to carve $\mathcal{L}^{\mathcal{T}}$, TRANSFER executes $\mathcal{L}^{\mathcal{T}}$ to invoke the vulnerable library function, *VF*, and collects the sequences of function calls and arguments to construct and initialize objects. Based on inputs and outputs of *VF*, the relevant parts of the program state are extracted (① in Figure 8.4).

Search Goals

To achieve the high-level objective of reproducing the vulnerable behavior, TRANSFER attempts to direct test case generation towards code that executes *VF* and bring the program to the same state as $\mathcal{L}^{\mathcal{T}}$. To do so, TRANSFER constructs the search goals (② in Figure 8.4) that will guide the test generator. TRANSFER uses two types of search goals: line coverage goals and a goal that represents the vulnerability's triggering conditions. After computing a static call graph, TRANSFER determines the paths from the client program that lead to *VF*. For the functions on the paths, line coverage goals are constructed for the lines in the functions. From the carved object states obtained from the execution of $\mathcal{L}^{\mathcal{T}}$, TRANSFER extracts the vulnerability's *triggering conditions*, σ.

Each search goal type is associated with a fitness function that estimates the distance of a test case from satisfying the goal. For a line coverage goal, the fitness functions measure how close the test case came to covering the line. For the triggering conditions, the fitness function is an estimate of similarity of the program state when invoking VF to σ (described in Section 8.3.3).

Line coverage goals for directed test generation. While our objective is to satisfy the conditions of triggering the vulnerability when transitively invoking *VF*, generating a test case that invokes *VF* through the client program can be challenging. To address this challenge, TRANSFER

constructs line coverage goals for the lines of the functions in the call graph between the client program and VF, directing the search towards test cases that are closer (covering code on paths in the call graph that end at VF) to invoking VF.

TRANSFER uses a standard fitness function for line coverage using branch distance [283, 66]. Based on the control dependencies of the line, the branch distance heuristically estimates how close a test case is to taking a correct branch that the line depends on (e.g. how close the branch predicate is to becoming true) when an undesired control-flow path is taken [283, 66].

Vulnerability-triggering conditions. TRANSFER carves $\mathcal{L}^{\mathcal{T}}$ to obtain the program state associated with the triggering of the vulnerability, from which it extracts values and properties of the objects to form the vulnerability triggering conditions. Test cases are generated and evolved towards satisfying the triggering conditions.

To guide test generation towards the vulnerable behavior, the fitness functions favor test cases that, through functions of the client project, invokes VF with inputs and output having states that come closer to satisfying σ. We provide more details in Section 8.3.3.

Evolutionary Test Generation

For test generation, TRANSFER uses an evolutionary algorithm to favor the generation of fitter test cases (③ in Figure 8.4, with a simplified version shown in Algorithm10). TRANSFER seeds the object pool with the carved objects from $\mathcal{L}^{\mathcal{T}}$, allowing TRANSFER to invoke the function calls to create complex objects from the library. Due to EvoSuite's maturity and effectiveness [55, 134], we use its representation (i.e., a sequence of function calls) and implementation of test chromosomes, crossover, and mutation operators in TRANSFER. TRANSFER uses the multi-objective DYNAMOSA [311] algorithm, shown to be the state-of-the-art for search-based test generation.

The DYNAMOSA [311] algorithm was proposed for test generators to fully achieve the coverage of each individual goal. Other algorithms may instead converge in a local optimum that minimizes its distance from multiple objectives, but does not necessarily satisfy any individual goal [311]. This characteristic of DYNAMOSA makes it suitable for our work as our primary focus is to reproduce the program state of the vulnerability regardless of how close the test case is to covering the other (line coverage) goals.

TRANSFER's effectiveness stems from *test mimicry*, i.e. dynamically constructing search goals determined from the execution of $\mathcal{L}^{\mathcal{T}}$. In TRANSFER, the test case with the highest fitness with respect to σ is produced as the output, $\mathcal{C}^{\mathcal{T}}$, if its fitness is sufficiently high (set to a fitness threshold of

0.9 in our experiments).

Once the triggering conditions are met or the search budget has been used up, TRANSFER outputs the test case by considering only the search goal representing the triggering conditions. We do not consider line coverage goals when determining the best test case, as an exploit, $\mathcal{C}^{\mathcal{T}}$, may not have covered all lines in the functions between the client program and *VF* in the call graph.

8.3.3 Satisfying a vulnerability's triggering conditions

TRANSFER's primary objective is the reproduction of the behavior revealed by the $\mathcal{L}^{\mathcal{T}}$. To detect that a test case has reproduced the same behavior, TRANSFER uses the program state relevant to the input and output of *VF* carved from the execution of $\mathcal{L}^{\mathcal{T}}$, extracting a set of conditions as the *triggering conditions*, σ. σ abstracts over the program state reached by $\mathcal{L}^{\mathcal{T}}$.

To detect that a generated test case satisfies σ, TRANSFER compares the inputs and outputs of *VF* with σ. If the test case successfully (transitively through the client program) invokes *VF* with inputs and outputs that match σ, then σ is satisfied and we consider that the test case has invoked *VF* with the same behavior as $\mathcal{L}^{\mathcal{T}}$. We emphasize that the test generator **does not produce test cases that directly call** *VF*. Instead, TRANSFER produces test cases that execute the client program, i.e., *VF* is transitively invoked through the client program.

For example, based on the vulnerability discussed in Section 8.2.3, TRANSFER identifies that the vulnerability is triggered from the program state reached by passing "blah@goggle.com:80@google.com" as an input to a function, `URIUtils.extractHost` used by the library to extract the host of a URL. TRANSFER generates test cases of the client program that calls the library functions that may directly or transitively invoke the `URIUtils.extractHost` function.

During test generation on the client project, we compute the fitness of the test case with respect to the triggering conditions. The fitness function estimates the similarity of the program state reached by the generated test case as compared to $\mathcal{L}^{\mathcal{T}}$.

Vulnerability triggering conditions. The triggering conditions, σ, are predicates over the values of the input and output of the *VF*. To detect if σ is satisfied, we can view it as the comparison of the relevant objects captured in σ during the execution of $\mathcal{L}^{\mathcal{T}}$ (② in Figure 8.4) and their corresponding values currently observed during the invocation of *VF* (as part of test generation, ③ in Figure 8.4). The input of *VF* is its method receiver and arguments, and its output is either its return value or an exception thrown

during its execution. To simplify our description, let us define ***actuals*** to refer to the inputs and output of *VF* when transitively invoked (through the client program) by a generated test case. We define ***references*** as the corresponding values in σ, which capture the input and output values observed during the execution of $\mathcal{L}^{\mathcal{T}}$. During test generation, if all of *actuals* match their corresponding parts in *references*, then σ is satisfied and $\mathcal{C}^{\mathcal{T}}$ has been found.

The similarity between *actuals* and *references* is the average similarity between the individual elements of *actuals* and their corresponding element in *references*. The best similarity value is 1 and the worst similarity value is 0.

$$similarity(actuals, references) = \frac{1}{N} \times \sum_{i=1}^{N} similarity(actuals[i], references[i])$$

For comparison between individual elements of *actuals* and *references*, TRANSFER compares them based on their type. We consider the following types of values:

- Primitive and enumeration values: For primitive values (e.g., `long`, `int`, `char`) as well as values of enums, TRANSFER directly compares their values.
- String-like values: For values of types `byte[]`, `char[]`, `String`, TRANSFER compares them by computing the edit distance normalized by the maximum length of the values.
- Objects: To compare two objects, TRANSFER compares their externally observable states [127].
- Files: For files, TRANSFER treats them as objects, however, any files read by $\mathcal{L}^{\mathcal{T}}$ are also made available for reading to $\mathcal{C}^{\mathcal{T}}$. File names are stored in the constant pool to allow test cases to have a greater chance of reading the file.
- Exceptions: Exceptions can be thrown by *VF*. TRANSFER considers any exception as the output of *VF*, comparing its type and exception message.

Comparing primitive or enumeration values

TRANSFER directly compares primitive values and enums. Primitive types include long, int, short, double, float, char, boolean. If *actual* matches *reference*, then a similarity value of 1 is returned, otherwise a similarity value of 0 is returned.

Algorithm 11: *similarity_stringlike*(*actual*, *reference*): Computing the similarity of *actual* and *reference* when they are string-like (e.g. `String`, `char[]`)

Inputs: **1.** *actual*, one of the objects in *actuals*.
2. *reference*, one of the objects in *references*.
Output: *similarity* between *actual* and *reference*, a float between 0 and 1.
function *similarity_stringlike*(*actual*, *reference*):

```
1 edit_dist = edit_distance(actual, reference)
2 max_len = max(length(actual), length(reference))
3 bug localization techni_len > 0 ? 1 - edit_dist / max_len : 0
```

Comparing string-like values

For values that are string-like (i.e., `char[]`, `byte[]`, `String`), As shown in Algorithm 11, during the generation of test cases, TRANSFER computes the edit distance between the two string-like values. The fewer edits needed to convert one value to the other, the more similar they are. The similarity is then obtained by subtracting the edit distance normalized by the length of the longer value from 1.

Comparing objects

For objects, TRANSFER characterizes them by their externally observable state, using inspector functions [127] defined on the class. Inspector functions refer to public methods that return a value and are side-effect free (determined through a simple static analysis [158]). The state of each object is then constructed using the values returned by the inspectors. An example object state for a `java.net.URI` object is given in Figure 8.5.

Similarity of two objects Algorithm 12 shows the computation of the similarity between *actual* and *reference*. First, TRANSFER checks for `null` for both the *actual* and *reference* objects (Lines 1-9). If both of them are `null`, then TRANSFER considers them to be a match, otherwise, if either *actual* or *reference* is `null`, TRANSFER assigns the lowest similarity score (similarity=0).

TRANSFER compares inspectors only if both of them are non-null and are objects of the same class (Lines 10 to 17). The similarity of the objects is computed as the average similarity between the inspectors of the objects, which will be described in the next paragraph. *similarity*(*actual*, *reference*) returns a value between 0 to 1, where 1 is the highest similarity. TRANS-

```
{
  getHost() -> "google.com",
  getPath() -> "/",
  getPort() -> 80,
  ...
  isOpaque() -> true,
  isAbsolute() -> true,
}
```

Figure 8.5: Simplified example of the object state of a URI object, identified through a mapping of its inspector functions to their return values

FER conservatively rejects the match if no inspectors are found (Line 16).

Similarity of two objects w.r.t an inspector. Algorithm 13 shows the computation of the similarity of *actual* and *reference* with respect to a single, given inspector. Its return value is between 0 and 1, inclusively. For each inspector function, TRANSFER uses a different similarity function based on the function return type. If the return type of the inspector is primitive, then the similarity of the two values is computed using *similarity_primitive* (described in Section 8.3.3). If the return type is string-like, then *similarity_stringlike* (Section 3) is invoked on the return values of the inspectors instead. Otherwise, if the inspector returns another object, we recursively invoke *similarity* (Figure 12). To avoid infinite recursion (e.g. a function returning the another object of the same class), a depth parameter can be configured in TRANSFER. For example, the `normalize()` function of a `java.net.URI` returns another `java.net.URI` object. By default, we limit our consideration of objects only up to a depth of two.

Comparing files

For files that are accessed during the execution of the vulnerability-witnessing test, $\mathcal{L}^{\mathcal{T}}$, they are treated as objects (e.g. typically files are accessed through an `InputStream` object or a `File` object, and can be treated as such). The files on the filesystem are copied and made available during the generation of $\mathcal{C}^{\mathcal{T}}$. The filenames are added as strings to the constants pool, which allows TRANSFER to use the filename when generating test cases. As they are treated as objects, TRANSFER uses the same similarity function (Algorithm 12).

Algorithm 12: *similarity*(*actual*, *reference*): computing the similarity of two objects, *actual* and *reference*.

Inputs: 1. *actual*, one of the objects in *actuals*.
2. *reference*, one of the objects in *references*.
Output: *similarity* between *actual* and *reference*, a float between 0 and 1.
function *similarity*(*actual*, *reference*):

```
1  if reference == null and actual == null then
2  |  return 1
3  end
4  if reference == null or actual == null then
5  |  return 0
6  end
7  if actual.getClass() ≠ reference.getClass() then
8  |  return 0
9  end
10 ins = get_inspectors(actual)
11 if ins is not empty then
12 |  obj_sim ← Σ_{in∈ins} similarity(actual, reference, in)
13 |  obj_sim ← 1/|ins| × obj_sim
14 |  return obj_sim
15 else
16 |  return 0
17 end
```

Comparing exceptions

TRANSFER compares exceptions thrown by *VF* by their types (e.g. `NullPointerException`) and exception messages. While other studies [133] have used more sophisticated methods of comparing exceptions, we find that the use of the exception type and message is sufficient to guide test generation in our experiments.

Computing the fitness of a test case with respect to σ

From a single test case, the client program may invoke *VF* multiple times. To compute the fitness score of a test case, TRANSFER takes the highest similarity obtained among the multiple invocations of *VF*. Each invocation of *VF* is associated with a set of inputs and outputs. The similarity of an invocation, *inv*, to σ is computed based on *similarity*(*actuals*, *references*),

Algorithm 13: *similarity*(*actual*, *reference*, *inspector*): Computing the similarity score of an object to a reference object with respect to a given inspector function

Inputs: **1.** *actual*, an object from *actuals*
2. *reference*, an object from *references*
3. *inspector*, a single inspector function
Output: *similarity*, between 0 and 1. 1 indicates that the two objects are equivalent wrt to *inspector*.
function *similarity*(*actual*, *reference*, *inspector*):

```
1  actual_ins ← inspector(actual)
2  reference_ins ← inspector(reference)
3  if inspector returns a string-like value then
4  |  return similarity_stringlike(actual_ins, reference_ins)
5  end
6  if inspector returns a primitive value then
7  |  return similarity_primitive(actual_ins, reference_ins)
8  end
9  if inspector returns another object then
10 |  return similarity(actual_ins, reference_ins)
11 end
```

where *actuals* are the inputs and output associated with *inv*. In turn, the test case is as fit as the similarity of the invocation most similar to σ:

$$similarity(inv, references) = similarity(actuals, references)$$
$$fitness(testcase) = max(similarity(inv)), inv \in invocations$$

8.3.4 Implementation

We implement our tool, TRANSFER, that realizes our novel framework of test mimicry, on top of the implementation of the genetic algorithm and other infrastructure of EvoSuite 1.1.0. TRANSFER uses the DYNAMOSA [311] evolutionary algorithm for generating test cases. TRANSFER uses the default configuration of EvoSuite as prior work has shown that tuning EvoSuite's parameters does not lead to improved performance over its default configuration [67]. The size of the population of test cases is 50 individuals. The default crossover operator is used, which is the single-point crossover with probability of 0.75. Test cases are selected using tournament selection, with a tournament size of 10. We use ASM [98] to instrument the code, in particular, the vulnerable function *VF*.

8.4 Empirical Evaluation

Our experiments aim to answer the following research questions:

RQ1. Can TRANSFER generate exploits in client programs that demonstrate library vulnerabilities?

This research question is concerned with the efficacy of TRANSFER in generating test cases that reveal client usage of vulnerable library functions. Using a benchmark of library vulnerabilities and code from client projects that we have identified by hand, we compute the number of true positives and compute its accuracy. We use SIEGE [203] as the baseline for comparison.

RQ2. How do the different search goals affect the effectiveness of TRANSFER?

TRANSFER has two types of search goals: the primary search goal of satisfying the triggering conditions, and line coverage goals for directing test generation towards the vulnerable library function. We investigate the impact of the goal types on the effectiveness of TRANSFER via an ablation study.

8.4.1 Experimental Setup

Experimental subjects. In our experiments, we analyzed 22 recent vulnerabilities from 18 libraries. We manually selected the vulnerabilities for their diversity; the vulnerabilities manifest a range of behaviors, ranging from timeouts and out-of-memory errors (crash behaviors) to incorrect functional behaviors (non-crash behaviors). Similarly, we selected widely-used libraries that cover a range of domains, from file compression (Junrar), parsing XML (Jackson, XStream) and JSON (Json-Smart, OSWAP Json), cryptographic libraries (Bouncy Castle), to HTTP requests (HttpClient). Consequently, the vulnerabilities are not restricted to a single domain. Apart from considering vulnerabilities publicly disclosed as CVEs in the NVD database, we studied several vulnerabilities that were fixed silently [352, 115]. These vulnerabilities were identified from publicly accessible vulnerability databases [38, 39].

The vulnerabilities used in our experiments are provided in Table 8.1. Half of these vulnerabilities (11 vulnerabilities) result in crashes, exceptions, or timeouts. The other 11 vulnerabilities do result in non-crashing behaviors. For example, two vulnerabilities from Apache HttpClient manifest as incorrect functional behavior (e.g. constructing a URI with an incorrect hostname).

For each vulnerability, we identified the vulnerable library function and a vulnerability-witnessing test case through **manual analysis**. Next, we

Table 8.1: The vulnerabilities used in our experiments, including the names and descriptions of the vulnerable libraries, and the effect of the vulnerability. ✕ indicates that an exploit could not be generated. A single ✓ indicates one exploit for a client program was successfully constructed. Two ✓ indicates that exploits were successfully constructed for both client programs.

Vulnerability	**Library**	**Siege**	**TRANSFER**
CVE-2020-28052	Bouncy Castle	✕	✓✓
CODEC-134 [12]	Apache Codecs	✕	✓✓
CVE-2020-13956	Apache HttpClient	✕	✓✓
HTTPCLIENT-1803 [35]	Apache HttpClient	✕	✕
CVE-2019-14900	Hibernate	✕	✕
CVE-2020-15250	JUnit	✓✓	✓✓
CVE-2021-23899	OWASP JSON Sanitizer	✕	✓
CVE-2020-26217	XStream	✕	✓✓
CVE-2019-12415	Apache POI	✕	✕
CVE-2018-1000632	Dom4J	✕	✓
CVE-2020-10693	Hibernate	✕	✕
CVE-2018-1000873	Jackson	✕	✕
CVE-2019-12402	Apache Compress	✕	✓
CVE-2018-12418	Junrar	✕	✓✓
CVE-2019-10094	Apache Tika	✕	✓
TwelveMonkeys-595 [33]	TwelveMonkeys	✕	✕
CVE-2020-28491	Jackson	✕	✕
CVE-2018-1274	Spring Framework	✕	✕
CVE-2021-27568	Json-smart	✓	✓
Zip4J-263 [34]	Zip4J	✓✓	✓✓
Spring Security-8317 [36]	Spring Framework	✕	✓✓
CVE-2017-7957	XStream	✕	✓✓
	Total	5	23

manually identified up to two vulnerable, real client projects for each library vulnerability. For the 22 vulnerabilities, we investigated a total of 64 client projects with source code available on GitHub. From 64 client projects, we obtained 42 pairs of (vulnerability, client program) where the client program is able to trigger the library vulnerability. In the other 24 cases, we[2] confirmed that the client programs are unable to trigger the vulnerability despite calling the vulnerable function, using this to validate that TRANSFER does not generate test cases unnecessarily. All pairs that we investigated are provided on the project website[3].

While SIEGE was evaluated on 11 vulnerabilities by Iannone et al. [203], the experiments involved only toy client programs that were written by hand. As such, the client programs may not reflect actual usage of the libraries. We do not use handwritten experimental programs in our experiments. Instead, we use realistic client programs from open-source repositories on GitHub.

Vulnerabilities may depend on many conditions to be triggered, including environmental factors such as software configuration. These environmental factors differ between vulnerabilities. In our experiments, we do not model environmental factors (e.g. specific configurations of the library or client project) or global variables, although it is possible to expand a vulnerability's triggering conditions to account for them.

Baselines. As baseline, we compare TRANSFER to SIEGE [203]. SIEGE is a test generator that targets a single vulnerable line of code given as input. We use the code provided in the replication package of SIEGE. To use SIEGE in our experiments, we identified a line of code for each vulnerability. In the ablation study, after removing both types of goals from TRANSFER, we compare TRANSFER against EVOSUITE, which optimizes for code coverage in the client program.

Experimental setup. We ran TRANSFER and SIEGE for up to 10 minutes. In studies related to test generation, experimental durations range from a few minutes [91, 133, 370, 338] to several days [194]. We selected 10 minutes for the same experimental duration as studies on crash reproduction [370]. Our experiments were run on a machine with 2.3 GHz Dual-Core Intel Core i5 with 8GB of RAM.

To mitigate the effect of randomness, we repeat each run 20 times. We consider that a vulnerability is shown to be exploitable from a client program if a tool (i.e., TRANSFER or SIEGE) is able to construct an exploit at least 50% of the time, similar to previous studies on crash reproduction [370].

A true positive ("TP") is the successful generation of a test case for

[2]The first and second authors manually investigated the client projects.

[3]`https://github.com/soarsmu/transfer`

Table 8.2: Number of true positives, false negatives, and proportion of vulnerabilities detected by TRANSFER, TRANSFER without directed test generation, TRANSFER without the triggering conditions, and a baseline EvoSuite.

Approach	**# TP**	**# FN**	**% detected**
TRANSFER	23	19	55%
– directed generation	20	22	48%
– triggering conditions	4	38	10%
– both (EvoSuite)	4	38	10%

the client program that triggers the library's vulnerability. A false negative ("FN") is the failure to generate a test case triggering the library's vulnerability when the vulnerability can be triggered. To determine if a test case triggers the vulnerability, we manually inspected and ran them to check that they indeed recreated the conditions that triggers the vulnerabilities. We compute the proportion of vulnerable client programs for which TRANSFER/Siege successfully generates an exploit.

8.4.2 Experimental Results

RQ1. Efficacy of TRANSFER

The client projects and vulnerabilities detected by TRANSFER are given in Table 8.1. TRANSFER confirmed the exploitability of the library's vulnerability in 55% (23 / 42) of the client programs. In contrast, Siege [203] was only able to construct an exploit for 5 client projects (12% of the total vulnerable client programs). TRANSFER outperforms Siege in 13 cases. This shows that TRANSFER can generate exploits for client programs, outperforming the state-of-the-art test generator by a large margin (over 40% increase in number of generated exploits).

Among the client programs where TRANSFER succeeded in generating an exploit, TRANSFER required an average of 27 generations, and took an average of 41 seconds to produce an exploit. This suggests that TRANSFER could quickly construct exploits for the vulnerabilities in our experiments.

A desirable quality of TRANSFER is the absence of false alarms. Therefore, TRANSFER should not produce a test case when the vulnerability cannot be exploited. Among the 24 cases where the client programs are unable to trigger the vulnerability, we found that TRANSFER exhibited the correct behavior by not producing test cases. Note that the call graphs com-

puted for these pairs indicate that the vulnerable function could be invoked from the client programs, suggesting that existing call graph-based detectors would incorrectly report false alarms.

Among the total of 17 libraries considered, TRANSFER is able to trigger the vulnerabilities in 14 libraries. Of the 22 vulnerabilities, 11 are not crash-related and TRANSFER detects 7 of these 11 vulnerabilities. This indicates that TRANSFER is able to detect non-crash vulnerabilities. As for the other 11 vulnerabilities with exceptions, crashes, or timeouts, TRANSFER detects 7 of them, suggesting that TRANSFER is effective for both crash and non-crash vulnerabilities.

Answer to RQ1: TRANSFER is able to generate exploits for vulnerabilities for 14 libraries, producing 23 exploits for 64 client programs. The baseline SIEGE was only able to generate 5 exploits.

RQ2. Ablation study

Next, we performed an ablation study on TRANSFER by disabling the search goals accordingly. Table 8.2 shows the results of this experiment.

Without the primary goal of the triggering conditions, TRANSFER created exploits of only 3 vulnerabilities from the client programs. In this setting, TRANSFER is equivalent to EVOSUITE if EVOSUITE considers only the line and branch coverage goals of the path along the call graph and the vulnerable function are provided. The decrease in effectiveness shows that the triggering conditions are essential. Without them, TRANSFER is unable to select a test case that confirms that the vulnerability is exploitable.

By dropping the line coverage goals that direct test generation towards the vulnerable library function, TRANSFER is unable to trigger the vulnerability in 3 cases (*CVE-2020-28052*, *CVE-2020-13956*, *CVE-2019-10094*). Without the line coverage goals, TRANSFER had to generate a test case that reached the vulnerable function through completely random mutations. As such, the generated test cases are less likely to invoke the vulnerable function.

Table 8.3 shows the four cases where we observed that directed test generation substantially changed the number of generations required for TRANSFER to discover an exploit. For the other cases, we did not notice a significant change in the number of generations required. In 3 of 4 cases, the line coverage goals allowed TRANSFER to discover the exploit in a smaller number of generations. However, in one case (*CODEC-134*), the line coverage goals slowed down the search. In the other 16 cases, TRANSFER performed similarly with or without the line coverage goals.

Table 8.3: The average number of generations required (number of seconds given in parentheses) for TRANSFER to find an exploit, with and without the use of the line coverage goals that direct test generation towards code on the call graph between the client program and the vulnerable function.

Vulnerability	**TRANSFER**	**−directed test generation**
CODEC-134	34 (21s)	20 (15s)
CVE-2020-13956	26 (41s)	58 (50s)
CVE-2021-23899	2 (20s)	6 (108s)
CVE-2019-12402	7 (11s)	343 (328s)

Along with the 3 cases where TRANSFER could not reveal the vulnerability in the client programs, there are a total of 7 cases (41% of the 17 cases where TRANSFER could detect the vulnerability in the client program) where the line coverage goals increased the effectiveness of TRANSFER. Therefore, we see that the line coverage goals were important, although not essential, in our experiments.

Answer to RQ2: While test generation directed by line coverage helped, guiding test generation to satisfy the triggering conditions extracted from the carved program state was essential to TRANSFER.

8.5 Discussion

We qualitatively analyse our results to better understand the performance of TRANSFER and discuss threats to validity.

8.5.1 Qualitative Analysis

Why did TRANSFER work?

Generating complex inputs. Both TRANSFER and SIEGE direct test generation towards the vulnerable library code. However, there are 18 client programs where SIEGE was not able to uncover the library vulnerability while TRANSFER was able to. Our ablation study reveals that capturing the triggering conditions of the vulnerabilities was the key to TRANSFER's effectiveness. Indeed, many vulnerabilities are corner cases and exceptional behaviors of the software system, and TRANSFER targets the triggering conditions, which provides better guidance for uncovering the vulnerability.

```
arg1.getRawAuthority()==
    "blah@goggle.com:80@google.com"
&& arg1.getScheme() == "http"
&& arg1.getRawPath() == "/"
...
```

Figure 8.6: Part of the triggering conditions identified for CVE-2020-13956

As an example, we use CVE-2020-13956 from Section 8.2.3, a simplified version of the triggering conditions captured from the library's test case is shown in Figure 8.6. The vulnerability manifests when an invalid URL (e.g. "blah@goggle.com:80@google.com") is passed into the vulnerable function, and causes a connection to an unexpected host. Neither EvoSuite nor Siege account for the necessary inputs and program state for the exploit. EvoSuite aims to covers all other behaviors, while TRANSFER targets a single behavior encapsulated by the triggering conditions, σ. Siege has to construct an invalid URL from scratch, while TRANSFER leverages the input from the library's test case.

Guidance from triggering conditions. In Table 8.2, TRANSFER using only directed test generation without the triggering conditions performs identically to EvoSuite. Given that only one class is targeted, the search budget of 10 minutes may have provided enough time for EvoSuite to cover the same code locations that TRANSFER was directed to. This indicates that the guidance from the triggering conditions contributed to the improvements of TRANSFER over EvoSuite.

Compared to Siege, TRANSFER and Siege fundamentally differ in how they guide the test generator. Siege checks for the coverage of a vulnerable line of code in the library, while TRANSFER checks if program state reached by the library test case has been reproduced. In 7 vulnerabilities in our experiments, both benign and malicious inputs would cover the same lines of code, and, hence, Siege does not succeed in creating an exploit. In these cases, guiding the test generator by code coverage may not be enough for producing an exploit.

Complementary to existing tools.

We suggest test mimicry is complementary to existing, call graph-based techniques [156, 328]. Call graph-based techniques produce false positives, which may lead to low adoption [239, 215]. If TRANSFER succeeds in demonstrating that a vulnerability can be exploited, then developer effort can be

reduced. However, if TRANSFER fails in generating a test case, it does not mean that the vulnerability is not exploitable, and developers will have to expend effort in investigating the vulnerability.

Test mimicry relies on a test case from the library. While the test case demonstrates the vulnerability, the extracted triggering conditions, σ, do not characterize all possible triggers of the vulnerability. For example, the test case of CVE-2020-13956 (Figure 8.6) shows one possible URL out of many that could trigger the vulnerability. Nevertheless, as our goal is to demonstrate that the vulnerability may be exploited from a client program, having a single example may suffice to guide the creation of a test case demonstrating the vulnerability's exploitability.

Next Steps

While TRANSFER was successful in a number of cases in our evaluation, we identified some limitations to be addressed in future work.

Dependence on inspector functions. In *TwelveMonkeys-595*, TRANSFER is unable to produce a test case as the arguments to the vulnerable function lacked inspector functions. TRANSFER could not extract triggering conditions that capture the program state. In the future, we plan to additionally use other means of extracting vulnerability-triggering conditions beyond the use of inspectors.

Irrelevant program states. In Figure 8.6, the triggering conditions captured aspects that were irrelevant to triggering the vulnerability (e.g. `getScheme()` == "http") along with the the necessary aspects (having `getRawAuthority()` set to an invalid URL, "blah@goggle.com:80@google.com"). In other words, the triggering conditions captured by TRANSFER are more specific than the actual conditions required to trigger the vulnerability. While TRANSFER is able to construct a test case that satisfies the triggering conditions in Figure 8.6, including its irrelevant aspects, we found that in other cases, TRANSFER may have been limited by its inability to satisfy the irrelevant aspects of the captured triggering conditions. In the future, we will explore techniques from the area of test input minimization for the extracted triggering conditions.

8.5.2 Threats to Validity

A threat to validity is the selection of vulnerabilities in our experiments. The vulnerabilities and client projects were selected through manual analysis. While the type of test case is not a limitation of the method, the test case has to target only the vulnerable behavior (i.e., it does not exercise unrelated

behaviors of the vulnerable code location). Otherwise, there is no way to correctly identify the right behavior associated with the vulnerability.

Our experiments focused on the effectiveness of the test mimicry technique. As such, we focused on evaluating the design decisions unique to TRANSFER and used the default parameters of EvoSuite. If we fine-tune the parameters of the search parameters, TRANSFER may perform better in our experiments and we leave these experiments for future work. Likewise, we used the DynaMOSA search algorithm as it has been shown to outperform other search algorithms in generating test cases [311, 102].

Regarding the generalizability of our findings, we selected vulnerabilities that range across multiple domains for our experiments. Table 8.4 shows the description of each library in our experiments while Table 8.5 shows the number of successfully generated test cases for the domains of libraries considered in our experiments. As our experiments covered various domains, we believe our approach is not limited to specific domains and we expect it to generalize to domains beyond our evaluation. Still, it is possible that it will not work on some domains that we have not identified yet. Compared to the benchmark of Iannone et al. [203], our benchmark is larger and uses realistic client programs instead of toy programs.

8.6 Related Work

Software Composition Analysis and Search-based Test Generation are discussed in Section 8.2. Here, we discuss other related works.

Automated Exploit Generation. Our work is related to the field of Automated Exploit Generation [73, 107], with the same goal of producing exploits for known vulnerabilities. Brumley et al. [97] have shown the possibility of generating exploits based on the patch fixing the vulnerability. While our work relies on information in a patch, we demonstrate that the test case accompanying the bug fix can be used to generate exploits, while prior work used only the modified code. Moreover, Brumley et al. [97] only creates exploits of vulnerabilities associated with missing input validation.

Other studies that automatically generate exploits study specific types of vulnerabilities for a narrow range of software systems [418, 190, 411, 197, 113, 434]. For example, Chen et al. [113] generate exploits for vulnerabilities caused by out-of-bounds writes, generating 11 exploits. You et al. [434] proposes the use of natural language processing to extract information to guide fuzzing, triggering 18 vulnerabilities in the Linux kernel. In contrast, TRANSFER construct test cases for a wide range of vulnerabilities for Java programs.

Table 8.4: Descriptions of the libraries and vulnerabilities analyzed in this study

Vulnerability	**Library**	**Effect of vulnerability**
CVE-2020-28052	Bouncy Castle	wrong functional behavior
CODEC-134 [12]	Apache Codecs	wrong functional behavior
CVE-2020-13956	Apache HttpClient	wrong functional behavior
HTTPCLIENT-1803 [35]	Apache HttpClient	wrong functional behavior
CVE-2019-14900	Hibernate	SQL injection
CVE-2020-15250	JUnit	wrong file permissions
CVE-2021-23899	OWASP JSON Sanitizer	arbitrary code injection
CVE-2020-26217	XStream	remote code execution
CVE-2019-12415	Apache POI	XXE Injection
CVE-2018-1000632	Dom4J	XXE Injection
CVE-2020-10693	Hibernate	bypass input sanitization
CVE-2018-1000873	Jackson	slow performance
CVE-2019-12402	Apache Compress	infinite loop
CVE-2018-12418	Junrar	infinite loop
CVE-2019-10094	Apache Tika	infinite loop
TwelveMonkeys-595 [33]	TwelveMonkeys	infinite loop
CVE-2020-28491	Jackson	out of memory
CVE-2018-1274	Spring Framework	out of memory
CVE-2021-27568	Json-smart	exception
Zip4J-263 [34]	Zip4J	exception
Spring Security-8317 [36]	Spring Framework	exception
CVE-2017-7957	XStream	exception

Table 8.5: The domains of the libraries considered and the number of generated exploits in our experiments.

Domain	**Libraries**	**# exploits**
File compression	Zip4J, Commons Compress	5
Serialization/deserialization	XStream, jackson, dom4j	6
Web development/utilities	Spring, Hibernate ORM	4
Data encoding/cryptography	Bouncy castle, Apache POI	4
File formats (PDFs, images)	Apache Tika, Twelvemonkey	1
Test Framework	JUnit	1

Test input generation. There is a large amount of work on test input generation [1, 219, 91, 282, 195, 194, 308, 253, 394, 89, 111, 164]. These studies use fuzzing or symbolic execution, focus on maximizing coverage [1, 91, 308, 111, 164], or finding vulnerabilities with oracles such as crashes or poor performance [253, 256, 322]. Compared to these studies, apart from generating test inputs (e.g. a string), TRANSFER constructs the sequence of different function calls (e.g. constructors, setter functions) that set up the object states necessary to trigger the vulnerability. Provided with a vulnerability-witnessing test case, TRANSFER focuses on demonstrating a single, vulnerable behavior and is able to detect vulnerabilities that do not manifest as crashes.

Directed fuzzing [89, 410, 109] focuses fuzzing efforts on selected code locations. Similarly, TRANSFER directs test generation towards the vulnerable code location. However, beyond reaching the same code locations, TRANSFER focuses on reproducing the program state reached in a library test case to check if a library vulnerability can be exploited from a client program.

Crash reproduction. Botsing [133] is an approach that uses search-based test generation to reproduce crashes given stacktraces. Botsing is complementary to our approach in detecting vulnerabilities that result in crashes. However, not all vulnerabilities manifest as exceptions and Botsing cannot be used without a stacktrace. As such, we do not use Botsing in our experiments as just 4 out of the 22 studied vulnerabilities result in exceptions.

Reusing test cases. Researchers have proposed techniques that reuses existing test cases from other projects [448]. Given similar but not identical programs, Zhang and Kim [448] try to reuse existing test cases to enable behavioral comparison between the programs. Our work is similar in reusing properties of an existing test case in a different software system, but we focus on generating tests for client programs rather than similar programs. While Mujahid et al. [293] proposes the use of client test cases to detect breaking changes in libraries, our work proposes the use of library test cases to detect the exploitability of library vulnerabilities from client programs.

8.7 Summary

We propose *test mimicry* to leverage vulnerability-witnessing test cases from open-source libraries, generating test cases for client projects that demonstrate the exploitability of the library vulnerabilities. Our tool, TRANSFER, captures the program states reached by the library test cases, directing test case generation for client programs towards reconstructing them.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

Today, software projects frequently use third-party software components, e.g. libraries. Working with libraries and their APIs is, therefore, an essential part of software development. Likewise, improving the reliability of software that reuse existing software components is now a critical task that demands greater research attention.

This dissertation presents several techniques for fortifying software systems, paying particular attention to issues that may arise at the seams [148] of software, where multiple parts of the software system interaction. We summarize the impact of this dissertation in three aspects:

1. We develop techniques to learn patterns regarding API usages and constraints. We employ active learning for mining subgraph patterns from GitHub to train an API misuse detector. We find that active learning can weaken the reliance on frequency to determine the correctness of a usage pattern (Chapter 3). We use search-based test generator to generate execution traces of underexplored API usage patterns, providing counterexamples to incorrectly learned rules (Chapter 5). While one study focuses on static analysis, while the other on dynamic analysis, the experiments from both studies highlight that the importance of the inclusion of uncommon API usage patterns when mining and inferring models.

2. We support existing static analysis and dynamic analysis techniques. Our replication study showed that there is room for improving techniques that postprocess warnings from static analyzers. To reduce the need for a large number of labelled examples, we carefully select a small

number of examples for labelling to maximize their informativeness when employing in-context learning to filter out false alarms (Chapter 4). When building a fuzzer of deep learning libraries, our approach invokes the library with informative inputs that allow for the refinement of its model of the API's input constraints (Chapter 6). This enables the generation of valid, diverse inputs. Both studies had experimental results that suggested that the consideration of small number of informative data points can lead to a large improvement in bug detection effectiveness of both static and dynamic analysis.

3. Finally, we highlight the challenge and importance of analyzing a software system comprising multiple interacting components. In an Internet-of-Things environment, our experiments show that handcrafted rules may lead to large number of false alarms, and may fail to detect malicious behaviors (Chapter 7). When a program includes a library, vulnerabilities in the library may be exploited from the program (Chapter 8). Our work paves the way forward in addressing these problems, proposing approaches that automatically reveal unexpected behaviors and confirming that a vulnerability may be exploited. As developers have to judicious about constructing complex software systems, more work has to go into supporting them.

9.2 Future Work

Here, we describe potential work that may advance the work presented in this dissertation.

Obtaining more types of information from oracles. Our work relies on active learning, in which an oracle is queried for feedback. We explored both the use of feedback from human oracles as well as from the execution of programs. The proposed approaches had an objective to reduce the number of queries presented to the oracle. Still, each query is expensive. Instead of reducing the number of queries, future work may benefit from the exploration of types of queries that may provide *more information*. It may be worthwhile to explore other means of obtaining information and feedback from the oracles, such as the human-in-the-loop. For feedback from human oracles, one possible direction is that a human annotator can provide explanations for each label. For feedback from program executions, collecting more information by applying more instrumentation may be able to provide more contextual information.

More expressive representations and models. We have explored a

wide variety of representations of abstractions for modelling API constraints and usages. Nevertheless, each type of representation make trade-offs between different desirable qualities. A formal, symbolic representation allows for precise rules and reasoning, while a continuous, vector representation seems to better support learning on vast corpora of publicly-available open-source data. Future studies should explore more ways of expressing knowledge about APIs, such as the use of neurosymbolic approaches.

Providing more context and explanations when producing alerts. A recurring theme in this dissertation is the reluctance of developers to adopt tools when provided a large number of false alarms and the lack of precision in models (warnings from static analysis, handcrafted models/rules for monitoring a system, alerts about library vulnerabilities). This suggests that when developer tools are developed, more consideration can be placed on usability by providing more context to the alerts such that inspecting and suppressing a false alarm requires only some effort. Apart from providing greater context, developers are also known to prefer using tools that enable a short feedback loop when making decisions. Future work can explore the types of information that are useful to distinguish between true and false alarms when providing an alert to developers, as well as the interaction between the developer and the tool.

To conclude, as the use of libraries and reusable software components grow, there are tremendous challenges and opportunities. The work in this dissertation has addressed some aspects of software reliability related to the interaction between software systems. We hope our work encourages future work into understanding and addressing the problems we have highlighted. We believe that there will be more advances and breakthroughs made in this field, which will enable greater developer productivity and software reliability when reusing software.

Bibliography

[1] American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/technical_details.txt.

[2] Apache Bigtop. https://github.com/apache/bigtop.

[3] Apache Commons-Lang. https://github.com/apache/commons-lang.

[4] Apache Commons-Math. https://github.com/apache/commons-math.

[5] Apache Commons-Text. https://github.com/apache/commons-text.

[6] Apache Curator. https://github.com/apache/curator.

[7] Apache FOP. https://github.com/apache/fop.

[8] Apache Fortress. https://github.com/apache/directory-fortress.

[9] Apache PDFBox. https://github.com/apache/pdfbox.

[10] Apple HomeKit. https://www.apple.com/ios/home/.

[11] BCEL. https://github.com/apache/commons-bcel.

[12] CODEC-134 from Apache Commons Codecs's issue tracker. https://issues.apache.org/jira/browse/CODEC-134.

[13] CVE-2019-12402. https://nvd.nist.gov/vuln/detail/CVE-2019-12402.

[14] Google Home. https://store.google.com/sg/category/connected_home.

[15] Google security blog: Understanding the impact of apache log4j vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.

[16] H2 database. https://github.com/h2database/h2database.

[17] Home Assistant. https://www.home-assistant.io/.

[18] Hyrum's law. https://www.hyrumslaw.com/. Accessed 2 Dec 2020.

[19] IFTTT. https://ifttt.com/.

[20] Illuminated Response to Unexpected Visitors. https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/IlluminatedResponsetoUnexpectedVisitors.txt.groovy.

[21] IoTBench. https://github.com/IoTBench/IoTBench-test-suite.

[22] IoTBox artifact website. https://github.com/iotboxdeveloper/iotbox.

[23] ITextPDF. https://github.com/itext/itextpdf.

[24] Jackrabbit. https://github.com/apache/jackrabbit.

[25] JFreeChart. https://github.com/jfree/jfreechart.

[26] Lock Door after X minutes. https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/LockDoorafterXminutes.txt.groovy.

[27] Motion Mode Change. https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/MotionModeChange.txt.groovy.

[28] Replication package. https://github.com/SA-retrospective/study.

[29] Report: Millions (and Millions) of Devices Vulnerable in latest Mirai Attacks. https://securityledger.com/2016/11/report-millions-and-millions-of-devices-vulnerable-in-latest-mirai-attacks/

[30] Samsung SmartThings. https://www.smartthings.com/.

[31] Santuario. https://github.com/apache/santuario-java.

[32] SmartThings Simulator. https://graph.api.smartthings.com/ide/apps.

[33] SNYK-JAVA-COMTWELVEMONKEYSIMAGEIO-1083830 from SNYK. https://snyk.io/vuln/SNYK-JAVA-COMTWELVEMONKEYSIMAGEIO-1083830.

[34] SNYK-JAVA-NETLINGALAZIP4J-1074967 from SNYK. https://snyk.io/vuln/SNYK-JAVA-NETLINGALAZIP4J-1074967.

[35] SNYK-JAVA-ORGAPACHEHTTPCOMPONENTS-31517 from SNYK. https://snyk.io/vuln/SNYK-JAVA-ORGAPACHEHTTPCOMPONENTS-31517.

[36] SNYK-JAVA-ORGSPRINGFRAMEWORKSECURITY-570204 from SNYK. https://snyk.io/vuln/SNYK-JAVA-ORGSPRINGFRAMEWORKSECURITY-570204.

[37] Snyk's state of open source security report 2022. https://resources.snyk.io/state-of-open-source-security-report-2022.

[38] Snyk's vulnerability database. https://snyk.io/vuln?type=maven.

[39] SourceClear's vulnerability database. https://www.sourceclear.com/vulnerability-database/.

[40] SwingX. https://github.com/ebourg/swingx.

[41] Unlock Door. https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/Unlockdoor.txt.groovy.

[42] Vacation Light Director. https://community.smartthings.com/t/new-app-vacation-light-director/7230.

[43] Wildfly-Eytron. https://github.com/wildfly-security/wildfly.

[44] Wired: The log4j vulnerability will haunt the internet for years. https://www.wired.com/story/log4j-log4shell/.

[45] Zapier. https://zapier.com/.

[46] Findbugs filter file. http://findbugs.sourceforge.net/manual/filter.html, September, 2021.

[47] OSS-Fuzz. https://github.com/google/oss-fuzz, 2022. Accessed: 2022-10-10.

[48] SkipFuzz's GitHub repository. https://github.com/skipfuzz/skipfuzz, 2022.

[49] TensorFlow security policy. https://github.com/tensorflow/tensorflow/security/policy, 2022. Accessed: 2022-04-20.

[50] Aarts, F., De Ruiter, J., and Poll, E. Formal models of bank cards for free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (2013), IEEE, pp. 461–468.

[51] Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P., and Smith, M. Sok: Lessons learned from android security research for appified software platforms. In *2016 IEEE Symposium on Security and Privacy* (2016), IEEE, pp. 433–451.

[52] Alfadel, M., Costa, D. E., and Shihab, E. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2021), IEEE, pp. 446–457.

[53] Alhanahnah, M., Stevens, C., and Bagheri, H. Scalable analysis of interaction threats in IoT systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020), pp. 272–285.

[54] Allamanis, M. The adverse effects of code duplication in machine learning models of code. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)* (2019), pp. 143–153.

[55] Almasi, M. M., Hemmati, H., Fraser, G., Arcuri, A., and Benefelds, J. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)* (2017), IEEE, pp. 263–272.

[56] Alrawi, O., Lever, C., Antonakakis, M., and Monrose, F. Sok: Security evaluation of home-based IoT deployments. In *2019 IEEE Symposium on Security and Privacy* (2019), IEEE, pp. 1362–1380.

[57] Alshahwan, N., and Harman, M. Augmenting test suites effectiveness by increasing output diversity. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 1345–1348.

[58] Amann, S. *A systematic approach to benchmark and improve automated static detection of Java-API misuses*. PhD thesis, Universitäts- und Landesbibliothek Darmstadt, 2018.

[59] Amann, S., Nadi, S., Nguyen, H. A., Nguyen, T. N., and Mezini, M. MUBench: a benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), pp. 464–467.

[60] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. A systematic evaluation of static API-misuse detectors. *IEEE Transactions on Software Engineering 45*, 12 (2018), 1170–1188.

[61] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 265–275.

[62] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. *ACM Sigplan Notices 37*, 1 (2002), 4–16.

[63] Angluin, D. Learning regular sets from queries and counterexamples. *Information and computation 75*, 2 (1987), 87–106.

[64] Angluin, D. Queries and concept learning. *Machine learning 2*, 4 (1988), 319–342.

[65] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. Understanding the Mirai botnet. In *26th USENIX security symposium (USENIX Security 17)* (2017), pp. 1093–1110.

[66] Arcuri, A. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability 23*, 2 (2013), 119–147.

[67] Arcuri, A., and Fraser, G. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering 18*, 3 (2013), 594–623.

[68] Argyros, G., Stais, I., Jana, S., Keromytis, A. D., and Kiayias, A. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), pp. 1690–1701.

[69] Arp, D., Spreitzenbarth, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket.

[70] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. Nautilus: Fishing for deep bugs with grammars. In *NDSS* (2019).

[71] Asyrofi, M. H., Thung, F., Lo, D., and Jiang, L. AUSearch: Accurate API usage search in github repositories with type resolution. In *IEEE International Conference on Software Analysis, Evolution and Reengineering* (2020).

[72] Asyrofi, M. H., Yang, Z., Yusuf, I. N. B., Kang, H. J., Thung, F., and Lo, D. Biasfinder: Metamorphic test generation to uncover bias for sentiment analysis systems. *IEEE Transactions on Software Engineering (TSE)* (2021).

[73] Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., and Brumley, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[74] Ayewah, N., and Pugh, W. The google findbugs fixit. In *19th International Symposium on Software Testing and Analysis (ISSTA 2010)* (2010), pp. 241–252.

[75] Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., and Penix, J. Using static analysis to find bugs. *IEEE Software 25*, 5 (2008), 22–29.

[76] Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L., and Wang, W. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 975–985.

[77] Balachandran, V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *35th International Conference on Software Engineering (ICSE 2013)* (2013), IEEE, pp. 931–940.

[78] Banerjee, S., Clapp, L., and Sridharan, M. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 740–750.

[79] Bao, L., Le, T.-D. B., and Lo, D. Mining sandboxes: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 445–455.

[80] Barnett, M., Naumann, D. A., Schulte, W., and Sun, Q. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)* (2004).

[81] Bastys, I., Balliu, M., and Sabelfeld, A. If this then what? controlling flows in IoT apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1102–1119.

[82] Bavishi, R., Lemieux, C., Fox, R., Sen, K., and Stoica, I. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–27.

[83] Beck, K. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[84] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)* (2016), IEEE Computer Society, pp. 470–481.

[85] Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M. D., and Krishnamurthy, A. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering 41*, 4 (2014), 408–428.

[86] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), ACM, pp. 267–277.

[87] Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M. D., Pezzè, M., and Castellanos, S. D. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), pp. 242–253.

[88] Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), pp. 109–120.

[89] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017), pp. 2329–2344.

[90] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1032–1043.

[91] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering (TSE) 45*, 5 (2017), 489–506.

[92] Bowring, J. F., Rehg, J. M., and Harrold, M. J. Active learning for automatic classification of software behavior. *ACM SIGSOFT Software Engineering Notes 29*, 4 (2004), 195–205.

[93] Brackenbury, W., Deora, A., Ritchey, J., Vallee, J., He, W., Wang, G., Littman, M. L., and Ur, B. How users interpret bugs in trigger-action programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (2019), pp. 1–12.

[94] Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (2000), pp. 93–104.

[95] Brito, A., Xavier, L., Hora, A., and Valente, M. T. Apidiff: Detecting api breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 507–511.

[96] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[97] Brumley, D., Poosankam, P., Song, D., and Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy (S&P)* (2008), IEEE, pp. 143–157.

[98] Bruneton, E., Lenglet, R., and Coupaye, T. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems 30*, 19 (2002).

[99] Busany, N., and Maoz, S. Behavioral log analysis with statistical guarantees. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), IEEE, pp. 877–887.

[100] Calimeri, F., Gebser, M., Maratea, M., and Ricca, F. Design and results of the fifth answer set programming competition. *Artificial Intelligence 231* (2016), 151–181.

[101] Cambronero, J. P., Dang, T. H., Vasilakis, N., Shen, J., Wu, J., and Rinard, M. C. Active learning for software engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (2019), pp. 62–78.

[102] Campos, J., Ge, Y., Albunian, N., Fraser, G., Eler, M., and Arcuri, A. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology (IST) 104* (2018), 207–235.

[103] Cao, Z., Tian, Y., Le, T.-D. B., and Lo, D. Rule-based specification mining leveraging learning to rank. *Automated Software Engineering 25*, 3 (2018), 501–530.

[104] Celik, Z. B., Babun, L., Sikder, A. K., Aksu, H., Tan, G., McDaniel, P., and Uluagac, A. S. Sensitive information tracking in commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 1687–1704.

[105] Celik, Z. B., McDaniel, P., and Tan, G. Soteria: Automated IoT safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 147–158.

[106] Celik, Z. B., Tan, G., and McDaniel, P. D. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In *NDSS 2019*.

[107] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)* (2012), IEEE, pp. 380–394.

[108] Chalin, P. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*. Springer, 2006, pp. 100–113.

[109] Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., and Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018), pp. 2095–2108.

[110] Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R., and Zhang, K. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *NDSS* (2018).

[111] Chen, P., and Chen, H. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)* (2018), IEEE, pp. 711–725.

[112] Chen, T. Y., Kuo, F.-C., Merkel, R. G., and Tse, T. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software 83*, 1 (2010), 60–66.

[113] Chen, W., Zou, X., Li, G., and Qian, Z. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *USENIX Security Symposium (USENIX Security)* (2020), pp. 1093–1110.

[114] Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Jiao, X., and Su, Z. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 1967–1983.

[115] Chen, Y., Santosa, A. E., Yi, A. M., Sharma, A., Sharma, A., and Lo, D. A machine learning approach for vulnerability curation. In *International Conference on Mining Software Repositories (MSR)* (2020), pp. 32–42.

[116] Chi, H., Zeng, Q., Du, X., and Yu, J. Cross-app interference threats in smart homes: Categorization, detection and handling. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2020), IEEE, pp. 411–423.

[117] Chow, C. On optimum recognition error and reject tradeoff. *IEEE Transactions on information theory 16*, 1 (1970), 41–46.

[118] Chow, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 3 (1978), 178–187.

[119] Christakis, M., and Bird, C. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016* (2016), ACM, pp. 332–343.

[120] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM) 50*, 5 (2003), 752–794.

[121] Cohen, H., and Maoz, S. Have we seen enough traces?(t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 93–103.

[122] Cohn, D. A., Ghahramani, Z., and Jordan, M. I. Active learning with statistical models. *Journal of artificial intelligence research 4* (1996), 129–145.

[123] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., and Vigna, G. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)* (2017), pp. 2123–2138.

[124] Cortes, C., DeSalvo, G., and Mohri, M. Learning with rejection. In *International Conference on Algorithmic Learning Theory* (2016), Springer, pp. 67–82.

[125] Dagan, I., and Engelson, S. P. Committee-based sampling for training probabilistic classifiers. In *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 150–157.

[126] Dallmeier, V., Knopp, N., Mallon, C., Hack, S., and Zeller, A. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis* (2010), ACM, pp. 85–96.

[127] Dallmeier, V., Lindig, C., Wasylkowski, A., and Zeller, A. Mining object behavior with adabu. In *Proceedings of the 2006 international workshop on Dynamic systems analysis* (2006), ACM, pp. 17–24.

[128] de Caso, G., Braberman, V., Garbervetsky, D., and Uchitel, S. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering 38*, 1 (2010), 141–162.

[129] De Ruiter, J., and Poll, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 193–206.

[130] Decan, A., Mens, T., and Constantinou, E. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories (MSR)* (2018), pp. 181–191.

[131] Dekel, U., and Herbsleb, J. D. Improving API documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 320–330.

[132] Deng, Y., Yang, C., Wei, A., and Zhang, L. Fuzzing deep-learning libraries via automated relational API inference. In *2022 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)* (2022).

[133] Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A., and van Deursen, A. Botsing, a search-based crash reproduction framework for java. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), IEEE, pp. 1278–1282.

[134] Devroey, X., Panichella, S., and Gambi, A. Java unit testing tool competition: Eighth round. In *IEEE/ACM International Conference on Software Engineering Workshops* (2020), pp. 545–548.

[135] Dietl, W., Dietzel, S., Ernst, M. D., Muşlu, K., and Schiller, T. W. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), pp. 681–690.

[136] Ding, W., and Hu, H. On the safety of IoT device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 832–846.

[137] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. Scaling static analyses at facebook. *Communications of the ACM 62*, 8 (2019), 62–70.

[138] Du, X., Xie, X., Li, Y., Ma, L., Liu, Y., and Zhao, J. DeepStellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)* (2019), pp. 477–487.

[139] Dupont, P., Lambeau, B., Damas, C., and Lamsweerde, A. v. The qsm algorithm and its application to software behavior model induction. *Applied Artificial Intelligence 22*, 1-2 (2008), 77–115.

[140] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M.,

ET AL. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), pp. 475–488.

[141] DWARAKANATH, A., AHUJA, M., SIKAND, S., RAO, R. M., BOSE, R. J. C., DUBASH, N., AND PODDER, S. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)* (2018), pp. 118–128.

[142] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)* (1999), IEEE, pp. 411–420.

[143] DYER, R., NGUYEN, H. A., RAJAN, H., AND NGUYEN, T. N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 422–431.

[144] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 73–84.

[145] ELBAUM, S., CHIN, H. N., DWYER, M. B., AND DOKULIL, J. Carving differential unit test cases from system test cases. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2006), pp. 253–264.

[146] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The daikon system for dynamic detection of likely invariants. *Science of computer programming 69*, 1-3 (2007), 35–45.

[147] ERTEKIN, S., HUANG, J., BOTTOU, L., AND GILES, L. Learning on the border: active learning in imbalanced data classification. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management* (2007), pp. 127–136.

[148] FEATHERS, M. *Working Effectively With Legacy Code: Work Effect Leg Code _p1*. Prentice Hall Professional, 2004.

[149] Feldt, R., Poulding, S., Clark, D., and Yoo, S. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2016), IEEE, pp. 223–233.

[150] Feldt, R., Torkar, R., Gorschek, T., and Afzal, W. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (2008), IEEE, pp. 178–186.

[151] Fernandes, E., Jung, J., and Prakash, A. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)* (2016), IEEE, pp. 636–654.

[152] Fernandes, E., Rahmati, A., Jung, J., and Prakash, A. Decentralized action integrity for trigger-action IoT platforms. In *Proceedings 2018 Network and Distributed System Security Symposium* (2018).

[153] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)* (2002), pp. 234–245.

[154] Fleiss, J. L. Measuring nominal scale agreement among many raters. *Psychological bulletin 76*, 5 (1971), 378.

[155] Foo, D., Chua, H., Yeo, J., Ang, M. Y., and Sharma, A. Efficient static checking of library updates. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2018), pp. 791–796.

[156] Foo, D., Yeo, J., Xiao, H., and Sharma, A. The dynamics of Software Composition Analysis. *Automated Software Engineering (ASE) (Late Breaking Results)* (2019).

[157] Fowkes, J., and Sutton, C. Parameter-free probabilistic API mining across github. In *2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2016), pp. 254–265.

[158] Fraser, G., and Arcuri, A. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM*

SIGSOFT symposium and the 13th European conference on Foundations of software engineering (2011), ACM, pp. 416–419.

[159] Fraser, G., and Zeller, A. Exploiting common object usage in test case generation. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (2011), IEEE, pp. 80–89.

[160] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[161] Fu, W., and Menzies, T. Easy over hard: A case study on deep learning. In *2017 11th joint meeting on foundations of software engineering (FSE)* (2017), pp. 49–60.

[162] Fujii, A., Tokunaga, T., Inui, K., and Tanaka, H. Selective sampling for example-based word sense disambiguation. *Computational linguistics-Association for Computational Linguistics 24*, 4 (1998), 573–597.

[163] Gabbay, D., Pnueli, A., Shelah, S., and Stavi, J. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1980), pp. 163–173.

[164] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., and Chen, Z. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (S&P)* (2018), IEEE, pp. 679–696.

[165] Gao, J., Kong, P., Li, L., Bissyandé, T. F., and Klein, J. Negative results on mining crypto-API usage rules in Android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 388–398.

[166] Gardner, A. B., Krieger, A. M., Vachtsevanos, G., and Litt, B. One-class novelty detection for seizure analysis from intracranial eeg. *Journal of Machine Learning Research 7*, Jun (2006), 1025–1044.

[167] Gay, G. The fitness function for the job: Search-based generation of test suites that detect real faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017), IEEE, pp. 345–355.

[168] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. A user's guide to gringo, clasp, clingo, and iclingo.

[169] Gharehyazie, M., Ray, B., and Filkov, V. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), IEEE, pp. 291–301.

[170] Goffi, A., Gorla, A., Ernst, M. D., and Pezzè, M. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th international symposium on software testing and analysis* (2016), pp. 213–224.

[171] Groce, A., Peled, D., and Yannakakis, M. Adaptive model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2002), Springer, pp. 357–370.

[172] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), pp. 78–88.

[173] Gros, D., Sezhiyan, H., Devanbu, P., and Yu, Z. Code to comment "translation": Data, metrics, baselining & evaluation. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)* (2020), IEEE, pp. 746–757.

[174] Grove, D., and Chambers, C. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23*, 6 (2001), 685–746.

[175] Gu, X., Zhang, H., and Kim, S. Codekernel: A graph kernel based approach to the selection of API usage examples. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), IEEE, pp. 590–601.

[176] Gu, X., Zhang, H., Zhang, D., and Kim, S. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2016), pp. 631–642.

[177] Guo, J., Jiang, Y., Zhao, Y., Chen, Q., and Sun, J. DLFuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018) (2018), pp. 739–743.

[178] Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y., Li, X., and Shen, C. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)* (2020), IEEE, pp. 486–498.

[179] Habib, A., and Pradel, M. How many of all bugs do we find? a study of static bug detectors. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)* (2018), IEEE, pp. 317–328.

[180] Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using NetworkX. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[181] Hanam, Q., Tan, L., Holmes, R., and Lam, P. Finding patterns in static analysis alerts: improving actionable alert ranking. In *11th Working Conference on mining software repositories (MSR 2014)* (2014), pp. 152–161.

[182] Haryono, S. A., Thung, F., Kang, H. J., Serrano, L., Muller, G., Lawall, J., Lo, D., and Jiang, L. Automatic android deprecated-api usage update by learning from single updated example. In *Proceedings of the 28th international conference on program comprehension* (2020), pp. 401–405.

[183] He, P., Meister, C., and Su, Z. Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)* (2021), IEEE, pp. 410–422.

[184] He, W., Martinez, J., Padhi, R., Zhang, L., and Ur, B. When smart devices are stupid: negative experiences using home smart devices. In *2019 IEEE Security and Privacy Workshops (SPW)* (2019), IEEE, pp. 150–155.

[185] He, X., Xie, X., Li, Y., Sun, J., Li, F., Zou, W., Liu, Y., Yu, L., Zhou, J., Shi, W., et al. SoFi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CSS 2021)* (2021), pp. 2229–2242.

[186] Heckman, S., and Williams, L. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2008)* (2008), pp. 41–50.

[187] Heckman, S., and Williams, L. A model building process for identifying actionable static analysis alerts. In *International Conference on Software Testing Verification and Validation (ICST 2009)* (2009), IEEE, pp. 161–170.

[188] Heckman, S., and Williams, L. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology (IST) 53*, 4 (2011), 363–387.

[189] Heckman, S., and Williams, L. A comparative evaluation of static analysis actionable alert identification techniques. In *9th International Conference on Predictive Models in Software Engineering (PROMISE 2013)* (2013), pp. 1–10.

[190] Heelan, S., Melham, T., and Kroening, D. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2019), pp. 1689–1706.

[191] Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)* (2018), pp. 152–162.

[192] Hellendoorn, V. J., and Devanbu, P. Are deep neural networks the best choice for modeling source code? In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)* (2017), pp. 763–773.

[193] Herbei, R., and Wegkamp, M. H. Classification with reject option. *The Canadian Journal of Statistics/La Revue Canadienne de Statistique* (2006), 709–721.

[194] Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 445–458.

[195] Hoschele, M., and Zeller, A. Mining input grammars with autogram. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (2017), IEEE, pp. 31–34.

[196] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *ACM SIGPLAN notices 39*, 12 (2004), 92–106.

[197] Hu, H., Chua, Z. L., Adrian, S., Saxena, P., and Liang, Z. Automatic generation of data-oriented exploits. In *USENIX Security Symposium (USENIX Security)* (2015), pp. 177–192.

[198] Huang, J., and Cakmak, M. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2015), pp. 215–225.

[199] Huang, S.-J., Jin, R., and Zhou, Z.-H. Active learning by querying informative and representative examples. In *Advances in neural information processing systems* (2010), pp. 892–900.

[200] Huang, W., and Milanova, A. Reiminfer: method purity inference for java. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 38.

[201] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[202] Huth, M., and Ryan, M. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[203] Iannone, E., Di Nucci, D., Sabetta, A., and De Lucia, A. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *IEEE/ACM International Conference on Program Comprehension (ICPC)* (2021), IEEE, pp. 396–400.

[204] Imtiaz, N., Khanom, A., and Williams, L. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering* (2022).

[205] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019) (2019), pp. 510–520.

[206] Ispoglou, K., Austin, D., Mohan, V., and Payer, M. Fuzzgen: Automatic fuzzer generation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (2020), pp. 2271–2287.

[207] Jackson, D. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[208] Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., and Papernot, N. High accuracy and high fidelity extraction of neural networks. In *29th USENIX security symposium (USENIX Security 20)* (2020), pp. 1345–1362.

[209] Jamrozik, K., von Styp-Rekowsky, P., and Zeller, A. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering* (2016), pp. 37–48.

[210] Jia, L., Zhong, H., Wang, X., Huang, L., and Lu, X. An empirical study on bugs inside TensorFlow. In *International Conference on Database Systems for Advanced Applications* (2020), Springer, pp. 604–620.

[211] Jia, L., Zhong, H., Wang, X., Huang, L., and Lu, X. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software (JSS) 177* (2021), 110935.

[212] Jia, Y. J., Chen, Q. A., Wang, S., Rahmati, A., Fernandes, E., Mao, Z. M., Prakash, A., and Unviersity, S. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *NDSS* (2017).

[213] Jiang, J., Xu, H., and Zhou, Y. RULF: Rust library fuzzing via api dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)* (2021), IEEE, pp. 581–592.

[214] Jin, D., Meredith, P. O., Lee, C., and Roşu, G. JavaMOP: Efficient parametric runtime monitoring framework. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 1427–1430.

[215] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 672–681.

[216] Johnson, B., Song, Y., Murphy-Hill, E. R., and Bowdidge, R. W. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering, (ICSE 2013)* (2013), IEEE Computer Society, pp. 672–681.

[217] Juuti, M., Szyller, S., Marchal, S., and Asokan, N. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019), IEEE, pp. 512–527.

[218] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. The promises and perils of mining github. In *11th working conference on Mining Software Repositories (MSR 2014)* (2014), pp. 92–101.

[219] Kampmann, A., Havrikov, N., Soremekun, E. O., and Zeller, A. When does my program do this? learning circumstances of software behavior. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2020), pp. 1228–1239.

[220] Kang, H. J., Aw, K. L., and Lo, D. Detecting false alarms from automatic static analysis tools: How far are we? In *Proceedings of the IEEE/ACM International Conference on Software Engineering 2022* (2022).

[221] Kang, H. J., Bissyandé, T. F., and Lo, D. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), IEEE, pp. 1–12.

[222] Kang, H. J., and Lo, D. Active learning of discriminative subgraph patterns for api misuse detection. *IEEE Transactions on Software Engineering* (2021).

[223] Kang, H. J., and Lo, D. Adversarial specification mining. *ACM Transactions on Software Engineering and Methodology (TOSEM) 30*, 2 (2021), 1–40.

[224] Kang, H. J., Nguyen, T. G., Le, B., Păsăreanu, C. S., and Lo, D. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022), pp. 276–288.

[225] Kang, H. J., Sim, S. Q., and Lo, D. Iotbox: Sandbox mining to prevent interaction threats in iot systems. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (2021), IEEE, pp. 182–193.

[226] Kang, H. J., Thung, F., Lawall, J., Muller, G., Jiang, L., and Lo, D. Semantic patches for java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)* (2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[227] Kaufman, S., Rosset, S., Perlich, C., and Stitelman, O. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD) 6*, 4 (2012), 1–21.

[228] Kazerounian, M., Foster, J. S., and Min, B. Simtyper: sound type inference for ruby using type equality prediction. *Proceedings of the ACM on Programming Languages 5*, OOPSLA (2021), 1–27.

[229] Kechagia, M., Devroey, X., Panichella, A., Gousios, G., and van Deursen, A. Effective and efficient API misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 192–203.

[230] Khan, S. S., and Madden, M. G. One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review 29*, 3 (2014), 345–374.

[231] Kharkar, A., Moghaddam, R. Z., Jin, M., Liu, X., Shi, X., Clement, C., and Sundaresan, N. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the IEEE/ACM International Conference on Software Engineering 2022* (2022).

[232] Kim, S., and Ernst, M. D. Prioritizing warning categories by analyzing software history. In *4th International Workshop on Mining Software Repositories (MSR 07: ICSE Workshops 2007)* (2007), IEEE, pp. 27–27.

[233] Kim, S., Whitehead, E. J., and Zhang, Y. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering (TSE) 34*, 2 (2008), 181–196.

[234] Koc, U., Wei, S., Foster, J. S., Carpuat, M., and Porter, A. A. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In *12th IEEE conference on software testing, validation and verification (ICST 2019)* (2019), IEEE, pp. 288–299.

[235] Kochhar, P. S., Bissyandé, T. F., Lo, D., and Jiang, L. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *2013 17th european conference on software maintenance and reengineering* (2013), IEEE, pp. 353–356.

[236] Kochhar, P. S., Thung, F., and Lo, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE International conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015), IEEE, pp. 560–564.

[237] Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., and Lo, D. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), IEEE, pp. 1–10.

[238] Kochhar, P. S., Tian, Y., and Lo, D. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 803–814.

[239] Kochhar, P. S., Xia, X., Lo, D., and Li, S. Practitioners' expectations on automated fault localization. In *25th International Symposium on Software Testing and Analysis* (2016), pp. 165–176.

[240] Kong, X., Fan, W., and Yu, P. S. Dual active feature and sample selection for graph classification. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (2011), pp. 654–662.

[241] Kremenek, T., Ashcraft, K., Yang, J., and Engler, D. Correlation exploitation in error ranking. *ACM SIGSOFT Software Engineering Notes 29*, 6 (2004), 83–93.

[242] Krka, I., Brun, Y., and Medvidovic, N. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 178–189.

[243] Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering* (2019).

[244] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. Do developers update their library dependencies? *Empirical Software Engineering 23*, 1 (2018), 384–417.

[245] Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., and Smaragdakis, Y. Static analysis of shape in TensorFlow programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)* (2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[246] Lamothe, M., and Shang, W. When APIs are intentionally bypassed: An exploratory study of API workarounds. In *42nd International Conference on Software Engineering (ICSE)* (2020), vol. 2020.

[247] Lamothe, M., Shang, W., and Chen, T.-H. P. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* (2020).

[248] Landis, J. R., and Koch, G. G. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[249] Le, T.-D. B., Bao, L., Lo, D., Gao, D., and Li, L. Towards mining comprehensive Android sandboxes. In *2018 23rd International conference on engineering of complex computer systems (ICECCS)* (2018), IEEE, pp. 51–60.

[250] Le, T.-D. B., Le, X.-B. D., Lo, D., and Beschastnikh, I. Synergizing specification miners through model fissions and fusions (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 115–125.

[251] Le, T.-D. B., and Lo, D. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015), IEEE, pp. 331–340.

[252] Le, T.-D. B., and Lo, D. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), ACM, pp. 106–117.

[253] Le, X.-B. D., Pasareanu, C., Padhye, R., Lo, D., Visser, W., and Sen, K. SAFFRON: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes 44*, 4 (2019), 14–14.

[254] Le-Cong, T., Kang, H. J., Nguyen, T. G., Haryono, S. A., Lo, D., Le, X.-B. D., and Huynh, Q. T. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 520–532.

[255] Legunsen, O., Hassan, W. U., Xu, X., Roşu, G., and Marinov, D. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2016), IEEE, pp. 602–613.

[256] Lemieux, C., Padhye, R., Sen, K., and Song, D. PerfFuzz: Automatically generating pathological inputs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2018), pp. 254–265.

[257] Lemieux, C., Park, D., and Beschastnikh, I. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 81–92.

[258] Leon, D., and Podgurski, A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* (2003), IEEE, pp. 442–453.

[259] Lewis, D. D., and Catlett, J. Heterogeneous uncertainty sampling for supervised learning. In *Machine learning proceedings 1994*. Elsevier, 1994, pp. 148–156.

[260] Lewis, D. D., and Gale, W. A. A sequential algorithm for training text classifiers. In *SIGIR'94* (1994), Springer, pp. 3–12.

[261] Li, L., Gao, J., Bissyandé, T. F., Ma, L., Xia, X., and Klein, J. Characterising deprecated android apis. In *Proceedings of the*

15th International Conference on Mining Software Repositories (2018), pp. 254–264.

[262] Li, Z., and Zhou, Y. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes 30*, 5 (2005), 306–315.

[263] Liang, G., Wu, L., Wu, Q., Wang, Q., Xie, T., and Mei, H. Automatic construction of an effective training set for prioritizing static analysis warnings. In *IEEE/ACM international conference on Automated Software Engineering (ASE 2010)* (2010), pp. 93–102.

[264] Lin, B., Wang, S., Liu, K., Mao, X., and Bissyandé, T. F. Automated comment update: How far are we? In *IEEE/ACM 29th International Conference on Program Comprehension (ICPC 2021)* (2021), IEEE, pp. 36–46.

[265] Lin, B., Zampetti, F., Bavota, G., Di Penta, M., Lanza, M., and Oliveto, R. Sentiment analysis for software engineering: How far can we go? In *40th International Conference on Software Engineering (ICSE 2018)* (2018), pp. 94–104.

[266] Lin, Z., Zhong, H., Chen, Y., and Zhao, J. Lockpeeker: detecting latent locks in Java APIs. In *Proc. ASE* (2016), ACM, pp. 368–378.

[267] Liu, B., Zhang, C., Gong, G., Zeng, Y., Ruan, H., and Zhuge, J. FANS: Fuzzing Android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 307–323.

[268] Liu, C., Lu, J., Li, G., Yuan, T., Li, L., Tan, F., Yang, J., You, L., and Xue, J. Detecting TensorFlow program bugs in real-world industrial environment. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)* (2021), IEEE, pp. 55–66.

[269] Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., and Chen, W. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804* (2021).

[270] Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., and Le Traon, Y. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* (2018).

[271] Liu, K., Koyuncu, A., Kim, D., and Bissyandé, T. F. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)* (2019), IEEE, pp. 1–12.

[272] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[273] Liu, Z., Xia, X., Hassan, A. E., Lo, D., Xing, Z., and Wang, X. Neural-machine-translation-based commit message generation: how far are we? In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)* (2018), pp. 373–384.

[274] Lo, D., Khoo, S.-C., and Liu, C. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice 20*, 4 (2008), 227–247.

[275] Lo, D., Mariani, L., and Pezzè, M. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2009), ACM, pp. 345–354.

[276] Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 1–28.

[277] Lorenzoli, D., Mariani, L., and Pezzè, M. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering* (2008), ACM, pp. 501–510.

[278] Lu, H., and Cukic, B. An adaptive approach with active learning in software fault prediction. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering* (2012), pp. 79–88.

[279] Malik, R. S., Patra, J., and Pradel, M. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 304–315.

[280] Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *IEEE/ACM 27th International Conference on Program Comprehension (ICPC 2019)* (2019), IEEE, pp. 209–219.

[281] Mashhadi, E., and Hemmati, H. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (2021), IEEE, pp. 505–509.

[282] Mathis, B., Gopinath, R., and Zeller, A. Learning input tokens for effective fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2020), pp. 27–37.

[283] McMinn, P. Search-based software test data generation: a survey. *Software testing, Verification and reliability 14*, 2 (2004), 105–156.

[284] Mezzetti, G., Møller, A., and Torp, M. T. Type regression testing to detect breaking changes in node. js libraries. In *32nd european conference on object-oriented programming (ECOOP 2018)* (2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[285] Miller, B., Zhang, M., and Heymann, E. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering (TSE)* (2020).

[286] Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. *Communications of the ACM 33*, 12 (1990), 32–44.

[287] Miranda, B., Cruciani, E., Verdecchia, R., and Bertolino, A. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering* (2018), ACM, pp. 222–232.

[288] Mirhosseini, S., and Parnin, C. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), IEEE, pp. 84–94.

[289] Molina, U. R., Kifetew, F., and Panichella, A. Java unit testing tool competition-sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)* (2018), IEEE, pp. 22–29.

[290] Møller, A., Nielsen, B. B., and Torp, M. T. Detecting locations in javascript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages 4*, OOPSLA (2020), 1–25.

[291] Monperrus, M., Bruch, M., and Mezini, M. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming* (2010), Springer, pp. 2–25.

[292] Mora, F., Li, Y., Rubin, J., and Chechik, M. Client-specific equivalence checking. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018), IEEE, pp. 441–451.

[293] Mujahid, S., Abdalkareem, R., Shihab, E., and McIntosh, S. Using others' tests to identify breaking updates. In *International Conference on Mining Software Repositories (MSR)* (2020), pp. 466–476.

[294] Nachtigall, M., Schlichtig, M., and Bodden, E. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022), pp. 532–543.

[295] Nadi, S., Krüger, S., Mezini, M., and Bodden, E. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (2016), pp. 935–946.

[296] Nam, D., Horvath, A., Macvean, A., Myers, B., and Vasilescu, B. Marble: Mining for boilerplate code to identify api usability problems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), IEEE, pp. 615–627.

[297] Nam, J., Wang, S., Xi, Y., and Tan, L. Designing bug detection rules for fewer false alarms. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (2018), pp. 315–316.

[298] Nam, J., Wang, S., Xi, Y., and Tan, L. A bug finder refined by a large set of open-source projects. *Information and Software Technology 112* (2019), 164–175.

[299] Naumann, D. A. Observational purity and encapsulation. *Theoretical Computer Science 376*, 3 (2007), 205–224.

[300] Nguyen, A. T., Hilton, M., Codoban, M., Nguyen, H. A., Mast, L., Rademacher, E., Nguyen, T. N., and Dig, D. API code recommendation using statistical learning from fine-grained changes. In *2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2016), pp. 511–522.

[301] Nguyen, D. T., Song, C., Qian, Z., Krishnamurthy, S. V., Colbert, E. J., and McDaniel, P. IoTSan: Fortifying the safety of IoT systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies* (2018), pp. 191–203.

[302] Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., and Di Penta, M. Focus: A recommender system for mining API function calls and usage patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 1050–1060.

[303] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE)* (2009), pp. 383–392.

[304] Nguyen-Truong, G., Kang, H. J., Lo, D., Sharma, A., Santosa, A. E., Sharma, A., and Ang, M. Y. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2022), IEEE, pp. 51–62.

[305] Nielsen, B. B., Torp, M. T., and Møller, A. Semantic patches for adaptation of javascript programs to evolving libraries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), IEEE, pp. 74–85.

[306] Pacheco, C., and Ernst, M. D. Randoop: feedback-directed random testing for java. In *OOPSLA Companion* (2007), pp. 815–816.

[307] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 75–84.

[308] Padhye, R., Lemieux, C., Sen, K., Papadakis, M., and Le Traon, Y. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 329–340.

[309] Pailoor, S., Aday, A., and Jana, S. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 729–743.

[310] Panichella, A., Kifetew, F. M., and Tonella, P. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), IEEE, pp. 1–10.

[311] Panichella, A., Kifetew, F. M., and Tonella, P. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering 44*, 2 (2017), 122–158.

[312] Panichella, S., Arnaoudova, V., Di Penta, M., and Antoniol, G. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)* (2015), IEEE, pp. 161–170.

[313] Papi, M. M., Ali, M., Correa Jr, T. L., Perkins, J. H., and Ernst, M. D. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis* (2008), pp. 201–212.

[314] Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., and Massacci, F. Vulnerable open source dependencies: Counting those that matter. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2018), pp. 1–10.

[315] Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., and Massacci, F. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering (TSE)* (2020).

[316] Pavese, E., Soremekun, E. O., Havrikov, N., Grunske, L., and Zeller, A. Inputs from hell: Generating uncommon inputs from common samples. *CoRR abs/1812.07525* (2018).

[317] Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering 16*, 11 (2004), 1424–1440.

[318] Peled, D., Vardi, M. Y., and Yannakakis, M. Black box checking. In *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, 1999, pp. 225–240.

[319] Peng, H., Shoshitaishvili, Y., and Payer, M. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (S&P)* (2018), IEEE, pp. 697–710.

[320] Peng, Y., Gao, C., Li, Z., Gao, B., Lo, D., Zhang, Q., and Lyu, M. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 2019–2030.

[321] Perera, P., Nallapati, R., and Xiang, B. OCGAN: One-class novelty detection using gans with constrained latent representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), pp. 2898–2906.

[322] Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017), pp. 2155–2168.

[323] Pham, H. V., Lutellier, T., Qi, W., and Tan, L. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 1027–1038.

[324] Pham, L. H., Thi, L. L. T., and Sun, J. Assertion generation through active learning. In *International Conference on Formal Engineering Methods* (2017), Springer, pp. 174–191.

[325] Pham, V.-T., Böhme, M., and Roychoudhury, A. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)* (2020).

[326] Pham, V.-T., Böhme, M., Santosa, A. E., Căciulescu, A. R., and Roychoudhury, A. Smart greybox fuzzing. *IEEE Transactions on Software Engineering 47*, 9 (2019), 1980–1997.

[327] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science* (1977), IEEE, pp. 46–57.

[328] Ponta, S. E., Plate, H., and Sabetta, A. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2018), IEEE, pp. 449–460.

[329] Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., and Dangremont, C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 383–387.

[330] Pradel, M., Gousios, G., Liu, J., and Chandra, S. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 209–220.

[331] Pradel, M., Jaspan, C., Aldrich, J., and Gross, T. R. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 925–935.

[332] Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., and Lo, D. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering 26*, 4 (2021), 1–34.

[333] Provos, N. Improving host security with system call policies.

[334] Rabin, M. R. I., Bui, N. D., Wang, K., Yu, Y., Jiang, L., and Alipour, M. A. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology (IST) 135* (2021), 106552.

[335] Radhakrishna, A., Lewchenko, N. V., Meier, S., Mover, S., Sripada, K. C., Zufferey, D., Chang, B.-Y. E., and Cernỳ, P. Droidstar: callback typestates for android classes. In *2018 IEEE/ACM*

40th International Conference on Software Engineering (ICSE) (2018), IEEE, pp. 1160–1170.

[336] Rahman, F., Posnett, D., and Devanbu, P. Recalling the "imprecision" of cross-project defect prediction. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)* (2012), pp. 1–11.

[337] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 861–875.

[338] Reddy, S., Lemieux, C., Padhye, R., and Sen, K. Quickly generating diverse valid test inputs with reinforcement learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), IEEE, pp. 1410–1421.

[339] Reinhardt, A., Zhang, T., Mathur, M., and Kim, M. Augmenting stack overflow with API usage patterns mined from GitHub. In *2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 880–883.

[340] Rennie, J. D., Shih, L., Teevan, J., and Karger, D. R. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the 20th international conference on machine learning (ICML)* (2003), pp. 616–623.

[341] Ribeiro, M. T., Singh, S., and Guestrin, C. "why should I trust you?" explaining the predictions of any classifier. In *22nd ACM SIGKDD international conference on Knowledge Discovery and Data mining* (2016), pp. 1135–1144.

[342] Rieck, K., Trinius, P., Willems, C., and Holz, T. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security 19*, 4 (2011), 639–668.

[343] Robbes, R., and Lungu, M. A study of ripple effects in software ecosystems (NIER track). In *33rd International Conference on Software Engineering* (2011), pp. 904–907.

[344] Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T. Automated API property inference techniques. *IEEE Transactions on Software Engineering 39*, 5 (2012), 613–637.

[345] Rojas, J. M., Campos, J., Vivanti, M., Fraser, G., and Arcuri, A. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering* (2015), Springer, pp. 93–108.

[346] Rojas, J. M., Fraser, G., and Arcuri, A. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability 26*, 5 (2016), 366–401.

[347] Ronen, E., and Shamir, A. Extended functionality attacks on IoT devices: The case of smart lights. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 3–12.

[348] Rountev, A., Kagan, S., and Gibas, M. Static and dynamic analysis of call chains in Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2004), pp. 1–11.

[349] Roy, C. K., and Cordy, J. R. Benchmarks for software clone detection: A ten-year retrospective. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)* (2018), IEEE, pp. 26–37.

[350] Rutar, N., Almazan, C. B., and Foster, J. S. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004)* (2004), IEEE, pp. 245–256.

[351] Ruthruff, J., Penix, J., Morgenthaler, J., Elbaum, S., and Rothermel, G. Predicting accurate and actionable static analysis warnings. In *30th ACM/IEEE International Conference on Software Engineering (ICSE 2008)* (2008), IEEE, pp. 341–350.

[352] Sabetta, A., and Bezzi, M. A practical approach to the automatic classification of security-relevant commits. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2018), IEEE, pp. 579–582.

[353] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at google. *Communications of the ACM 61*, 4 (2018), 58–66.

[354] Sadowski, C., Van Gogh, J., Jaspan, C., Soderberg, E., and Winter, C. Tricorder: Building a program analysis ecosystem. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)* (2015), vol. 1, IEEE, pp. 598–608.

[355] Safyallah, H., and Sartipi, K. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension (ICPC'06)* (2006), IEEE, pp. 84–88.

[356] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (2016), pp. 1157–1168.

[357] Sato, T., Shen, J., Wang, N., Jia, Y., Lin, X., and Chen, Q. A. Dirty road can attack: Security of deep learning based automated lane centering under {Physical-World} attack. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 3309–3326.

[358] Schiller, T. W., Donohue, K., Coward, F., and Ernst, M. D. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)* (2014), pp. 596–607.

[359] Schumi, R., and Sun, J. ExAIS: Executable ai semantics. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2022)* (2022).

[360] Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. In *54th Annual Meeting of the Association for Computational Linguistics* (2016), Association for Computational Linguistics (ACL), pp. 1715–1725.

[361] Settles, B. Active learning literature survey. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 2009.

[362] Settles, B., and Craven, M. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing* (2008), pp. 1070–1079.

[363] Settles, B., Craven, M., and Ray, S. Multiple-instance active learning. *Advances in neural information processing systems 20* (2007).

[364] Shen, H., Fang, J., and Zhao, J. Efindbugs: Effective error ranking for findbugs. In *4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)* (2011), IEEE, pp. 299–308.

[365] Shen, J., and Rinard, M. C. Active learning for inference and regeneration of applications that access databases. *ACM Transactions on Programming Languages and Systems (TOPLAS) 42*, 4 (2021), 1–119.

[366] Shen, Z., Roongta, R., and Dolan-Gavitt, B. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 1275–1290.

[367] Shin, D., Yoo, S., and Bae, D.-H. Diversity-aware mutation adequacy criterion for improving fault detection capability. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2016), IEEE, pp. 122–131.

[368] Sim, J., and Wright, C. C. The kappa statistic in reliability studies: use, interpretation, and sample size requirements. *Physical therapy 85*, 3 (2005), 257–268.

[369] Sim, S. E., Easterbrook, S., and Holt, R. C. Using benchmarking to advance research: A challenge to software engineering. In *25th International Conference on Software Engineering (ICSE 2003)* (2003), IEEE, pp. 74–83.

[370] Soltani, M., Panichella, A., and Van Deursen, A. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering (TSE) 46*, 12 (2018), 1294–1317.

[371] Soremekun, E., Udeshi, S. S., and Chattopadhyay, S. Astraea: Grammar-based fairness testing. *IEEE Transactions on Software Engineering (TSE)* (2022).

[372] Sun, P., Brown, C., Beschastnikh, I., and Stolee, K. T. Mining specifications from documentation using a crowd. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), IEEE, pp. 275–286.

[373] Surbatovich, M., Aljuraidan, J., Bauer, L., Das, A., and Jia, L. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *Proceedings of the*

26th International Conference on World Wide Web (2017), pp. 1501–1510.

[374] Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 1433–1443.

[375] Tahaei, M., Vaniea, K., Beznosov, K., and Wolters, M. K. Security notifications in static analysis tools: Developers' attitudes, comprehension, and ability to act on them. In *2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–17.

[376] Takashima, Y., Martins, R., Jia, L., and Păsăreanu, C. S. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)* (2021), pp. 899–913.

[377] Tan, S. H., Marinov, D., Tan, L., and Leavens, G. T. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (2012), IEEE, pp. 260–269.

[378] Tax, D. M., and Duin, R. P. Growing a multi-class classifier with a reject option. *Pattern Recognition Letters 29*, 10 (2008), 1565–1570.

[379] Thoma, M., Cheng, H., Gretton, A., Han, J., Kriegel, H.-P., Smola, A., Song, L., Yu, P. S., Yan, X., and Borgwardt, K. Near-optimal supervised feature selection among frequent subgraphs. In *Proceedings of the 2009 SIAM International Conference on Data Mining* (2009), SIAM, pp. 1076–1087.

[380] Thummalapenta, S., and Xie, T. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 496–506.

[381] Thummalapenta, S., and Xie, T. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering 18*, 3-4 (2011), 293–323.

[382] Thung, F., Le, X.-B. D., and Lo, D. Active semi-supervised defect categorization. In *2015 IEEE 23rd International Conference on Program Comprehension* (2015), IEEE, pp. 60–70.

[383] Thung, F., Lo, D., Jiang, L., Rahman, F., Devanbu, P. T., et al. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)* (2012), IEEE, pp. 50–59.

[384] Tómasdóttir, K. F., Aniche, M., and van Deursen, A. Why and how javascript developers use linters. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)* (2017), IEEE, pp. 578–589.

[385] Tomassi, D. A., and Rubio-González, C. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 292–303.

[386] Trimananda, R., Aqajari, S. A. H., Chuang, J., Demsky, B., Xu, G. H., and Lu, S. Understanding and automatically detecting conflicting interactions between smart home IoT applications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020).

[387] Tu, F., Zhu, J., Zheng, Q., and Zhou, M. Be careful of when: an empirical study on time-related misuse of issue tracking data. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)* (2018), pp. 307–318.

[388] Uddin, G., and Robillard, M. P. How api documentation fails. *Ieee software 32*, 4 (2015), 68–75.

[389] Vasilakis, N., Benetopoulos, A., Handa, S., Schoen, A., Shen, J., and Rinard, M. C. Supply-chain vulnerability elimination via active learning and regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS 2021)* (2021), pp. 1755–1770.

[390] Vasilevskii, M. Failure diagnosis of automata. *Cybernetics 9*, 4 (1973), 653–665.

[391] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., and Zaidman, A. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering (EMSE) 25*, 2 (2020), 1419–1457.

[392] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).

[393] Vendome, C., Linares-Vásquez, M., Bavota, G., Di Penta, M., German, D., and Poshyvanyk, D. License usage and changes: a large-scale study of java projects on github. In *2015 IEEE 23rd International Conference on Program Comprehension* (2015), IEEE, pp. 218–228.

[394] Visser, W., Păsăreanu, C. S., and Khurshid, S. Test input generation with Java PathFinder. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2004), pp. 97–107.

[395] Vu, D.-L., Massacci, F., Pashchenko, I., Plate, H., and Sabetta, A. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 780–792.

[396] Vu, D. L., Pashchenko, I., Massacci, F., Plate, H., and Sabetta, A. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020), pp. 2093–2095.

[397] Walkinshaw, N., Bogdanov, K., Holcombe, M., and Salahuddin, S. Reverse engineering state machines by interactive grammar inference. In *14th Working Conference on Reverse Engineering (WCRE 2007)* (2007), IEEE, pp. 209–218.

[398] Wan, Z., Lo, D., Xia, X., Cai, L., and Li, S. Mining sandboxes for Linux containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017), IEEE, pp. 92–102.

[399] Wang, D., Zhang, Z., Zhang, H., Qian, Z., Krishnamurthy, S. V., and Abu-Ghazaleh, N. SyzVegas: Beating kernel fuzzing

odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2741–2758.

[400] Wang, J., Chen, B., Wei, L., and Liu, Y. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 579–594.

[401] Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., and Zhang, D. Mining succinct and high-coverage API usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)* (2013), IEEE, pp. 319–328.

[402] Wang, J., Huang, Y., Wang, S., and Wang, Q. Find bugs in static bug finders. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)* (2022), IEEE, pp. 516–527.

[403] Wang, J., Lutellier, T., Qian, S., Pham, H. V., and Tan, L. Eagle: Creating equivalent graphs to test deep learning libraries. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2022)* (2022).

[404] Wang, J., Wang, S., and Wang, Q. Is there a "golden" feature set for static warning identification?: an experimental evaluation. In *12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, (ESEM 2018)* (2018), ACM, pp. 17:1–17:10.

[405] Wang, J., Wang, S., and Wang, Q. Is there a" golden" feature set for static warning identification? an experimental evaluation. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement* (2018), pp. 1–10.

[406] Wang, Q., Datta, P., Yang, W., Liu, S., Bates, A., and Gunter, C. A. Charting the attack surface of trigger-action IoT platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 1439–1453.

[407] Wang, Q., Hassan, W. U., Bates, A., and Gunter, C. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium* (2018).

[408] Wang, T., Wei, T., Gu, G., and Zou, W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy (S&P* (2010), IEEE, pp. 497–512.

[409] Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y., and Liu, Y. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2020), IEEE, pp. 35–45.

[410] Wang, Y., Wu, Z., Wei, Q., and Wang, Q. NeuFuzz: Efficient fuzzing with deep neural network. *IEEE Access 7* (2019), 36340–36352.

[411] Wang, Y., Zhang, C., Zhao, Z., Zhang, B., Gong, X., and Zou, W. MAZE: Towards automated heap feng shui. In *USENIX Security Symposium (USENIX Security)* (2021).

[412] Wang, Z., Yan, M., Chen, J., Liu, S., and Zhang, D. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 788–799.

[413] Wasylkowski, A., and Zeller, A. Mining temporal specifications from object usage. *Automated Software Engineering 18*, 3-4 (2011), 263–292.

[414] Wasylkowski, A., Zeller, A., and Lindig, C. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE* (2007), pp. 35–44.

[415] Wei, A., Deng, Y., Yang, C., and Zhang, L. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2022)* (2022).

[416] Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.-C., and Su, Z. Exposing library API misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 866–877.

[417] Williams, C. C., and Hollingsworth, J. K. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering (TSE) 31*, 6 (2005), 466–480.

[418] Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., and Zou, W. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *USENIX Security Symposium (USENIX Security)* (2018), pp. 781–797.

[419] Xavier, L., Brito, A., Hora, A., and Valente, M. T. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2017), IEEE, pp. 138–147.

[420] Xia, C. S., and Zhang, L. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 959–971.

[421] Xie, D., Li, Y., Kim, M., Pham, H. V., Tan, L., Zhang, X., and Godfrey, M. W. DocTer: documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)* (2022), pp. 176–188.

[422] Xie, T., and Notkin, D. Mutually enhancing test generation and specification inference. In *International Workshop on Formal Approaches to Software Testing* (2003), Springer, pp. 60–69.

[423] Xie, T., and Pei, J. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories* (2006), pp. 54–57.

[424] Xie, X., Ho, J. W., Murphy, C., Kaiser, G., Xu, B., and Chen, T. Y. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software (JSS) 84*, 4 (2011), 544–558.

[425] Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., and See, S. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)* (2019), pp. 146–157.

[426] Xin Tan, Kai Gao, M. Z. L. Z. An exploratory study of deep learning supply chain. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2022).

[427] Yan, X., and Han, J. gSpan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* (2002), IEEE, pp. 721–724.

[428] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering* (2006), ACM, pp. 282–291.

[429] Yang, X., Chen, J., Yedida, R., Yu, Z., and Menzies, T. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering (EMSE) 26*, 3 (2021), 56.

[430] Yang, X., Chen, J., Yedida, R., Yu, Z., and Menzies, T. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering 26*, 3 (2021), 1–24.

[431] Yang, X., and Menzies, T. Documenting evidence of a reproduction of 'is there a "golden" feature set for static warning identification?—an experimental evaluation'. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)* (2021), pp. 1603–1603.

[432] Yang, X., Yu, Z., Wang, J., and Menzies, T. Understanding static code warnings: An incremental AI approach. *Expert Syst. Appl. 167* (2021), 114134.

[433] Yang, X., Yu, Z., Wang, J., and Menzies, T. Understanding static code warnings: An incremental ai approach. *Expert Systems with Applications 167* (2021), 114134.

[434] You, W., Zong, P., Chen, K., Wang, X., Liao, X., Bian, P., and Liang, B. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017), pp. 2139–2154.

[435] Yüksel, U., and Sözer, H. Automated classification of static code analysis alerts: A case study. In *IEEE International Conference on Software Maintenance (ICSM 2013)* (2013), IEEE, pp. 532–535.

[436] Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., and Di Penta, M. How open source projects use static code analysis

tools in continuous integration pipelines. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR 2017)* (2017), IEEE, pp. 334–344.

[437] Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., and Ihara, A. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2018), IEEE, pp. 559–563.

[438] Zeller, A. Test complement exclusion: Guarantees from dynamic analysis. In *2015 IEEE 23rd International Conference on Program Comprehension* (2015), IEEE, pp. 1–2.

[439] Zeng, E., Mare, S., and Roesner, F. End user security and privacy concerns with smart homes. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)* (2017), pp. 65–80.

[440] Zeng, E., and Roesner, F. Understanding and improving security and privacy in multi-user smart homes: a design exploration and in-home user study. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 159–176.

[441] Zeng, Z., Zhang, Y., Zhang, H., and Zhang, L. Deep just-in-time defect prediction: how far are we? In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)* (2021), pp. 427–438.

[442] Zerouali, A., Mens, T., Decan, A., and De Roover, C. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering 27*, 5 (2022), 1–45.

[443] Zhang, C., Lin, X., Li, Y., Xue, Y., and Liu, Y. Apicraft: Fuzz driver generation for closed-source {SDK} libraries. In *30th {USENIX} Security Symposium ({USENIX} Security 21)* (2021), pp. 2811–2828.

[444] Zhang, C., Yang, J., Zhang, Y., Fan, J., Zhang, X., Zhao, J., and Ou, P. Automatic parameter recommendation for practical API usage. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 826–836.

[445] Zhang, J., Jiang, H., Ren, Z., Zhang, T., and Huang, Z. Enriching API documentation with code samples and usage scenarios

from crowd knowledge. *IEEE Transactions on Software Engineering* (2019).

[446] Zhang, L., He, W., Martinez, J., Brackenbury, N., Lu, S., and Ur, B. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 281–291.

[447] Zhang, M., Liu, J., Ma, F., Zhang, H., and Jiang, Y. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), IEEE, pp. 318–327.

[448] Zhang, T., and Kim, M. Automated transplantation and differential testing for clones. In *IEEE/ACM International Conference on Software Engineering (ICSE)* (2017), IEEE/ACM, pp. 665–676.

[449] Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., and Kim, M. Are code examples on an online q&a forum reliable?: a study of API misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 886–896.

[450] Zhang, T., Xu, B., Thung, F., Haryono, S. A., Lo, D., and Jiang, L. Sentiment analysis for software engineering: How far can pre-trained transformer models go? In *IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)* (2020), IEEE, pp. 70–80.

[451] Zhang, X., Sun, N., Fang, C., Liu, J., Liu, J., Chai, D., Wang, J., and Chen, Z. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)* (2021), pp. 400–412.

[452] Zheng, Q., Mockus, A., and Zhou, M. A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)* (2015), pp. 637–648.

[453] Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., and Sun, L. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 1099–1114.

[454] Zhong, H., and Mei, H. An empirical study on API usages. *IEEE Transactions on Software Engineering 45*, 4 (2017), 319–334.

[455] Zhong, H., Meng, N., Li, Z., and Jia, L. An empirical study on API parameter rules. In *2020 IEEE/ACM 42th International Conference on Software Engineering (ICSE)* (2020), pp. 899–911.

[456] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming* (2009), Springer, pp. 318–343.

[457] Zhou, H., Li, W., Kong, Z., Guo, J., Zhang, Y., Yu, B., Zhang, L., and Liu, C. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), IEEE, pp. 347–358.

[458] Zhou, J., Pacheco, M., Wan, Z., Xia, X., Lo, D., Wang, Y., and Hassan, A. E. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 705–716.

[459] Zong, P., Lv, T., Wang, D., Deng, Z., Liang, R., and Chen, K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 2255–2269.